AUTOMATED PROOFS OF OBJECT CODE FOR A WIDELY USED MICROPROCESSOR

by

YUAN YU, B.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December, 1992

Acknowledgments

First, I want to thank my thesis advisor Bob Boyer. He has been an immense source of knowledge, ideas, and inspiration. Without his support (intellectual and financial), this work would be impossible.

I would like to thank my committee members Woody Bledsoe, Don Good, Warren Hunt, Matt Kaufmann, and Bill Schelter. All contributed to making the dissertation better than I could have done on my own.

Thanks to Bill Bevier, Art Flatau, J Moore, Sakthi Subramanian, Matt Wilding, and Bill Young for many constructive discussions.

Bob Boyer, Don Good, Jim Horning, Warren Hunt, Matt Kaufmann, and Tim Leonard have carefully read earlier drafts of this dissertation, and provided many valuable comments and corrections.

Special thanks to Fay Goytowski for her meticulous reading of our MC68020 formal specification, which revealed a dozen or so errors.

I am greatly indebted to my wife Renshi for her love and support during the preparation of this dissertation.

The research described herein was supported in part by NSF Grant MIP-9017499.

Yuan Yu

The University of Texas at Austin
December, 1992

AUTOMATED PROOFS OF OBJECT CODE FOR A WIDELY USED MICROPROCESSOR

Publication No._____

Yuan Yu, Ph.D.

The University of Texas at Austin, 1992

Supervisor: Robert S. Boyer

Computing devices can be specified and studied mathematically. Formal specification of computing devices has many advantages – it provides a precise characterization of the computational model and allows for mathematical reasoning about models of the computing devices and programs executed on them. While there has been a large body of research on program proving, work has almost exclusively focused on programs written in high level programming languages. This thesis addresses the very important but largely ignored problem of machine code program proving. In this thesis we have formally described a substantial subset of the MC68020, a widely used microprocessor built by Motorola, within the mathematical logic of the automated reasoning system Nqthm, a.k.a. the Boyer-Moore Theorem Proving System. Based on this formal model, we have mechanized a mathematical theory to facilitate automated reasoning about object code programs. We then have mechanically checked the correctness of MC68020 object code programs for binary search, Hoare's Quick Sort, the Berkeley Unix C string library, and other well-known algorithms. The object code for these examples was generated using the Gnu C, the Verdix Ada, and the AKCL Common Lisp compilers.

Table of Contents

Acknowledgments i

Abstract ii

Table of Contents iii

List of Tables ix

List of Figures x

1. Introduction 1

1.1 The Thesis 1

1.2 Related Work 4

1.3 Outline of the Thesis 8

2. Formal Specification and Machine Code Verification 10

2.1 An Instruction Set Specification of the MC68020 10

2.1.1 The Interpreter Semantics 10

2.1.2 The Specification 12

2.2 Machine Code Verification 14

2.2.1 Machine Code Programs 14

2.2.2 The Correctness Statement 16

2.3 The Automated Reasoning System Nqthm 19

2.3.1 The Logic 19

2.3.2 The Theorem Prover 21

2.3.3 An Interactive Enhancement to Nqthm 22

3. The MC68020 Instruction Set Specification **23**

3.1 Basic Concepts 23

3.1.1 Natural Numbers 24

3.1.2 Integer Arithmetic 24

3.1.3 Bit Vector Arithmetic 24

3.2 The User Visible State 27

3.2.1 The Processor Status Word 28

3.2.2 The Register File 29

3.2.3 The Program Counter 29

3.2.4 The Condition Code Register 29

3.2.5 The Memory 30

3.3 Internal States and Effective Address Calculation 31

3.4 The Specification of the SUB Instruction 31

3.5 Discussion 34

4. The Mechanization of Machine Code Reasoning **37**

4.1 Integer Arithmetic 38

4.2 Bit Vector Arithmetic 38

4.3 Interpretations of Bit Vector Operations 39

4.4 Machine State Management 41

4.4.1 The Register File 42

4.4.2 The Memory 42

4.5 Interpretations of Condition Codes 43

4.6 The Interpreter Lemmas 45

5. Machine Code Program Proving **47**

5.1 The Approach 48

5.1.1 The Formulation 48

5.1.2 The Proof 50
5.2 Greatest Common Divisor 50
5.2.1 The Formalization 51
5.2.2 The Proof 52
5.2.3 A Simple Timing Analysis 53
5.3 Integer Square Root 54
5.3.1 The Formalization 55
5.3.2 The Proof 56
5.3.3 A Simple Timing Analysis 57
5.4 Binary Search 57
5.4.1 The Formalization 58
5.4.2 The Proof 60
5.4.3 A Simple Timing Analysis 61
5.5 Quicksort 61
5.5.1 The Formalization 62
5.5.2 The Proof 65
5.5.3 A Simple Stack Space Analysis 66
5.6 The Boyer-Moore Majority Voting Algorithm 67
5.6.1 The Formalization 69
5.6.2 The Proof 72
5.6.3 A Simple Timing Analysis 73

6. Issues in Machine Code Program Proving **74**

6.1 Subroutine Calling 74
6.2 Functional Parameters 79
6.3 Switch Statement 85
6.4 Embedded Assembly Code 87

7. Proving Theorems about the Berkeley Unix C String Library 90

7.1 The Berkeley Unix C String Library 91

7.1.1 The `memcpy` Function 92

7.1.2 The `memmove` Function 92

7.1.3 The `strcpy` Function 92

7.1.4 The `strncpy` Function 93

7.1.5 The `strcat` Function 93

7.1.6 The `strncat` Function 93

7.1.7 The `memcmp` Function 93

7.1.8 The `strcmp` Function 94

7.1.9 The `strcoll` Function 94

7.1.10 The `strncmp` Function 94

7.1.11 The `strxfrm` Function 95

7.1.12 The `memchr` Function 95

7.1.13 The `strchr` Function 95

7.1.14 The `strcspn` Function 96

7.1.15 The `strpbrk` Function 96

7.1.16 The `strrchr` Function 96

7.1.17 The `strspn` Function 96

7.1.18 The `strstr` Function 97

7.1.19 The `strtok` Function 97

7.1.20 The `memset` Function 98

7.1.21 The `strlen` Function 98

7.2 Proving the String Functions Correct 98

7.2.1 Proving the `memmove` Function Correct 98

7.2.2 Proving the `strstr` Function Correct 103

7.3 Programming Errors 106

7.3.1 The Bug in the Berkeley `strxfrm` Function 107
7.3.2 The Bug in the Berkeley `memmove` Function 107
7.3.3 The Bug in Plauger's `strtok` Function 108

8. Conclusions **109**
8.1 The State of the Project 109
8.2 The Significance of the Project 110
8.3 Future Work 111

A. Syntax Summary **113**

B. The Nqthm Script of The MC68020 Model and Lemma Library **115**
B.1 An Integer Sublibrary 115
B.2 A Formal Model of Some MC68020 User-Mode Instructions 140
B.3 A Lemma Library For Machine Code Program Proving 222

C. The Nqthm Script of Program Proofs **354**
C.1 Greatest Common Divisor 354
C.2 Integer Square Root 361
C.3 Binary Search 370
C.4 Quick Sort 381
C.5 Boyer-Moore Majority Voting 413
C.6 A Case Study of Subroutine Call 431
C.7 A Case Study of Switch Statement 438
C.8 A Case Study of Functional Parameters 440
C.9 A Case Study of Embedded Assembly Code 449
C.10 The `memchr` Function 451
C.11 The `memcmp` Function 457
C.12 The `memcpy` Function 463

C.13 The `memmove` Function 469
C.14 The `memset` Function 515
C.15 The `strcat` Function 521
C.16 The `strchr` Function 529
C.17 The `strcmp` Function 535
C.18 The `strcoll` Function 541
C.19 The `strcpy` Function 545
C.20 The `strcspn` Function 550
C.21 The `strlen` Function 561
C.22 The `strncat` Function 566
C.23 The `strncmp` Function 577
C.24 The `strncpy` Function 584
C.25 The `strpbrk` Function 594
C.26 The `strrchr` Function 605
C.27 The `strspn` Function 612
C.28 The `strstr` Function 621
C.29 The `strtok` Function 640
C.30 The `strxfrm` Function 665
C.31 Theorems About the String Functions 676

BIBLIOGRAPHY **696**

INDEX **700**

List of Tables

4.1 The Bcc Condition Codes 44

List of Figures

1.1 The Components of the Project 2

2.1 The User Visible Machine State 13

6.1 How to Use the Correctness of GCD in GCD3 78

Chapter 1

Introduction

Computing has not yet made its full potential contribution to provide control mechanisms for machinery because the reliability of computing systems is far from what it needs to be. One of the main reasons for our lack of confidence in computing systems is the lack of mathematical theories to forecast accurately the behaviors of computing systems. Simulation and testing can be a never-ending proposition. Only the most trivial systems can be tested exhaustively [44]. However, if computing systems are modeled in some mathematical theory, they can be studied as mathematical objects, and therefore program proving becomes possible. By program proving, we refer to a mathematical proof that a program executed according to a certain (mathematical) model of computation meets some specification.

There has been a large body of research on the topic of program proving since the very beginning of computing. Because correctness proofs can be extremely large and tedious, it seems to be difficult for humans to check (most importantly, correctly) all the proof details. To reduce the chance of mistakes in such proofs, the idea of mechanically proving the correctness of computer programs has been extensively studied (cf. the survey in [8]). It seems possible that the use of formal, mathematical, mechanical methods for ensuring the reliability of computing systems will eventually be required in safety critical applications [42]. While there has been a large body of research on program proving, work has almost exclusively focused on programs written in high-level programming languages. The problem we have been investigating in this thesis is the feasibility of mechanically verifying machine code programs executed upon existing and widely used hardware.

1.1 The Thesis

This thesis is about formally specifying and mechanically proving the correctness of machine code programs using the automated reasoning system Nqthm, also known as the Boyer-Moore Theorem Proving System. On top of Nqthm, we

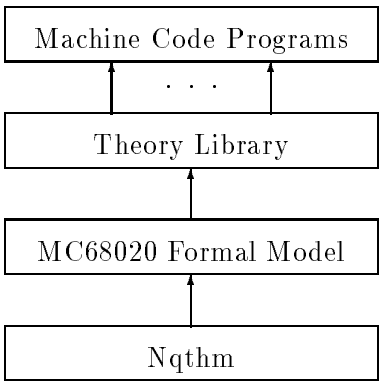

Figure 1.1: The Components of the Project

formally defined a mathematical model of the MC68020, a widely used microprocessor built by Motorola, at the instruction set level. We proceeded to mechanize a mathematical theory tailored to machine code reasoning. Finally, we studied the idea of mechanically verifying MC68020 object code produced by industrial strength optimizing high-level language compilers, e.g., the Gnu C or Verdix Ada compilers. Many such machine code programs have been successfully verified using Nqthm. An overall view of this work is shown in Figure 1.1.

Most previous work on program verification has focused on proving the correctness of programs written in high-level programming languages. Why study program proving at the machine-code level? We believe there are many good reasons for such practice.

- Work at the processor level, e.g., for a compiler correctness proof, is ultimately a necessary ingredient in program proving, if we take as our goal ensuring that programs are executed correctly on a particular processor.[1]

- Some of the most sensitive programs in the world are currently studied at the object-code level,[2] even though originally written in high-level programming languages, and for several good reasons:

[1] It is relevant to review Knuth's defense, in the Preface to *The Art of Computer Programming* [34], of his decision to present algorithms in assembler rather than in a higher-level language.

[2] For example, at several US Government agencies, including the DoD and the FAA, examiners look with great care at the machine code of critical systems.

  - Many high-level programming languages, especially those typically used in industrial practice, are not precisely specified. It is not easy, or even possible, to give the semantics of some programming language features, for example, the `volatile` type in C.
  - Some "industrial strength" compilers produce erroneous code. Until production compilers can be proved correct, we can not rely on the code they produce. Validation at the machine-code level is the only alternative.

- Programs written in high-level languages may have assembly code embedded in them, in order to communicate with external devices. But no high-level formal language semantics we have seen has made clear the semantics of the embedding of assembler instructions.

- Real-time analysis is typically done at the machine instruction level, because manufacturers often state how long an instruction takes to execute, but the definers of higher-level languages do not.

Our approach of proving theorems about object code rather than higher level programs addresses all these problems. When we are proving theorems about object code, we have no need for a formal semantics of the higher-level language in which the program may have originally been written. Any mistakes in the object code introduced by the compiler can be revealed by proving the object code correct. The semantics of embedded assembly code in programs written in high-level languages is made clear in the object code. This work also provides a formal basis to study the correctness proof of a high-level programming language compiler. Of course, we do need a formal semantics for machine code programs. But in sharp contrast to high-level programming language semantics, the formal semantics at the instruction set level, according to our experience, can be extremely clearly and rigorously defined.

It has been argued that formal verification for sequential programs has been thoroughly studied and is well understood. But from an engineering point of view, we are still very far from what we expect from formal verification. So far, we have failed to deliver any practical verification system that can be used to verify moderate-sized programs that are in *real* use, and this, in our opinion, is where we need to invest our research efforts. This thesis represents one modest step in this direction.

It is worth emphasizing that we are not advocating a return to programming in assembly or binary. Instead, our approach of studying the object code

produced by high-level programming language compilers permits a programmer to continue to program in any high-level programming language while the correctness of the program is investigated at the machine-code level.

1.2 Related Work

There is a large body of literature on the topic of program proving. This section is by no means an exhaustive survey of the whole scientific field. Rather, we simply intend to provide a detailed account of related work, with an emphasis on *mechanical* program proving.

Our work has built on the work of many others. Of historic interest is the early work of Turing [51] and von Neumann [20]. In the classic paper of Goldstine and von Neumann, we find discussed the specification and correctness proofs for fifteen programs at the machine-code level. Perhaps these were the earliest writings on program proving.

Methods for program proving have been advanced by McCarthy [39], Floyd [19], Hoare [23], and others. In the last twenty years, many research projects have focused on investigating the formal, mechanical verification of programs written in higher-level programming languages such as Pascal [25], Lisp [6], Fortran [5], and Gypsy [16]. Most of these projects are based on Floyd's inductive assertion method, and therefore in the same spirit as the early mechanical verification work of King [33]. Our work differs from all these works in that we address the correctness of programs at the machine-code level executed on a widely used processor.

It is well known that formal verification at the present time is extremely expensive. The very few cases where the cost of verification may be very well justified are applications of a safety-critical or security-related nature. But the industrial application of static code analysis has mostly been confined to proving the correctness of machine code mainly because we can not afford to trust compilers in such safety-critical applications. We feel that formal verification for programs written in high-level programming languages has not adequately addressed industrial needs, which seems to be one of the main reasons for low acceptance of formal verification in industry.

In only a very few cases does research on formal, mechanical software verification address the correctness of programs at the machine-code level. To the best of our knowledge, Maurer [37] was the first one to address one of the major

problems with machine-code verification – a machine code program may modify itself. His solution was based on Floyd's inductive assertion method. The idea was to extend each verification condition with one additional assertion asserting the contents of the program segment. Hand proofs of a few very simple machine code programs executed on toy hardware were given there. Maurer [36] later developed an IBM 370 assembly language verifier, and used it to verify some simple programs such as GCD.

Clutterbuck and Carré in [13] argued for the importance of the verification of low-level code, and, in a separate paper [26], reported their effort to analyze and verify the LUCOL assembly code modules used in the fuel control unit of the Rolls-Royce RB211-524G jet engine designed for Boeing 747-400. Like most work on software verification, this work is also based upon the use of a Floyd-style verification condition generator. The problem of assembler correctness has not been addressed in their work. Since the semantics of assembly language is normally rather complicated,[3] many restrictions have to be imposed on the assembly language, and complex annotations have to be inserted into the programs being verified.

In contrast to their work, our MC68020 instruction set model is defined in an extremely simple setting – a definition in the formal logic of the automated reasoning system being used. Our approach can be used to address the correctness of *any* machine code program that uses only instructions in the subset described by our formal model. Our proofs are completely based on this formal model. Simplicity greatly increases our confidence in our formal models and formal proofs.

Scientifically and methodologically, we have been most influenced by Bevier's Kit [2] and the 'CLI short stack' [3]. The general style of Nqthm formalization used in their work, which is also adopted in this work, is the product of over a decade of study by Boyer, Moore, and many of their students.

The work of Bevier [2, 3] is the first example we know of formal, mechanical verification of binary programs based on an operational semantics for a realistic von Neumann machine. In proving the correctness of a small operating system kernel, Bevier proved the correctness of several hundred lines of machine code produced by his own assembler for a rather realistic von Neumann machine of his own design.

[3] It is no simpler than high-level programming language semantics.

The 'CLI short stack' [3] is a small computing system consisting of a compiler, an assembler, a linker, and a gate-level design for a microprocessor that has been formally verified. In their work, Hunt proved the correctness of a gate-level design for the FM8502 microprocessor;[4] Moore proved the correctness of a compiler for the assembly-level programming language Piton targeted to the verified FM8502; Young proved the correctness of a code generator for the high-level programming language Micro-Gypsy targeted on the verified Piton. Their success inspired us to study the problem of specifying and verifying *real* programs executed on *widely used* hardware.

In contrast to our approach to machine code proof, compiler verification attempts to establish the correctness of the compiler, so that we are ensured that the compiler *always* generates correct binary code. The first example of compiler proving seems to be the McCarthy and Painter [40] proof of a compiler for expression evaluation. They prove, by hand, the correctness of an expression compiler for an idealized machine using recursion induction. Mechanical proofs of slightly varied versions of the McCarthy-Painter expression compiler were later obtained by many researchers [7].

Polak [45] seems the most ambitious compiler verification effort. Polak mechanically verified a compiler for a fairly substantial subset of Pascal. But his target machine is rather high-level and therefore unrealistic. In addition, it seemed he assumed a large collection of *unproven* lemmas which, in our opinion, should not be taken for granted.

Moore's Piton and Young's Micro-Gypsy, two components of the 'CLI short stack', are major compiler verification efforts targeted on a more realistic von Neumann architecture – the verified and fabricated FM9001. But the architecture, the programming languages, and the efficiency of compilation are still far from real world programming.

Even with such quite encouraging results, it seems that compiler verification will have little practical impact in the near future because of the sheer complexity of 'industrial strength' compilers. We believe our work may eventually contribute to compiler verification – a formal semantics for the target machine and formal reasoning at the machine-code level is a prerequisite for compiler verification.

[4] FM8502 is a von Neumann machine designed by Warren Hunt. Its successor FM9001 [24] has been successfully fabricated.

Microcode verification is closely related to our work. Among the most significant reported is the C/30 microcode verification using the State Delta Verification System(SDVS) [15]. A large majority of the C/30's instructions were proven to be correctly implemented by approximately 1000 MBB microinstructions. Hunt [52] and Cohn [14] are two major hardware design verification works involving microcode verification. Hunt reported some difficulties in microcode verification. We believe our techniques developed for machine-code verification would certainly contribute to microcode verification.

From a semantic point of view, our work is closely related to work on formal processor specification. A processor specification is a computer architecture description intended to provide a *complete* interface between processor and program. Intuitively, our formal MC68020 model is a processor specification that characterizes the behavior of MC68020 machine code programs. Leonard [35] provides an intensive survey of works on architecture specification.

Most of the work on formal processor specification has adopted the operational approach, i.e., the semantics of the machine is given by an abstract interpreter that describes how the state vector changes as the computation progresses [39]. Iverson proposed to use APL as an architecture specification language [27]. APL was later used to provide a complete formal description of the IBM system/360 architecture [17]. In the early 1970's, Bell and Newell introduced the specification language ISP [1]. ISP has been widely used to specify various computer architectures [47], most recently the SPARC [49]. ISP is one of the very few architecture specification languages that has achieved widespread use.

While the APL and ISP work was primarily motivated by providing a better notation for computer architecture description, McCarthy was perhaps the first one to connect the interpreter semantics with mathematical reasoning about programs. Our work is clearly along the same line McCarthy outlined in [39, 38].

Processor specification has been intensively studied in the hardware verification communities where the main goal is the formal verification that a hardware design meets its architectural specification. Gordon [22] introduced LCF-LSM, and demonstrated its use in specifying and verifying Gordon's machine. Hunt used the Boyer-Moore logic to specify and verify the FM8502 microprocessor. Cohn [14] used the HOL system to specify and verify the Viper microprocessor. Most of the architectures studied here are either "on paper" or novel.

A group of researchers at Oxford have been working on formal processor specification using the formal specification language Z. They have specified the Motorola M6800 architecture [4], parts of the Motorola M68000 architecture [46], and the Inmos transputer architecture [18]. It seems that they have primarily focused on issues in formal specification. Little has been reported on any formal verification effort in their work.

1.3 Outline of the Thesis

The main product of this dissertation is a powerful proof system built on top of Nqthm that allows us to reason about machine code programs for the Motorola MC68020 microprocessor. In order to give the reader a clear picture of this project, we provide, in their verbatim form, our formal specification for the MC68020 microprocessor and our lemma library for machine code reasoning in Appendix B. The complete script of all the program proofs presented in this dissertation is also given in Appendix C. The following is an outline of this dissertation.

In Chapter 2 we outline our general approach to formal specification and verification, and give a nontechnical account of this project. For uninitiated readers, we also provide a very brief introduction to the Boyer-Moore automated reasoning system.

We thus start to present our MC68020 formal specification in Chapter 3. The main contribution here is a user's behavioral-level model for a substantial subset of the MC68020 that is amenable to mathematical reasoning in a computational logic. In this chapter, we illustrate our MC68020 formal model by taking a tour all the way through the formalization of one particular instruction.

Based on the formal model defined in Chapter 3, we have developed a mathematical theory tailored to mechanically prove the correctness of machine code programs. The theory is mechanized in the Boyer-Moore theorem proving system as a library of derived lemmas. In Chapter 4, we discuss our experience in developing this lemma library.

With the MC68020 formal model and the mathematics so developed, we investigate formal reasoning about machine code programs. Chapter 5 consists of five specific examples that have helped us to sharpen our understanding of reasoning about machine code. We describe in this chapter the specification and verification of a few programs we have mechanically verified.

The semantics of some high-level programming language features have long posed great challenges to program verification. It is interesting to see how their semantics are recast into a different, but clearly understood world of a single addressing space. In Chapter 6, we use a few simple program examples to illustrate how we deal with those programming features at the machine-code level.

To demonstrate the usefulness of our system, we describe in Chapter 7 the formal verification of the Berkeley implementation of the ANSI/ISO C String Library [53, 28].[5] Three programming errors were revealed in the process of our verification. Two were in the Berkeley Unix C string library.[6] The other one was in Plauger's book *The Standard C Library* [44].[7]

In the final chapter, we summarize our main results and contributions, consider the possible applications to our methodology, and speculate on future research directions.

[5] The ANSI and ISO C Standards are essentially identical.

[6] One error was undetected when we reported it to the author [50], and will be corrected for the release of BSD4.4. The other one was fixed by the author [50] about one year ago.

[7] This error had been detected by the author by the time we reported it to him [43].

Chapter 2

Formal Specification and Machine Code Verification

There have been intensive discussions about what formal verification can or can not do. We feel it is both desirable and necessary to make clear at the outset what we have done in this thesis. This chapter is intended to give a characterization of this work before we dive into the technical details of the MC68020 formal model and machine code program proving.

We plan to focus on two of the most fundamental issues in this thesis – our MC68020 formal model and our correctness criteria for machine code programs. This chapter is divided into three sections. In the first section, we discuss how we have formalized the MC68020 instruction set in the Nqthm logic, and give an exact account of the subset of the MC68020 instructions formalized in our model, which, in turn, represents the class of machine code programs we are able to deal with in program proving. In the second section, we first give an example of the form of machine code programs (a list of natural numbers) we have studied in this thesis. We then define the meaning of a correct machine code program in our formalism. We also discuss in this section the assumptions we must make to connect our proofs with the real world. Lastly, we include a section in this chapter to introduce the automated reasoning tool Nqthm being used in this work.

2.1 An Instruction Set Specification of the MC68020

The MC68020 microprocessor is modeled as an abstract finite state machine with an interpreter semantics. This section introduces our modeling approach, and provides an overview of our MC68020 model.

2.1.1 The Interpreter Semantics

An abstract finite state machine is defined by a specification of the machine state and a specification of a state transition relation on machine states. The machine

state is specified as a vector of state components. The state transition relation is defined by an interpreter function acting on machine states.

In the Nqthm logic, the machine state is represented by a finite list with the state components as its elements. The MC68020 machine state in our formalism, for example, is simply defined to be a list of five components.

DEFINITION:
mc-state (*status*, *regs*, *pc*, *ccr*, *mem*) = list (*status*, *regs*, *pc*, *ccr*, *mem*)

Intuitively, mc-state (*status*, *regs*, *pc*, *ccr*, *mem*) represents a machine state with processor status word *status*, register file *regs*, program counter *pc*, condition code register *ccr*, and memory *mem*.

The interpreter function is then defined as a recursive function of the form 'stepn': $S \times O \rightarrow S$, where S is a set of machine states and O a set of oracles for a machine. This function 'stepn' models the behavior of a machine over a finite but arbitrary time span. The two roles of an oracle are to determine the finite time span of the operation of a machine invocation, and to introduce non-deterministic state changes into a machine that include communication with other machines.[1]

In the simple case that the set of natural numbers N is used as the oracle set, the interpreter models a machine whose behavior is determined completely by its states. Our MC68020 interpreter is defined in this simple setting.

DEFINITION:
stepn (*s*, *n*)
= **if** mc-haltp (*s*) ∨ (*n* ≃ 0) **then** *s*
else stepn (stepi (*s*), *n* − 1) **endif**

Intuitively, stepn (*s*, *n*) returns the machine state produced by running the machine n instructions with the initial state s. stepi (*s*) in the above definition is the *single-stepper* that advances the machine by one instruction according to the current state s.

This interpreter semantics describes the meaning of abstract machines in an intuitive and natural way. It can be easily understood by a wide range of computer

[1] This paragraph was taken from a paragraph of [3] by permission, and modified in the context of this thesis.

professionals. For example, our MC68020 interpreter may be simply viewed as an architectural simulator for an MC68020 microprocessor defined in a formal logic.

We will leave all the further formal details of our MC68020 formalization to Chapter 3.

2.1.2 The Specification

There are two main goals of our MC68020 formal specification. First, we intend to provide a formal model to reflect as closely as possible the user's manual view of the MC68020 [41]. Second, this formal model should be amenable to automated reasoning. The specification has been written with these two main goals in mind.

We have formalized most of the user programming model of the MC68020 microprocessor. However, we have not yet specified the supervisor level of the MC68020. Any exception caused by a user program simply halts our formalized machine. Figure 2.1 provides an informal, two dimensional picture of the user 'programming model' for the MC68020, as described in [41]. This model has 16 32-bit general-purpose registers (8 data registers, D0-D7, and 8 address registers, A0-A7), a 32-bit program counter PC, and an 8-bit condition code register, CCR. The address register A7 is also used as the user stack pointer (USP). The 5 least significant bits in CCR are condition codes for carry, overflow, zero, negative, and extend. Our model is the only part of the state of an MC68020 that a user program can read or write under our formal semantics. Not present in our model are such arcane actualities as the instruction cache, memory management, and the supervisor stack.

Our specification consists of about 80% of all the user available instructions and all eighteen MC68020 addressing modes. Most of the instructions we have left unspecified have some *undefined* effects on the machine state. For example, some of the condition codes of the instruction `CMP2` are described as *undefined* [41]. We have deliberately excluded such instructions from our specification. Fortunately, these instructions constitute only a small portion of the instruction set, and most of them are rarely used.[2] We summarize below the instructions we have formalized.

The instructions of the MC68020 instruction set are classified into ten categories according to their functions [41].

[2] We have not yet encountered such instructions in the machine code programs we have studied.

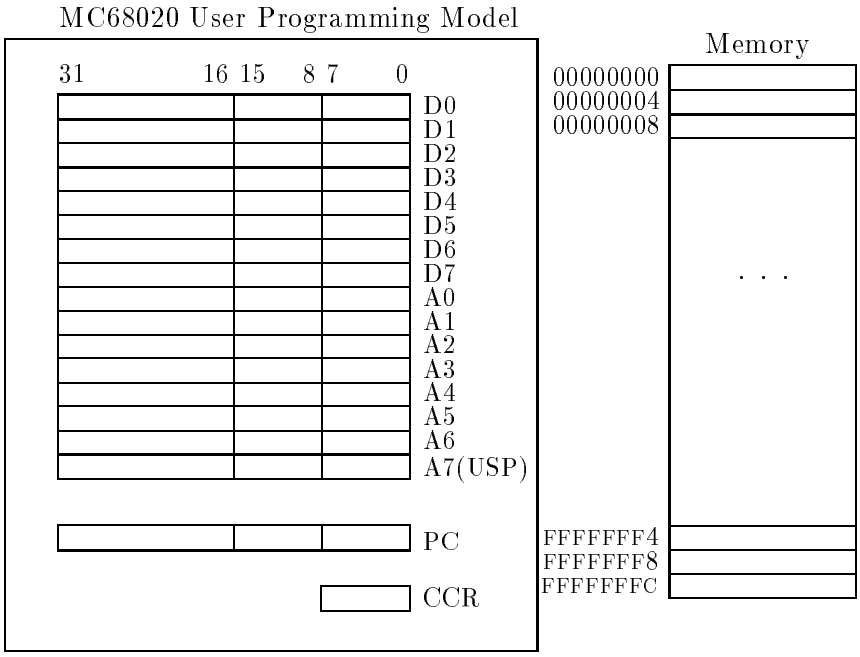

Figure 2.1: The User Visible Machine State

1. *Data Movement.* We have included all the data movement instructions: `EXG, LEA, LINK, MOVE, MOVEA, MOVEM, MOVEP, MOVEQ, PEA`.

2. *Integer Arithmetic.* We have included all the integer arithmetic instructions except `CMP2`: `ADD, ADDA, ADDI, ADDQ, ADDX, CLR, CMP, CMPA, CMPI, CMPM, DIVS, DIVSL, DIVU, DIVUL, EXT, EXTB, MULS, MULSL, MULU, MULUL, NEG, NEGX, SUB, SUBA, SUBI, SUBQ, SUBX`.

3. *Logical Operations.* We have included all the logical instructions: `AND, ANDI, EOR, EORI, NOT, OR, ORI, TAS, TST`

4. *Shift and Rotate.* We have included all the shift and rotate instructions: `ASL, ASR, LSL, LSR, ROL, ROR, ROXL, ROXR, SWAP`.

5. *Bit Manipulation.* We have included all the bit manipulation instructions: `BCHG, BCLR, BSET, BTST`.

6. *Bit Field.* We have included all the bit field instructions: `BFCHG, BFCLR, BFEXTS, BFEXTU, BFFFO, BFINS, BFSET, BFTST`.

7. *Binary coded decimal.* None of the binary coded decimal instructions has been considered.

8. *Program Control.* We have included all the program control instructions except the pair of instructions `CALLM` and `RTM`: `Bcc, DBcc, Scc, BRA, BSR, JMP, JSR, NOP, RTD, RTR, RTS`.

9. *System Control.* Only 5 of the 21 system control instructions are formalized: `ANDI to CCR, EORI to CCR, MOVE from CCR, MOVE to CCR, ORI to CCR`.

10. *Multiprocessor.* None of the multiprocessor instructions have been considered.

Our formal specification is about 128,000 bytes long, which, when printed, takes up approximately 80 pages of text. It consists of 569 function definitions in the Nqthm logic. The full text of this formal specification is given in Appendix B. The semantics of any machine code program written in this subset of MC68020 instructions is given formally by our MC68020 model.

Remark. The complexity of this model is not particularly surprising to us. Rather, we believe the complexity is intrinsic for a CISC architecture like the MC68020, and we are quite amazed and satisfied with the effectiveness of mathematical reasoning with this formal model.

2.2 Machine Code Verification

The verification approach we have taken in this work is simple and straightforward – we reason about MC68020 machine code programs based solely on the MC68020 formal model described above. The correctness of any machine code program written in our formalized subset of MC68020 instructions can be addressed, at least theoretically, by our verification system. In this section, we investigate what we mean by the correctness of machine code programs in our formalism. In particular, we present the exact form of objects (machine code programs) subject to verification, and discuss in general our correctness statement about machine code program.

2.2.1 Machine Code Programs

We have primarily focused on proving the correctness of object code generated by "industrial strength" high level language compilers. Our method of verifying optimized compiled code is very simple. We, for example, compile C programs using the Gnu C compiler, extract the machine code program using the Gnu debugger, and finally prove the machine code correct using our proof system developed in Nqthm.

To make it more concrete, we illustrate the idea with the following simple C program that computes the greatest common divisor of two nonnegative integers by Euclid's algorithm. This algorithm has been well studied in the program verification literature.

```
/* computes the greatest common divisor by Euclid's algorithm */
gcd(int a, int b)
{
  while (a != 0){
    if (b == 0) return (a);
    if (a > b)
      a = a % b;
    else b = b % a;
  };
  return (b);
}
```

We start with a file, say `gcd.c`, consisting of the C function `gcd` shown above. We compile `gcd.c` using the Gnu C compiler `gcc`, and then obtain the assembly code (for human's consumption) and the binary (for Nqthm's consumption) using the Gnu debugger `gdb`. The following session was from a Sun3-280.

```
rascal% gcc -g -O gcd.c
rascal% gdb -q a.out
Reading symbol data from /xy0e/u/all/yu/a.out...done.
(gdb) x/22i gcd
Reading in symbols for gcd.c...done.
0x22a0 <gcd>:           linkw fp,#0
0x22a4 <gcd+4>:         moveml d2-d3,sp@-
0x22a8 <gcd+8>:         movel fp@(8),d2
0x22ac <gcd+12>:        movel fp@(12),d3
0x22b0 <gcd+16>:        tstl d2
0x22b2 <gcd+18>:        beq 0x22d0 <gcd+48>
0x22b4 <gcd+20>:        tstl d3
0x22b6 <gcd+22>:        bne 0x22bc <gcd+28>
0x22b8 <gcd+24>:        movel d2,d0
0x22ba <gcd+26>:        bra 0x22d2 <gcd+50>
0x22bc <gcd+28>:        cmpl d2,d3
0x22be <gcd+30>:        bge 0x22c8 <gcd+40>
0x22c0 <gcd+32>:        divsll d3,d0,d2
0x22c4 <gcd+36>:        movel d0,d2
0x22c6 <gcd+38>:        bra 0x22b0 <gcd+16>
0x22c8 <gcd+40>:        divsll d2,d0,d3
0x22cc <gcd+44>:        movel d0,d3
0x22ce <gcd+46>:        bra 0x22b0 <gcd+16>
0x22d0 <gcd+48>:        movel d3,d0
0x22d2 <gcd+50>:        moveml fp@(-8),d2-d3
0x22d8 <gcd+56>:        unlk fp
0x22da <gcd+58>:        rts
```

```
(gdb) x/60ub gcd
<gcd>:     78     86     0      0      72     231    48     0
<gcd+8>:   36     46     0      8      38     46     0      12
<gcd+16>:  74     130    103    28     74     131    102    4
<gcd+24>:  32     2      96     22     182    130    108    8
<gcd+32>:  76     67     40     0      36     0      96     232
<gcd+40>:  76     66     56     0      38     0      96     224
<gcd+48>:  32     3      76     238    0      12     255    248
<gcd+56>:  78     94     78     117
(gdb) quit
rascal%
```

The above 60 unsigned integers `(78, 86, ..., 117)` are the bytes in the memory for the *relocatable* machine code program of the C function `gcd`. These numbers are the objects subject to verification, and therefore are the inputs to our verification system. A proof that these numbers do "compute" the greatest common divisor of two nonnegative integers will be presented in full detail in Chapter 5.

2.2.2 The Correctness Statement

Before we explain our correctness criteria for machine code programs, let us first examine the assumptions made when we attempt to connect our correctness theorems to the real world.

The Assumptions Under what assumptions does our program proving correspond to the real behavior of the program executed on a real MC68020 microprocessor? We believe this relevance issue should be addressed at a very early stage of any verification work, especially if we attempt to use our theory to predict the behavior of programs rather than merely to manipulate symbols in some formal mathematical logic.

The first assumption is, of course, the soundness of the underlying automated reasoning system being used. In our case, we assume that the Nqthm system does not prove false 'theorems'. To our knowledge, Nqthm has been by far the most reliable automated reasoning tool available.[3] Mathematical models are approximations to the real physical worlds. So is our model for the MC68020. The second assumption is that our MC68020 model accurately reflects the behavior of a *real*

[3]In its almost twenty years existence and intensive uses, only one soundness error was found in the released versions of Nqthm.

MC68020 microprocessor. While we can not prove the validity of our MC68020 specification, we have invested a great deal of effort to increase our confidence in this model. Boyer and Goytowski have read the specification very carefully. We defined the model in such a way that it is consistent relative to the consistency of the Nqthm logic. Our MC68020 model is executable, which allowed us to use the conventional simulation and testing methods to study the model. Ken Albin at Computational Logic, Inc. has been working on a testing suite for both Hunt's FM9001 and our MC68020 models.

Under these two assumptions, the program, when executed in an ideal environment, should behave the same as whatever the proved correctness theorem asserts. By ideal execution environment, we mean things like no power outage, no hardware failure, no interference from the operating system, etc.

The Correctness Statement The correctness statement for a machine code program should fully characterize the effects on the machine state of its execution. The most important requirement of the correctness statement is that it be 'context-free' and 'universally' applicable, so that we can reuse theorems about a program in other proofs. Our correctness theorem at the object-code level is more elaborate than one for a program written in a higher-level language. This is not particularly surprising, since our theorems assert more properties about a program than would higher-level program proving, because we have a more complicated model of the machine state.

In general, the theorem we prove for every machine code program has the following form.

$$\text{p-statep}(s) \Rightarrow \text{p-req}(s, \text{stepn}(s, \text{p-t}(s)))$$

Informally, the theorem says that, if the precondition p-statep(s) is satisfied, the properties specified by the relation *p-req* about the initial state s and the resulting state stepn(s, p-t(s))), obtained by running the machine p-t(s) instructions from the initial state s, holds. Note that this theorem is completely based on the semantics given by *stepn*.

The precondition *p-statep* and the requirement *p-req* in the above formula deserve further explanation. The precondition p-statep(s) imposes certain conditions on the initial state to ensure the correct execution of the program. The conditions imposed in our formalism are given as follows, informally.

- The machine state s is in the user mode.

- The program counter of s is even.[4]
- The program is stored in the memory of s, starting from the address pointed to by the program counter of s.
- There is "enough" memory space available, e.g., the stack has "enough" space available for the execution of the program.
- The program arguments satisfy certain program specific properties, e.g., they are placed in the right places on the stack.

The requirement *p-req* asserts some important properties of the program. In our formalism, we prove the following properties of programs.

1. The resulting machine state is 'normal,' e.g., no read or write to unavailable memory occurred, no illegal instruction was executed.
2. The program counter in the resulting state is set to the 'right' location.
3. The correct results are stored in the "right" place.
4. The register file is properly managed, e.g., A7, the User Stack Pointer, is set to the "right" location, and some registers used as temporary storage are restored to their original values.
5. The program accesses and changes only the intended portion of memory.

For readers who are familiar with program verification, requirements 1 and 2 state the program's termination property, and requirement 3 states the program's "partial correctness".

All the machine code programs presented in this thesis have been mechanically proved *correct* according to the above standards. Such a correctness theorem for a program can be used as a 'blackbox' for larger proofs where the program is a subprogram.

[4] The MC68020 microprocessor requires the program counter be aligned to a word boundary.

2.3 The Automated Reasoning System Nqthm

We briefly review the automated reasoning system Nqthm, also known as 'the Boyer-Moore Theorem Prover'. Detailed knowledge of Nqthm is unnecessary for those who are happy enough with the informal paraphrases of the formulas in the remainder of this thesis. For a thorough and precise description of the Nqthm logic, we refer the reader to the rigorous treatment in [9], especially Chapter 4, in which the logic is precisely defined.

Nqthm is a Common Lisp program for proving mathematical theorems. Since *A Computational Logic* [7] was published in 1979, Nqthm has been used by many users to check proofs of over 16,000 theorems from many areas of number theory, proof theory, and computer science. An extensive partial listing may be found in [9, pages 5–9]. In the body of this dissertation, we use a conventional syntax rather than the official Lisp-like syntax of Nqthm. The translation between the conventional syntax and the official Lisp-like syntax is discussed in [12], and given in Appendix A.

2.3.1 The Logic

The logic of Nqthm is a quantifier-free first order logic with equality. The basic theory includes axioms defining the following:

- the Boolean constants **t** and **f**, corresponding to the true and false truth values.
- equality. $x = y$ is **t** or **f** according to whether x is equal to y.
- an if-then-else function. **if** x **then** y **else** z **endif** is z if x is **f** and y otherwise.
- the Boolean arithmetic operations $x \wedge y$, $x \vee y$, $\neg x$, $x \Rightarrow y$, and $x \Leftrightarrow y$.

The logic of Nqthm contains three 'extension' principles under which the user can introduce new concepts into the logic with the guarantee of consistency.

- *The Shell Principle* allows the user to add axioms introducing 'new' inductively defined 'abstract data types.' Natural numbers, ordered pairs, and symbols are axiomatized in the logic by adding shells:
  - *Natural Numbers.* The nonnegative integers are built from the constant `0` by successive applications of the constructor function 'add1'. The function

'numberp' recognizes natural numbers. The function 'sub1' returns the predecessor of a non-`0` natural number. $x \in \mathbf{N}$ abbreviates numberp(x).

- *Symbols.* The data type of symbols, e.g., `'running`, is built using the primitive constructor 'pack' and `0`-terminated lists of ASCII codes. The symbol `'nil`, also abbreviated **nil**, is used to represent the empty list.
- *Ordered Pairs.* Given two arbitrary objects, the function 'cons' builds an ordered pair of these two objects. The function 'listp' recognizes ordered pairs. The functions 'car' and 'cdr' return the first and second component of such an ordered pair. Lists of arbitrary length are constructed with nested pairs. Thus list($arg_1, \ldots, arg_n$) is an abbreviation for cons(arg_1, ..., cons(arg_n, **nil**)).

- *The Definitional Principle* allows the user to define new functions in the logic. For recursive functions, there must be an ordinal measure of the arguments that can be proved to decrease in each recursion, which, intuitively, guarantees that one and only one function satisfies the definition. Many functions are added as part of the basic theory by this definitional principle. For example, we define for the natural numbers these familiar operations: $i + j$, $i - j$, $i < j$, $i * j$, $i \div j$, i **mod** j, and $\exp(i, j)$. $i \simeq \mathtt{0}$ returns **f** if and only if i is a positive integer.

- *The Constraint Principle* allows the user to introduce and constrain, rather than completely define, new function symbols in the logic. To avoid introducing any new inconsistency into the logic, the user is required to prove that the proposed constrains are satisfiable by providing some already defined "witness" functions for the new function symbols.

The rules of inference of the logic consist of:

1. *Propositional Calculus with Equality:* All tautologies and equality axioms are theorems.

2. *Induction Principle:* Each instance of an axiom schema for well-founded induction up to ε_0 is a theorem.

3. *Instantiation:* Any instance of a theorem is a theorem.

2.3.2 The Theorem Prover

The Nqthm theorem prover is a mechanization of the preceding logic. It takes as input a term in the logic, and repeatedly transforms it in an effort to reduce it to non-**f**. Many heuristics and decision procedures are implemented as part of the transformation mechanism.

The theorem prover is fully automatic in the sense that once a proof attempt has started, the system accepts no advice or directives from the user. The only way the user can interfere with the system is to abort the proof attempt. However, on the other hand, the theorem prover is interactive: the system may gain more proving power through its data base of lemmas, which have already been formulated by the user and proved by the system. Each conjecture, once proved, may be converted into some 'rules' which influence the prover's action in subsequent proof attempts.

The commands to the theorem prover include those for defining new functions, proving lemmas, and adding shells, etc. In this thesis, we only use the following four commands. The first two are the most often used.

- To admit a new function under the definitional principle, we invoke

  DEFINITION: fn-name (*args*) = *body*

- To initiate a proof attempt for the conjecture *statement*, naming it lemma-name, we invoke

  THEOREM: lemma-name
  statement

- To introduce an incomplete definition *term* under the constraint principle, we invoke

  CONSERVATIVE AXIOM: name
  axiom

- To initiate a proof attempt for the conjecture *statement*, using functional instantiation, we invoke

  THEOREM: lemma-name
  statement

- To introduce a quantified first-order formula *form*,[5] we invoke

[5] This is an extension to Nqthm by Matt Kaufmann, which is not documented in [9]. See [30] for details.

DEFINITION: fn-name (*args*) $\Leftrightarrow$ *form*

Typically, the checking of difficult theorems by Nqthm requires extensive user interaction. The behavior of the prover is influenced profoundly by the user's actions. The user first formalizes the problem to be solved in the logic. The formalization may involve many concepts and so the specification may be very complicated. The user then leads the theorem prover to a proof of the goal theorem by proving lemmas that, once proved, control the search for additional proofs. Typically, the user first discovers a hand proof, identifies the key steps in the proof, formulates them as a sequence of lemmas, and gets each checked by the prover. Successful users of the system must know how to prove theorems in the logic and must understand how the system interprets them as rules.

2.3.3 An Interactive Enhancement to Nqthm

While our work is completely built on top of Nqthm, we have found Kaufmann's PC-Nqthm system [29] a valuable tool for debugging Nqthm proofs. This system is fully integrated with Nqthm. Thus, the user can give commands at a low level (such as deleting a hypothesis) or at a high level (such as calling Nqthm).

As with a variety of proof-checking systems, PC-Nqthm is goal-directed: a proof is completed when the main goal and all subgoals have been proved. A notion of *macro commands* lets the user create compound commands, in the same spirit of the tactics and tacticals of LCF [21]. An interactive proof is complete when all goals have been proved. It is PC-Nqthm's low level features that help us understand when and why a goal fails.

Chapter 3

The MC68020 Instruction Set Specification

We have formally specified a substantial subset of the instruction set of the MC68020 microprocessor. The formal specification can be viewed as behavioral-level simulator in a formal logic, one intended to reflect the MC68020 microprocessor correctly and, in the meantime, to be amenable to mathematical reasoning. The main objective of this chapter is to communicate precisely this formal, mathematical formalization of the MC68020. As a result, we hope to convince the reader that our MC68020 formal specification appropriately models the behavior of the real chip at a certain abstract level.

The organization of this chapter requires some explanation. After formalizing in the Nqthm logic the basic concepts – the natural numbers, the integers, and the bit vectors, we describe our formalization of the machine states and the state components. We then discuss the specification of the MC68020 addressing modes. With all the necessary pieces in place, we will then investigate the formalization of one specific instruction. We start from the very top level of the specification, and descend down to the smallest details, described in the few preceding sections. We have chosen to study one of the most familiar instructions: the SUB instruction, which reflects our general modeling approach to all instructions in our MC68020 model. Finally, we conclude with some remarks about some of the interesting issues that have come up in the specification.

The entire MC68020 formal specification is given in Appendix B.2.

3.1 Basic Concepts

This section details how we formalize basic natural number, integer, and bit vector arithmetic in the Nqthm logic. Bit vector is the only type of object manipulated at the instruction-set level. Integer arithmetic, which has its use in program proving, is not used in this chapter.

3.1.1 Natural Numbers

Natural numbers are axiomatized in the Nqthm logic with Peano's axioms. Many common functions on natural numbers such as $x + y$, $x - y$, $x * y$, x **mod** y, $x \div y$, and $x < y$ have been built into the 'Ground-Zero' logic of the Nqthm system by its implementors. The only two other functions we need in our specification are the exponential function $\exp(x, y)$ and the logarithmic function $\log(b, x)$, which are defined as follows:

```
DEFINITION:
exp(x, y)
= if y ≃ 0 then 1
    else x * exp(x, y - 1) endif
```

```
DEFINITION:
log(b, x)
= if (b ≃ 0) ∨ (b = 1) then 0
    elseif x < b then 0
    else 1 + log(b, x ÷ b) endif
```

The reader may find the definitions of the built-in functions in [9].

3.1.2 Integer Arithmetic

The Nqthm logic adds the integers almost as an afterthought – all the integer operations have to be defined by the user. The integer functions we have defined in the Nqthm logic are 'integerp', 'iplus', 'idifference', 'itimes', 'iremainder', 'iquotient', and 'ilessp', which simply are the integer counterparts of those natural number functions in the preceding subsection.

Since the meanings of these functions are quite intuitive, we will not give their definitions here. The reader may find their definitions in Appendix B.2.

3.1.3 Bit Vector Arithmetic

Bit vectors are represented as natural numbers in our formalism. For example, the content of the program counter is represented as a nonnegative integer with range between 0 and $2^{32} - 1$, inclusive. Each of the operations on bit vectors is therefore formalized as some sort of operation on nonnegative integers. The decision to use natural number representation was not easy to make. Finite lists, for example, seemed an equally good representation for bit vectors, and have been used

successfully in hardware design verification [52]. In fact, we tried to use the finite list representation in our early version of the MC68020 specification. But we soon found it awkward for machine code program proving. The choice of representation should take into account the often much more difficult task of automated reasoning.

Next, let us see how we define the basic bit vector arithmetic. We present the definitions of all the operations because of their frequent appearance in the subsequent exposition.

The following functions define basic bit field manipulation operations that have their utility throughout the specification.

DEFINITION: $\mathrm{bcar}(x) = (x \ \mathbf{mod}\ 2)$

DEFINITION: $\mathrm{bcdr}(x) = (x \div 2)$

DEFINITION: $\mathrm{head}(x, n) = (x \ \mathbf{mod}\ \exp(2, n))$

DEFINITION: $\mathrm{tail}(x, n) = (x \div \exp(2, n))$

DEFINITION: $\mathrm{bitn}(x, n) = \mathrm{bcar}(\mathrm{tail}(x, n))$

DEFINITION: $\mathrm{mbit}(x, n) = \mathrm{bitn}(x, n - 1)$

DEFINITION: $\mathrm{bits}(x, i, j) = \mathrm{head}(\mathrm{tail}(x, i), 1 + (j - i))$

DEFINITION:
$\mathrm{setn}(x, n, c)$
$=$ **if** $n \simeq 0$ **then** $\mathrm{fix\text{-}bit}(c) + (2 * \mathrm{bcdr}(x))$
else $\mathrm{bcar}(x) + (2 * \mathrm{setn}(\mathrm{bcdr}(x), n - 1, c))$ **endif**

DEFINITION: $\mathrm{app}(n, x, y) = (\mathrm{head}(x, n) + (y * \exp(2, n)))$

Intuitively, 'head' returns the bit vector of the first n bits of x; 'tail' returns the bit vector obtained by discarding the first n bits of x; 'bcar' and 'bcdr' are simply the special cases of *head* and *tail* with $n = 1$; 'bitn' returns the nth bit of the bit vector x; 'mbit' is simply a special case of *bitn*, returning the most significant bit of x; 'bits' returns the bit vector consisting of bits i through j of x; 'setn' sets the nth bit of the bit vector x to c. 'app' returns the bit vector obtained by concatenating x and y.

The following functions formalize familiar logical functions that are used to specify the corresponding MC68020 logical instructions.

```
DEFINITION:  lognot(n, x) = ((exp(2, n) - head(x, n)) - 1)

DEFINITION:
logand(x, y)
= if (x ≃ 0) ∨ (y ≃ 0) then 0
  else b-and(bcar(x), bcar(y))
       + (2 * logand(bcdr(x), bcdr(y))) endif

DEFINITION:
logor(x, y)
= if x ≃ 0 then fix(y)
  elseif y ≃ 0 then fix(x)
  else b-or(bcar(x), bcar(y))
       + (2 * logor(bcdr(x), bcdr(y))) endif

DEFINITION:
logeor(x, y)
= if (x ≃ 0) ∧ (y ≃ 0) then 0
  else b-eor(bcar(x), bcar(y))
       + (2 * logeor(bcdr(x), bcdr(y))) endif
```

The functions 'lognot', 'logand', 'logor', and 'logeor' model the logical functions 'not', 'and', 'or', and 'eor', respectively.

The following functions formalize bit-vector addition, subtraction, and sign-extension, which have their use in specifying the corresponding MC68020 instructions and effective address calculation.

```
DEFINITION:  adder(n, c, x, y) = head(c + x + y, n)

DEFINITION:  add(n, x, y) = head(x + y, n)

DEFINITION:
subtracter(n, c, x, y) = adder(n, b-not(c), y, lognot(n, x))

DEFINITION:
sub(n, x, y) = head(y + (exp(2, n) - head(x, n)), n)

DEFINITION:
ext(n, x, size)
= if n < size
  then if b0p(bitn(x, n - 1)) then head(x, n)
       else app(n, x, exp(2, size - n) - 1) endif
  else head(x, size) endif
```

The function 'ext' sign-extends the bit vector x, with length n, into a bit vector with length *size*.

Finally, we formalize those bit vector shift and rotate operations that are mainly used in specifying the MC68020 shift and rotate instructions.

```
DEFINITION: lsl (len, x, cnt) = head (x * exp (2, cnt), len)

DEFINITION: asl (len, x, cnt) = head (x * exp (2, cnt), len)

DEFINITION: lsr (x, cnt) = tail (x, cnt)

DEFINITION:
asr (n, x, cnt)
= if x < exp (2, n - 1) then tail (x, cnt)
  elseif n < cnt then exp (2, n) - 1
  else app (n - cnt, tail (x, cnt), exp (2, cnt) - 1) endif

DEFINITION:
rol (len, x, cnt)
= let n be cnt mod len
  in
  app (n, tail (x, len - n), head (x, len - n)) endlet

DEFINITION:
ror (len, x, cnt)
= let n be cnt mod len
  in
  app (len - n, tail (x, n), head (x, n)) endlet
```

As suggested by their names, the functions 'lsl' and 'lsr' formalize logical shift; the functions 'asl' and 'asr' formalize arithmetic shift; the functions 'rol' and 'ror' formalize logical rotate.

3.2 The User Visible State

As briefly mentioned in Chapter 2, we formalize a user visible machine state as a list of five components that have their intuitive meanings as the processor status word, the register file, the program counter, the condition code register, and the memory, respectively.

```
DEFINITION:
mc-state (status, regs, pc, ccr, mem) = list (status, regs, pc, ccr, mem)

DEFINITION: mc-status (s) = car (s)

DEFINITION: mc-rfile (s) = cadr (s)

DEFINITION: mc-pc (s) = head (caddr (s), L)
```

DEFINITION: mc-ccr (s) = head (cadddr (s), B)

DEFINITION: mc-mem (s) = caddddr (s)

The function 'mc-state' constructs a machine state using its five arguments; the other functions 'mc-status', 'mc-rfile', 'mc-pc', 'mc-ccr', and 'mc-mem' are accessors to the five different components of a given machine state. There are four constants B, W, L, and Q in the logic to define the sizes of byte, word, longword, and quadword of the MC68020, respectively.

In the next five subsections, we describe the formalization of each of the five state components.

3.2.1 The Processor Status Word

The processor status word is either the symbol `'running` or one of the following symbols indicating some error message if an exception occurs. This status field is not actually present in any MC68020 chip. Rather, it is the artifice of our state formalization by which we indicate that an actual error has arisen or that an aspect of the MC68020 not defined in our formalization has been encountered during execution.

DEFINITION: READ-SIGNAL = `'read_unavailable_memory`

DEFINITION:
WRITE-SIGNAL = `'write_rom_or_unavailable_memory`

DEFINITION:
RESERVED-SIGNAL
= `'motorola_reserved_for_future_development`

DEFINITION: PC-SIGNAL = `'pc_outside_rom`

DEFINITION: PC-ODD-SIGNAL = `'pc_at_odd_address`

DEFINITION:
MODE-SIGNAL
= `'illegal_addressing_mode_in_current_instruction`

We say the machine state is *normal* if its status is `'running`.

3.2.2 The Register File

The register file is represented as a list of nonnegative integers, where the first eight represent the data registers D0 - D7 and the second eight represent the address registers A0 - A7.

```
DEFINITION:
read-rn (oplen, rn, regs) = head (get-nth (rn, regs), oplen)

DEFINITION:
write-rn (oplen, value, rn, regs)
=  put-nth (replace (oplen, value, get-nth (rn, regs)), rn, regs)
```

The functions 'read-rn' and 'write-rn' are the two basic operations used to obtain and modify the register *rn* in the register file *rfile*. Operations on the register file are formalized in terms of these two functions. The functions 'get-nth' and 'put-nth' are the list operations to fetch and modify the nth element of a list.

3.2.3 The Program Counter

The program counter PC is simply represented as a nonnegative integer. As an invariant, the PC always points to the next memory location to be considered throughout the specification. Consequently, the PC will point to the next instruction after the execution of the current instruction.

3.2.4 The Condition Code Register

The condition code register CCR is also represented as a nonnegative integer. The first five bits of CCR designate the carry, the overflow, the zero, the negative, and the extend condition codes, respectively.

```
DEFINITION:
cvznx (c, v, z, n, x)
=  (fix-bit (c)
    +  ((2 * fix-bit (v))
        +  ((4 * fix-bit (z))
            +  ((8 * fix-bit (n)) + (16 * fix-bit (x))))))

DEFINITION:  set-cvznx (cvznx, ccr) = replace (5, cvznx, ccr)
```

The function 'cvznx' "collects" the five condition codes, and the function 'set-cvznx' updates the five condition codes in the condition code register. These two functions are used to update the condition codes in CCR.

3.2.5 The Memory

The memory is represented as a pair of binary trees. A binary representation for memory provides some efficiency for simulating MC68020 instructions. One of the binary trees is a formalization of memory protection – one may specify that any byte of memory is `'ram`, `'rom`, or `'unavailable`; the other binary tree holds the data, i.e., the actual bytes stored. As discussed elsewhere in this chapter, we use the notion of read-only memory to deal with the issue of cache consistency. We also believe that it is unrealistic to assert the correctness of machine code programs without carefully characterizing which parts of memory are read and written – few MC68020 chips are connected to a full 4 gigabytes of RAM. Memory protection issues are not specified in [41].

The following functions are the basic memory read/write functions. Operations on memory are defined in terms of these three functions. The functions 'pc-read-mem' and 'read-mem' return a bit vector formed by the k bytes from the memory starting at address pc or x, respectively. The function 'write-mem' stores the *value* in the k bytes of the memory starting at x.

```
DEFINITION:
pc-read-mem (pc, mem, k)
= if k ≃ 0 then 0
  else app (B,
            pc-byte-read (add (32, pc, k − 1), mem),
            pc-read-mem (pc, mem, k − 1)) endif

DEFINITION:
read-mem (x, mem, k)
= if k ≃ 0 then 0
  else app (B,
            byte-read (add (32, x, k − 1), mem),
            read-mem (x, mem, k − 1)) endif

DEFINITION:
write-mem (value, x, mem, k)
= if k ≃ 0 then mem
  else write-mem (tail (value, B),
                  x,
                  byte-write (value, add (32, x, k − 1), mem),
                  k − 1) endif
```

For memory protection, there are also three basic functions: 'pc-read-memp' specifies that a portion of the memory is read-only; 'read-memp' specifies that a portion of the memory is readable; 'write-memp' specifies that a portion of the memory is writable. We omit their definitions here.

3.3 Internal States and Effective Address Calculation

Many of the MC68020 instructions are too complicated to specify in a single step, especially when there is more than one effective address calculation. Therefore, we often use the following function to introduce internal states in their specifications.

```
DEFINITION:
mc-instate (oplen, ins, s)
= let s&addr be effec-addr (oplen, s_mode (ins), s_rn (ins), s)
  in
  if cadr (s&addr) = 'm
  then if read-memp (cddr (s&addr), mc-mem (s), op-sz (oplen))
       then s&addr
       else cons (halt (READ-SIGNAL, s), nil) endif
  else s&addr endif endlet
```

The function 'mc-instate' takes the operation size, the operation word of the current instruction, and the current machine state as arguments, and returns a pair consisting of an internal state after the source effective address calculation and the calculated effective address.

The function 'effec-addr' formalizes the effective address calculation. All the eighteen MC68020 addressing modes have been specified. An addressing mode can specify a constant that is the operand, a register that contains the operand, or a location in memory where the operand is stored. For the informal description and the formal definition, please refer to [41] and Appendix B.2.

3.4 The Specification of the SUB Instruction

Having addressed some important aspects of our MC68020 specification, we discuss in this section the formalization of the individual instructions. We use the SUB instruction as our example, which generally reflects our modeling approach to the other instructions.

The top-level loop of our specification is defined by a pair of functions, the *single-stepper* function 'stepi' and the *stepper* function 'stepn'.

```
DEFINITION:
stepn (s, n)
= if mc-haltp (s) ∨ (n ≃ 0) then s
  else stepn (stepi (s), n − 1) endif
```

```
DEFINITION:
stepi (s)
= if evenp (mc-pc (s))
  then if pc-word-readp (mc-pc (s), mc-mem (s))
       then execute-ins (current-ins (mc-pc (s), s),
                         update-pc (add (L, mc-pc (s), WSZ), s))
       else halt (PC-SIGNAL, s) endif
  else halt (PC-ODD-SIGNAL, s) endif
```

The stepper 'stepn' executes n instructions by calling the single stepper 'stepi'. But 'stepn' halts prematurely if the status field of s ceases to be `'running`.

'stepi' calls 'execute-ins' to compute the new machine state from the current state s by executing the current instruction if the program counter is aligned on a word boundary, as required by the MC68020, and points to read-only memory, as is checked by the function 'pc-word-readp'; otherwise, returns a machine state with the corresponding error message in the status field.

Roughly speaking, 'execute-ins' decodes the current instruction according to the opcode and jumps to the specification of the instruction identified. The first argument of 'execute-ins' should be the first word (operation word) of the current instruction, and the second argument should be an internal state with the program counter incremented by 2. The very top-level operation decoding is given by Table 3-14 in [41].

If the current instruction is '`subx <ea>,Dn`',[1] 'execute-ins' will call the following function 'sub-ins1' that specifies the resulting state of the execution of this SUB instruction.

```
DEFINITION:
sub-ins1 (oplen, ins, s)
= if sub-addr-modep1 (oplen, ins)
  then let s&addr be mc-instate (oplen, ins, s)
       in
       if mc-haltp (car (s&addr)) then car (s&addr)
       else d-mapping (oplen,
                       sub-effect (oplen,
                                   operand (oplen,
                                            cdr (s&addr),
                                            s),
                                   read-dn (oplen,
                                            d_rn (ins),
```

[1] This is only one of the two cases in the SUB instruction; please refer to [41] and [12] for more details.

```
                                        s)),
                        d_rn (ins),
                        car (s&addr)) endif endlet
    else halt (MODE-SIGNAL, s) endif
```

The function 'sub-ins1' first tests if the addressing mode of the current instruction is allowed by the MC68020. The addressing modes available to this instruction are specified by the following function.

```
DEFINITION:
sub-addr-modep1 (oplen, ins)
= (addr-modep (s_mode (ins), s_rn (ins))
   ∧ (¬ byte-an-direct-modep (oplen, s_mode (ins))))
```

which states that all the addressing modes are available to the SUB instruction, except that byte operation is not allowed in address register direct mode.

Next, an internal state is created using 'mc-instate', and the function 'd-mapping' takes the effects of the SUB instruction and the internal state to create the resulting state of the execution of this SUB instruction.

The effects of the SUB instruction are formalized by 'sub-effect' that returns a pair consisting of the result of the subtraction and the new condition codes.

```
DEFINITION:
sub-cvznx (oplen, sopd, dopd)
= cvznx (sub-c (oplen, sopd, dopd),
         sub-v (oplen, sopd, dopd),
         sub-z (oplen, sopd, dopd),
         sub-n (oplen, sopd, dopd),
         sub-c (oplen, sopd, dopd))

DEFINITION:
sub-effect (oplen, sopd, dopd)
= cons (sub (oplen, sopd, dopd), sub-cvznx (oplen, sopd, dopd))
```

The function 'cvznx' puts together the five new condition codes of the SUB instruction, which are formalized by the following four functions, paraphrasing the description given in Table 3-11 of the MC68020 manual [41]the function 'cvznx'. The X flag is the same as the C flag.

```
DEFINITION:
sub-c (n, sopd, dopd)
= let result be sub (n, sopd, dopd)
  in
  b-or (b-or (b-and (mbit (sopd, n), b-not (mbit (dopd, n))),
              b-and (mbit (result, n), b-not (mbit (dopd, n)))),
        b-and (mbit (sopd, n), mbit (result, n))) endlet
```

```
DEFINITION:
sub-v (n, sopd, dopd)
= let result be sub (n, sopd, dopd)
  in
  b-or (b-and (b-and (b-not (mbit (sopd, n)), mbit (dopd, n)),
               b-not (mbit (result, n))),
        b-and (b-and (mbit (sopd, n), b-not (mbit (dopd, n))),
               mbit (result, n))) endlet

DEFINITION:
sub-z (oplen, sopd, dopd)
= if sub (oplen, sopd, dopd) = 0 then B1
  else B0 endif

DEFINITION:
sub-n (oplen, sopd, dopd) = mbit (sub (oplen, sopd, dopd), oplen)
```

To paraphrase this, the carry bit is set to $(Sm \wedge \overline{Dm}) \vee (Rm \wedge \overline{Dm}) \vee (Sm \wedge Rm)$; the overflow bit is set to $(\overline{Sm} \wedge Dm \wedge \overline{Rm}) \vee (Sm \wedge \overline{Dm} \wedge Rm)$; the zero bit is set iff the subtraction is equal to 0; the negative bit is set to Rm, where Sm, Dm, and Rm denote the most significant bit of source, destination and result, respectively.

3.5 Discussion

We have described the MC68020 specification in the preceding few sections. We would like to conclude this chapter with some remarks about some of the interesting issues that have come up in the specification.

The needs of mathematical reasoning were the main concern during the development of the formal specification. The impact on program proving is nevertheless sometimes too subtle to realize at the stage of writing the specification. The specification has gone through several major and many minor changes as we understand more about mathematics at the machine-code level. For example, even though the functions 'pc-read-mem' and 'read-mem' are mathematically equivalent, the use of two different functions was motivated by program proving considerations. Technically speaking, different rewrite rules are set up for these two functions.

Natural number representation for bit vectors seems better for machine code program proving, whereas finite list representation seems better suited for hardware verification. In the context of system verification, this conflict can be reconciled by an equivalence proof for these two representations. In fact, we proved their equivalence in Nqthm when we switched to a natural number representation.

The MC68020 has an on-chip instruction cache, but a write operation does not invalidate or modify the corresponding entry in the instruction cache. Rather than formalizing the details of the MC68020 cache (which has changed from MC680x0 processor to processor), we have adopted, for the time being, the strategy of *requiring* that instruction fetches be from *read-only* parts of the memory, and therefore, if the instruction cache is entirely valid at the beginning of the execution, it will remain valid all throughout the execution.

Some MC68020 instructions are sensitive to internal evaluation order. For instance, the MOVE instruction has two effective address calculations. Because of the side effect of effective address calculation, it is necessary to know which address is calculated first. This information is not specified in the Motorola literature, but by speaking with Motorola engineer Jim Eifert in April 1990, we learned that it is an internal Motorola policy that the source effective address is always calculated first.

Ideally, we would specify the condition codes in a way most natural to the 'user.' But in order to assure full compliance with the MC68020 specification [41], we have followed the syntactical definition described in Table 3-11 of [41]. For instance, the definition of 'sub-c' is perhaps not the way the programmer views the carry bit of a SUB (subtraction) instruction! One of the problems we have to deal with in the verification phase is to eliminate these 'semantic gaps.' This problem has been addressed in the lemma library.

The MC68020 provides a very rich set of addressing modes. The definition of effective address calculation is rather complicated and required great care to formalize completely and in a form amenable to formal reasoning.

In addition to using the Nqthm prover to prove general theorems about the correctness of MC68020 programs under the semantics provided by 'stepn', as we discuss in subsequent chapters, it is noteworthy that it is actually possible for us, within Nqthm, to *run* 'stepn' on concrete data. That is, Nqthm together with 'stepn' provides a simulator for the MC68020, albeit one that requires approximately 1,000,000 Sun-3 (MC68020) instructions to simulate a single MC68020 instruction. We mention this simulation possibility only to emphasize the important point: our 'semantics' for the MC68020 is an *operational* semantics in the strictest sense of the word. There are several advantages to having such an operational characterization of the semantics of our computational model:

- It is possible to "test" the specification's correctness by executing it on specific

data and comparing the result with the behavior of an actual MC68020.[2] Ken Albin at Computational Logic has been working on a testing suite for the MC68020 specification.

- By giving the MC68020 semantics entirely with definitions instead of with an *ad hoc* collection of axioms, we are guaranteed that the specification is consistent, relative to the consistency of elementary number theory.
- The executability of the formal model provides a fast means of symbolic manipulation in some cases during program proving.

[2]While testing does not find all bugs, it does find some, and that helps us gain confidence in our formal model!

Chapter 4

The Mechanization of Machine Code Reasoning

Specifying a computing device in a formal logic allows us to study its behavior mathematically. This is our main motivation for specifying the MC68020 microprocessor in the Nqthm logic. Starting in this chapter, we investigate the problem of mechanically verifying, using Nqthm, MC68020 machine code programs based on our MC68020 formal model.

The development of lemmas is a key to success in any use of an interactive theorem proving system, certainly of Nqthm. Lemmas are saved as derived inference rules that affect the future behavior of the system. The quality of the lemmas often determines the success of the entire proof effort. This chapter describes our effort of developing a lemma library that mechanizes a basic mathematical theory of machine code reasoning. Combining the MC68020 formal model and this lemma library, we have built, on top of Nqthm, a powerful proof system that has been used effectively to verify many MC68020 machine code programs mechanically.

We have invested more time developing our lemma database than on any other aspect of this multi-year project. First, mechanizing a theory is not a trivial step, practically. The lemmas need to be formulated to integrate nicely into the Nqthm proving engine so that the prover can find them at the "right time" and apply them automatically. Many proofs require the application of so many lemmas that a more manual proof-checking approach, in which each application of each lemma is suggested by the user, seems, practically speaking, out of the question. Second, we insist that all the lemmas be mechanically proved by Nqthm before being admitted into the system. Allowing the users of theorem provers to assert without proof the lemmas they think correct seems a pretty sure way to render their systems inconsistent. Finally, the management of the lemma library becomes very complicated and time consuming when the library is getting rather large. Interference among lemmas makes it extremely hard to predict the behavior of the system for any changes to the lemma library.

Our approach to developing the lemma library can be roughly viewed as 'bottom-up.' We carefully study each of the concepts involved, in the hope of proving a set of lemmas that fully characterizes these concepts. Our presentation of the library in this chapter certainly carries the same flavor. We will carefully address some of the important issues we have dealt with in the development of the library. Appendix B.3 contains the Nqthm script of the entire lemma library.

4.1 Integer Arithmetic

Integer arithmetic has been the basic theory of our program proving work. In our work, we have at least one more reason to develop a powerful sublibrary for integer arithmetic: all the bit vector operations are formalized with nonnegative integer arithmetic; hence theorems about bit vectors are merely theorems about nonnegative integers. Most of the lemmas in this sublibrary are concerned with these basic arithmetic functions: $x + y$, $x - y$, $x * y$, x **mod** y, $x \div y$, $\exp(x, y)$, $\log(b, x)$, and $x < y$. During the development, we have greatly benefited from an integer library [31] developed at Computational Logic, Inc.

The lemmas in this sublibrary are simply a collection of basic facts in elementary number theory, which are particularly useful in program proving at the machine-code level. Most of the lemmas have quite intuitive meanings. We will not elaborate on this sublibrary, except showing below two simple lemmas as examples.

THEOREM: quotient-times-cancel
$((x * y) \div (x * z))$
$=$ **if** $x \simeq 0$ **then** 0
else $y \div z$ **endif**

THEOREM: remainder-plus-remainder1
$((x + (y \textbf{ mod } z)) \textbf{ mod } z) = ((x + y) \textbf{ mod } z)$

We here offer no exposition as the lemmas have spoken for themselves. The rest of the library relies heavily on this integer sublibrary.

4.2 Bit Vector Arithmetic

Since we model the MC68020 at the machine-code level, it is inevitable that we study mathematical properties of bit vector operations. The purpose here is

to establish a set of proof rules to support bit vector arithmetic reasoning at a relatively high level of abstraction. Reducing bit vector reasoning to integer arithmetic reasoning all the time is practically intractable.

All the bit vector operations described in the preceding chapter have been addressed in our lemma library to some extent. We have no interest in mechanizing the mathematical reasoning of modulo arithmetic in general, which seems quite challenging to us. The class of bit vector lemmas we have proved is largely based on our needs. Furthermore, we did not expect much bit vector reasoning in the program proving phase, which turned out to be the case in verifying machine code programs.

Just to exhibit the lemmas of this nature, the following two simple lemmas are taken from the library.

THEOREM: add-associativity
add (n, add (n, x, y), z) = add (n, x, add (n, y, z))

THEOREM: bitn-tail
bitn (tail (x, i), j) = bitn (x, $i + j$)

Intuitively, 'add-associativity' establishes the associativity of bit vector addition, and 'bitn-tail' proves that the jth bit of '(tail x i)' is the $(i + j)$th bit of x. Note that 'add-associativity' is simply an immediate consequence of the lemma 'remainder-plus-remainder1' mentioned in the preceding section.

It is worth noting that we have included in our lemma library a few useful meta lemmas about bit vector arithmetic. For example, the following lemma cancels the like addends on two sides of an equality.

THEOREM: correctness-of-cancel-equal-add
eval$ (**t**, x, a) = eval$ (**t**, cancel-equal-add (x), a)

4.3 Interpretations of Bit Vector Operations

At the machine-code level, mathematical functions are modeled by bit vector operations. It is therefore necessary to establish the correspondence between the "real" mathematical functions and their bit vector "implementations". This important issue has been addressed by interpretation lemmas in our lemma library. Basically, there are two kinds of lemmas we have considered: unsigned and signed integer interpretations. We have proved the interpretation lemmas for the basic unsigned and signed integer operations supported by the MC68020 instruction set.

In this section, we intend to explain the basic ideas using the two interpretation lemmas for addition.

First, let us introduce the basic conversion functions about the few basic data types we are considering.

```
DEFINITION:  nat-to-uint (x) = fix (x)

DEFINITION:  uint-to-nat (x) = fix (x)

DEFINITION:
nat-to-int (x, n)
=  if x < exp (2, n − 1) then fix (x)
   else − (exp (2, n) − x) endif

DEFINITION:
int-to-nat (x, size)
=  if negativep (x) then exp (2, size) − negative-guts (x)
   else fix (x) endif
```

The conversion between bit vectors and unsigned integers is given by the functions 'nat-to-uint' and 'uint-to-nat'; the conversion between bit vectors and signed integers is given by the functions 'nat-to-int' and 'int-to-nat'.

Now, let us consider the interpretations for the bit vector operation 'add' whose definition was given in the previous chapter. As we know from two's complement addition, the function 'add' can be viewed as either unsigned or signed integer addition, depending on how we interpret the two bit vector inputs. Intuitively speaking, the lemma 'add-uint' establishes the relation between 'add' and 'plus', if the unsigned integer interpretation is taken; the lemma 'add-int' establishes the relation between 'add' and 'iplus', if the signed integer interpretation is taken.

```
THEOREM: add-uint
(nat-rangep (x, n) ∧ nat-rangep (y, n))
⇒  (nat-to-uint (add (n, x, y))
     =  if (nat-to-uint (x) + nat-to-uint (y)) < exp (2, n)
        then nat-to-uint (x) + nat-to-uint (y)
        else (nat-to-uint (x) + nat-to-uint (y))
             −  exp (2, n) endif)

THEOREM: add-int
(nat-rangep (x, n) ∧ nat-rangep (y, n))
⇒  (nat-to-int (add (n, x, y), n)
     =  if int-rangep (iplus (nat-to-int (x, n), nat-to-int (y, n)), n)
        then iplus (nat-to-int (x, n), nat-to-int (y, n))
```

```
elseif negativep(nat-to-int(x, n))
then iplus(nat-to-int(x, n),
           iplus(nat-to-int(y, n), exp(2, n)))
else iplus(nat-to-int(x, n),
           iplus(nat-to-int(y, n), − exp(2, n))) endif)
```

Roughly speaking, the lemma 'add-uint' proves the equivalence of add(n, x, y) and $x + y$, if there is no carry; the lemma 'add-uint' proves the equivalence of add(n, x, y) and iplus(x, y), if there is no overflow. The interpretation lemmas of the other bit vector operations are formulated in the same way.

The importance of these interpretation lemmas is two-fold. From the point of view of semantics, a higher level of abstraction is achieved through these interpretation lemmas. From the point of view of theorem proving, these interpretation lemmas get us into the familiar mathematical domains for which theorem provers are built.

4.4 Machine State Management

Machine state management is probably the most difficult part of the library to construct. It mainly concerns general theorems about the machine state and its components. In proofs of programs, machine states are the objects the theorem prover has to reason about and the user has to inspect when the proof fails. The machine state is often very complex and difficult to manage. By developing carefully a set of lemmas for each of the components of the machine state, we are able to gain some level of abstraction that helps the theorem prover focus on the relevant part of the proof and helps the user understand the proof script, in particular, when the proof attempt fails.[1] In a word, we want to have both automation and user control of the proofs. We think we have achieved this goal.

Intuitively, one might think of these lemmas as some kind of Hoare rules [23] for machine code program proving. But these lemmas are rather complicated and delicate because of the complexity of the MC68020 architecture. In this section, we briefly discuss the lemmas for the register file and the memory. The lemmas for the other state components are quite straightforward.

[1] Given such a large and complex model, we would regard it as a big win if the theorem prover responds to the proofs and prints out *readable* proof scripts.

4.4.1 The Register File

As described in Chapter 3, the functions 'read-rn' and 'write-rn' are the two main operations used to read and modify some register in the register file. We have proved a set of lemmas that captures the useful properties of these two functions, whose definitions were subsequently disabled.[2] The following theorem shows one of the key lemmas.

```
THEOREM: read-write-rn
 read-rn (n2, rn, write-rn (n1, value, rm, rfile))
 =  if fix (rm) = fix (rn)
      then if n2 ≤ n1  then head (value, n2)
                       else replace (n1, value, read-rn (n2, rn, rfile)) endif
      else read-rn (n2, rn, rfile) endif
```

Roughly, this lemma says that the result of reading the content of register rn after writing $value$ to register rm equals:

- the result of reading the previous content of register rn, if $rn \neq rm$.
- the first $n2$ bits of $value$, if $(rn = rm) \wedge (n2 \leq n1)$.
- the result of concatenating $value$ and the $n1$ to $n2$ bits of the previous content of register rn, if $(rn = rm) \wedge (n2 > n1)$.

As evidenced in this lemma, the main difficulty here is to deal with the various types of data in the registers.

4.4.2 The Memory

As described in Chapter 3, the functions 'read-mem' and 'write-mem' are the two main operations used to read from and write to the memory. A set of lemmas was proved to capture some useful properties of these two functions, whose definitions were subsequently disabled. We present here two key lemmas of this type whose functions are similar to the lemma 'read-write-rn' above.

```
DEFINITION:
read-write-mem-end (x, value, y, mem, m, n)
=  read-mem (x, write-mem (value, y, mem, m), n)
```

[2] By disabling an event, we prohibit the Nqthm prover from using the event in the subsequent proofs. See [9]

```
THEOREM: read-write-mem1
 read-mem (x, write-mem (value, y, mem, k), n)
 =  if disjoint (x, n, y, k) then read-mem (x, mem, n)
    else read-write-mem-end (x, value, y, mem, k, n) endif

THEOREM: read-write-mem2
 uint-rangep (n, 32)
 ⇒  (read-mem (x, write-mem (value, x, mem, n), n) = head (value, 8 * n))
```

Very roughly, this says that the result of reading at location x after writing $value$ at location y is either $value$, by the lemma 'read-write-mem2', or the previous contents of x, by the lemma 'read-write-mem1', according to whether x is equal to y or not. Mathematically, the function disjoint (x, m, y, n) is true iff $\{x, x+1, \ldots, x+(m-1)\} \cap \{y, y+1, \ldots, y+(n-1)\} = \phi$. 'disjoint' is used to specify that there is no overlap of two memory portions. The function 'read-write-mem-end' is used as a trick to truncate some portion of the proof space that is believed to be useless. Functions of this type are always disabled globally.

There are a large number of lemmas about 'disjoint' in the library which are primarily used to establish the disjointness of two memory segments in proofs. This class of lemmas was extremely difficult to formulate and manage efficiently in Nqthm. This perhaps is the price of our use of a single memory addressing space for the MC68020 model. Up to now, we do not think we have managed to produce a satisfactory proof automation of the seemingly very simple mathematics about disjointness in Nqthm. It seems to be a place in the lemma library that may need some more careful thinking and reimplementation if we have a chance to do it again.

4.5 Interpretations of Condition Codes

Another important class of lemmas that has its use in the branching instructions is the interpretation of the condition codes of various instructions. Again, we use the SUB instruction in our discussion.

In the preceding chapter, we have given the definition of the condition codes of the SUB instruction. But it is often not the most useful mathematical characterizations of the condition codes to use when it comes to program proving. We therefore need to prove a set of lemmas that provides the mathematical meaning of condition codes used in program proving. Table 4.1 shows all the condition codes that can be specified in the Bcc instruction. The following lemmas characterize the most useful semantics of these condition codes for program proving.

CC	carry clear	$\overline{C}$	LS	low or same	$C + Z$
CS	carry set	C	LT	less than	$N * \overline{V} + \overline{N} * V$
EQ	equal	Z	MI	minus	N
GE	greater or equal	$N * V + \overline{N} * \overline{V}$	NE	not equal	$\overline{Z}$
GT	greater than	$N * V * \overline{Z} + \overline{N} * \overline{V} * \overline{Z}$	PL	plus	$\overline{N}$
HI	high	$\overline{C} * \overline{Z}$	VC	overflow clear	$\overline{V}$
LE	less or equal	$Z + N * \overline{V} + \overline{N} * V$	VS	overflow set	V

Table 4.1: The Bcc Condition Codes

THEOREM: sub-bls
(nat-rangep (x, n) $\wedge$ nat-rangep (y, n) $\wedge$ ($n \not\simeq 0$))
$\Rightarrow$ (bls (sub-c (n, x, y), sub-z (n, x, y))
= **if** nat-to-uint (x) < nat-to-uint (y) **then** 0
else 1 **endif**)

THEOREM: sub-beq-uint
(nat-rangep (x, n) $\wedge$ nat-rangep (y, n))
$\Rightarrow$ (beq (sub-z (n, x, y))
= **if** nat-to-uint (x) = nat-to-uint (y) **then** 1
else 0 **endif**)

THEOREM: sub-bcs&cc
(nat-rangep (x, n) $\wedge$ nat-rangep (y, n) $\wedge$ ($n \not\simeq 0$))
$\Rightarrow$ (bcs (sub-c (n, x, y))
= **if** nat-to-uint (y) < nat-to-uint (x) **then** 1
else 0 **endif**)

THEOREM: sub-bvs&vc
(nat-rangep (x, n) $\wedge$ nat-rangep (y, n) $\wedge$ ($n \not\simeq 0$))
$\Rightarrow$ (bvs (sub-v (n, x, y))
= **if** int-rangep (idifference (nat-to-int (y, n), nat-to-int (x, n)),
n) **then** 0
else 1 **endif**)

THEOREM: sub-bmi
(nat-rangep (x, n) $\wedge$ nat-rangep (y, n) $\wedge$ ($n \not\simeq 0$))
$\Rightarrow$ (bmi (sub-n (n, x, y))
= **if** int-rangep (idifference (nat-to-int (y, n), nat-to-int (x, n)),
n)
then if ilessp (nat-to-int (y, n), nat-to-int (x, n)) **then** 1
else 0 **endif**
elseif ilessp (nat-to-int (y, n), nat-to-int (x, n)) **then** 0
else 1 **endif**)

THEOREM: sub-bge
(nat-rangep (x, n) $\wedge$ nat-rangep (y, n) $\wedge$ ($n \not\simeq 0$))
$\Rightarrow$ (bge (sub-v (n, x, y), sub-n (n, x, y))
= **if** ilessp (nat-to-int (y, n), nat-to-int (x, n)) **then** 0
else 1 **endif**)

THEOREM: sub-bgt
(nat-rangep (x, n) $\wedge$ nat-rangep (y, n) $\wedge$ ($n \not\simeq 0$))
$\Rightarrow$ (bgt (sub-v (n, x, y), sub-z (n, x, y), sub-n (n, x, y))
= **if** ilessp (nat-to-int (x, n), nat-to-int (y, n)) **then** 1
else 0 **endif**)

Roughly speaking, the lemma 'sub-bls' states that the condition LS is true iff $x \geq y$; the lemma 'sub-beq-uint' states that the condition EQ is true iff $x = y$; the lemma 'sub-bcs&cc' states that the condition CS is true iff $y < x$; the lemma 'sub-bvs&vc' states that the condition VS is true iff $-2^{(n-1)} \leq (y - x) < 2^{(n-1)}$; the lemma 'sub-bmi' states that the condition MI is true iff $x < y$ for no overflow case or $y < x$ for overflow case; the lemma 'sub-bge' states that the condition GE is true iff $y \geq x$; the lemma 'sub-bgt' states that the condition GT is true iff $y < x$.

After we proved these seven lemmas, the definitions of 'sub-c', 'sub-v', 'sub-z', and 'sub-n' are no longer useful, and are therefore disabled.

4.6 The Interpreter Lemmas

The last class of lemmas we want to explain in this chapter concerns the general (program independent) properties of the interpreter. The lemmas of this type basically take the form: $p(s) \Rightarrow p(\text{stepn}(s))$. This class of lemmas is not only useful in program proving, but also useful in sharpening our understanding about the MC68020 model.

Most of the lemmas in this class are quite intuitive. Again, we give a couple of simple examples to make our discussion concrete.

THEOREM: stepn-rom-addrp
rom-addrp (x, mc-mem (stepn (s, n)), k) = rom-addrp (x, mc-mem (s), k)

THEOREM: stepn-read-mem
rom-addrp (x, mc-mem (s), k)
$\Rightarrow$ (read-mem (x, mc-mem (stepn (s, n)), k) = read-mem (x, mc-mem (s), k))

The lemma 'stepn-rom-addrp' proves that the readability of any portion of the memory is not changed after the execution of any number of any instructions; the lemma 'stepn-read-mem' proves that the content of the read-only memory is not changed after the execution of any number of any instructions.

Typically, the proof of the lemmas of this class is very shallow mathematically, but tedious and painful practically. Because of the complexity of the MC68020

model, this kind of proof often ends up splitting into a huge number of cases, and some lemmas have to be provided to control the case analysis. We believe this problem is intrinsic: any theorem prover has to visit every corner of the interpreter in order to prove a single fact about the interpreter.

Chapter 5

Machine Code Program Proving

Among the possible applications of our MC68020 formal specification, we are currently primarily concerned with studying the verification of specific object code programs. This chapter, together with the next two chapters, addresses the problem of program proving at the machine-code level; this is the central theme of this dissertation. We illustrate our verification approach with some examples we have studied. We hope that these examples may provide concrete evidence that this work can be applied to some moderate sized *real* applications. It is also our intention, by presenting these examples, to provide the reader a fair account of the difficulty of formalizing and proving machine code programs.

In this chapter, we investigate the formal correctness proofs of the object code of five small programs written in high level programming languages. The first one is the C function `gcd` already given in Chapter 2. The second is an ADA program `isqrt` that computes the integer square root using Newton's method. The third and the fourth are slightly modified versions of binary search and quick sort taken from *The C Programming Language* [32]. The last one is a C program that implements the Boyer-Moore Majority Voting algorithm. The object code of these programs is generated by Gnu C or Verdix Ada compilers, as explained in Chapter 2.

There perhaps is another reason that we wanted to discuss these five examples in length. Proving programs has sharpened our understanding of the MC68020 model and the mathematics for machine code reasoning. These five examples have been particularly beneficial to us. We feel that a detailed discussion of them would be equally beneficial to those verificationists who happen to attempt these examples on their verification system.

This chapter contains six sections. The first details our approach to machine code program proving. The rest are devoted to the five examples: one for each section. For each example, we will discuss the formalization, the proof, and some other important issues such as the time analysis and memory space bounds of

the program. We advise the reader patiently to go through the GCD section first since many concepts are introduced there and not repeated in the other sections. Appendix C contains the complete proof script for the five examples described in this chapter.

5.1 The Approach

We have briefly described in Chapter 2 our approach to machine code program proving. This section provides a more rigorous mathematical treatment of our program proving methodology.

5.1.1 The Formulation

Given a machine code program p, what we need to formalize in the Nqthm logic are the following functions.

- a predicate p-statep (s) that characterizes the preconditions on the initial state s where the program starts.
- a time function p-t (s) that defines the number of instructions needed to complete the computation.
- a set of mathematical functions p-f1 (s), p-f2 (s), ..., p-fn (s) that specifies the intended functional behavior of the program.

The correctness of the given program is then formalized with the following eight theorems to be proved.

P-1. The resulting machine state is 'normal', i.e., the processor status word is equal to `'running`.

$$\text{p-statep}\,(s) \Rightarrow (\text{mc-status}\,(\text{stepn}\,(s,\ \text{p-t}\,(s))) = \texttt{'running})$$

P-2. The new program counter is set to the right location specified by rts-addr (s).

$$\text{p-statep}\,(s) \Rightarrow (\text{mc-pc}\,(\text{stepn}\,(s,\ \text{p-t}\,(s))) = \text{rts-addr}\,(s))$$

P-3. The value of the address register A6 in the resulting state is equal to the value of A6 in the initial state s.

$$\text{p-statep}\,(s) \Rightarrow (\text{read-an}\,(32,\ 6,\ \text{stepn}\,(s,\ \text{p-t}\,(s))) = \text{read-an}\,(32,\ 6,\ s))$$

The register A6 is conventionally used as the frame pointer by many compilers.

P-4. The value of the stack pointer A7 in the resulting state is incremented by 4. The return address is popped off the stack when control returns from a subprogram to the caller.

$$\text{p-statep}\,(s) \Rightarrow (\text{read-an}\,(32,\ 7,\ \text{stepn}\,(s,\ \text{p-t}\,(s))) = \text{add}\,(32,\ \text{read-an}\,(32,\ 7,\ s),\ 4))$$

P-5. The values of the data registers D2 - D7 and the address registers A2 - A5 are equal to their value in the initial state s.

$$\begin{array}{l}(\text{p-statep}\,(s) \wedge \text{d2-7a2-5p}\,(rn) \wedge (oplen \leq 32)) \\ \quad \Rightarrow \\ (\text{read-rn}\,(oplen,\ rn,\ \text{mc-rfile}\,(\text{stepn}\,(s,\ \text{p-t}\,(s)))) = \text{read-rn}\,(oplen,\ rn,\ \text{mc-rfile}\,(s)))\end{array}$$

Most of the compilers allow subprograms to use registers D0, D1, A0, and A1 without any conditions. Therefore, we do not have any obligations to assert anything about these registers.

P-6. The program only changes the intended portions of memory. For any x and k, if the memory segment $[x, x+1, \ldots, x+(k-1)]$ is disjoint from the portions of the memory the program intends to change, then its content is not modified by the program.

$$\begin{array}{l}(\text{p-statep}\,(s) \wedge \text{p-disjointness}\,(x,\ k,\ s)) \\ \quad \Rightarrow \\ (\text{read-mem}\,(x,\ \text{mc-mem}\,(\text{stepn}\,(s,\ \text{p-t}\,(s))),\ k) = \text{read-mem}\,(x,\ \text{mc-mem}\,(s),\ k))\end{array}$$

The disjoint predicate 'p-disjointness' normally takes the form of a disjunction of 'disjoint's.

P-7. The functional behavior of the given program p is equivalent to the mathematical functions p-f1 (s), p-f2 (s), ..., p-fn (s).

$$\text{p-statep}\,(s) \Rightarrow \text{p-sem-eq}\,(\text{stepn}\,(s,\ \text{p-t}\,(s)),\ \text{f1}\,(s),\ \text{f2}\,(s),\ \ldots\ \text{fn}\,(s))$$

The equivalence relation 'p-sem-eq' normally takes the form of a conjunct of 'equal's.

P-8. The functions p-f1 (s), p-f2 (s), ..., p-fn (s) meet their requirement specifications, which varies from program to program.

All the machine code programs presented in this dissertation are mechanically verified using the above formulation.

5.1.2 The Proof

Our formulation of the correctness theorem clearly divides the proof into two independent phases: the theorem **P-8** deals with the correctness of the underlying algorithm and the others deal with the correctness of its implementation. Success in separating the two issues in the formulation and tackling each of them in isolation makes the whole proof effort easier. To be more concrete, the correctness proof for a given machine code program is divided into the following two steps.

1. We attempt to prove the theorems **P-1** through **P-7**. In particular, **P-7** establishes the equivalence of the algorithm, formalized in the Nqthm logic as those machine independent functions 'p-f1', 'p-f2', ..., 'p-fn', with the result of running the MC68020 specification on the given machine code program. What we prove in this step is that the given machine code program does implement the algorithm, which, however, says nothing about the correctness of the algorithm.

2. We attempt to prove the theorem **P-8**, which establishes the correctness of the algorithm according to some specification. Note here we do not need to deal with any MC68020 related specifics in this step. So, we can focus completely on the mathematics of the algorithm, and fully enjoy many of the mathematical laws that are not available at the processor level.

To separate the two steps successfully, the formalization of the algorithm, the functions 'p-f1', 'p-f2', ..., 'p-fn', has to be machine independent. This can be done in a quite natural way – a straightforward paraphrase in the Nqthm logic of the given C/Ada/LISP program. We believe this poses no problem to us at all.

Step 2 is completely unrelated to MC68020 machine code programs, and is the kind of proof Nqthm users often do (and enjoy). Our main focus has been on step 1 in this work. The lemma library described in Chapter 4 is just a set of derived inference rules devoted to the proofs in step 1.

5.2 Greatest Common Divisor

The first example is the continuation of the GCD story started in Section 2.2.1 of Chapter 2, where we only explained how we generate the machine code to be verified. Here, we will show the correctness proof of that machine code.

5.2.1 The Formalization

According to our formulation, we need to formalize in the Nqthm logic the preconditions, the time function, and the functional behavior of this program GCD.

The function GCD-CODE formalizes the machine code of `gcd` as a list of 60 unsigned integers which have been obtained through GDB as described in Chapter 2. The function gcd-statep $(s,\ a,\ b)$ characterizes the preconditions on the initial state s.

```
DEFINITION:
GCD-CODE
=  '(78 86 0 0 72 231 48 0 36 46 0 8 38 46 0 12 74 130
     103 28 74 131 102 4 32 2 96 22 182 130 108 8 76
     67 40 0 36 0 96 232 76 66 56 0 38 0 96 224 32 3
     76 238 0 12 255 248 78 94 78 117)
```

```
DEFINITION:
gcd-statep (s, a, b)
=  ((mc-status (s) = 'running)
    ∧  evenp (mc-pc (s))
    ∧  rom-addrp (mc-pc (s), mc-mem (s), 60)
    ∧  mcode-addrp (mc-pc (s), mc-mem (s), GCD-CODE)
    ∧  ram-addrp (sub (32, 12, read-sp (s)), mc-mem (s), 24)
    ∧  (a = iread-mem (add (32, read-sp (s), 4), mc-mem (s), 4))
    ∧  (b = iread-mem (add (32, read-sp (s), 8), mc-mem (s), 4))
    ∧  (a ∈ N)
    ∧  (b ∈ N))
```

The function gcd-statep $(s,\ a,\ b)$ imposes the following conditions on the intitial state s.

- s is in the user mode, i.e., the processor status word is equal to `'running`.
- The program counter of s is even. The function evenp (x) asserts that x is even.
- The program GCD-CODE is stored in the 60 consecutive bytes in the memory, starting from the address pointed to by the program counter. The function mcode-addrp $(x,\ mem,\ code)$ asserts that *code* is stored in the memory *mem* starting from the address x. The function rom-addrp $(x,\ mem,\ n)$ asserts that the memory segment $[x, x+1, \ldots, (x+(n-1))]$ is ROM.
- The 24 bytes from $sp-12$ to $sp+12$ are RAM. The function ram-addrp $(x,$ *mem*, $n)$ asserts that the memory segment $[x, x+1, \ldots, (x+(n-1))]$ is RAM.

- The integers a and b are on the user stack, and both are nonnegative. The function iread-mem $(x,\ mem,\ n)$ returns the integer formed by the n bytes in the memory mem at location x.

The function gcd-t $(a,\ b)$ defines the number of instructions needed to complete the GCD program. Note that gcd-t $(a,\ b)$ can be viewed as just counting instructions needed in the execution of GCD.

```
DEFINITION:
gcd-t1 (a, b)
= if a ≃ 0 then 6
  elseif b ≃ 0 then 9
  elseif b < a then 9 + gcd-t1 (a mod b, b)
  else 9 + gcd-t1 (a, b mod a) endif

DEFINITION: gcd-t (a, b) = (4 + gcd-t1 (a, b))
```

The functional behavior of the program is specified by the following function gcd $(a,\ b)$, which is just a formalization in the Nqthm logic of the algorithm employed.

```
DEFINITION:
gcd (a, b)
= if a ≃ 0 then fix (b)
  elseif b ≃ 0 then a
  elseif b < a then gcd (a mod b, b)
  else gcd (a, b mod a) endif
```

5.2.2 The Proof

We follow strictly the two step proof outlined in Section 1. In the first step, we prove the following theorem that is a conjunct of seven formulas corresponding exactly to the theorems **P-1** to **P-7**.

```
THEOREM: gcd-correctness
 let sn be stepn (s, gcd-t (a, b))
 in
 gcd-statep (s, a, b)
 ⇒ ((mc-status (sn) = 'running)
     ∧ (mc-pc (sn) = rts-addr (s))
     ∧ (read-rn (32, 14, mc-rfile (sn))
          = read-rn (32, 14, mc-rfile (s)))
     ∧ (read-rn (32, 15, mc-rfile (sn))
          = add (32, read-an (32, 7, s), 4))
     ∧ ((d2-7a2-5p (rn) ∧ (oplen ≤ 32))
```

$\Rightarrow$ (read-rn ($oplen$, rn, mc-rfile (sn))
= read-rn ($oplen$, rn, mc-rfile (s))))
$\wedge$ (disjoint (x, k, sub (32, 12, read-sp (s)), 24)
$\Rightarrow$ (read-mem (x, mc-mem (sn), k)
= read-mem (x, mc-mem (s), k)))
$\wedge$ (iread-dn (32, 0, sn) = gcd (a, b))) **endlet**

In particular, the last formula in this theorem establishes that the content of data register D0 is equal to gcd (a, b) after executing gcd-t (a, b) instructions from an initial state s that satisfies the precondition gcd-statep (s, a, b). This equivalence allows us to study the Nqthm function gcd (a, b) instead of the machine code program.

The second step is therefore to prove that gcd (a, b) does compute the greatest common divisor of the two nonnegative integers a and b, which is asserted by the following two theorems:

THEOREM: gcd-is-cd
((a **mod** gcd (a, b)) = 0) $\wedge$ ((b **mod** gcd (a, b)) = 0)

THEOREM: gcd-the-greatest
($a \not\simeq 0$) $\wedge$ ($b \not\simeq 0$) $\wedge$ ((a **mod** x) = 0) $\wedge$ ((b **mod** x) = 0))
$\Rightarrow$ (gcd (a, b) $\not< x$)

The theorem 'gcd-is-cd' proves that gcd (a, b) is a common divisor of a and b, and the theorem 'gcd-the-greatest' proves that any common divisor of a and b is not greater than gcd (a, b).

5.2.3 A Simple Timing Analysis

The fact that the function gcd-t (a, b) returns the exact number of MC68020 instructions executed by the GCD program allows us to obtain the timing constraints of the GCD program by studying the mathematical properties of this 'gcd-t' function. This is how we analyze the real-time bounds of machine code programs.

In this GCD example, we have mechanically proved that gcd-t (a, b) is no more than 580, provided both a and b are less than 2^{31}.

THEOREM: gcd-t-ubound
(($a <$ exp (2, 31)) $\wedge$ ($b <$ exp (2, 31))) $\Rightarrow$ (gcd-t (a, b) $\leq$ 580)

This theorem tells us that the GCD program terminates within 580 instructions. Thus we can easily obtain a crude upper bound on the real-time execution of the GCD program, given a worst-case single instruction execution figure. For a less crude real-time bounds analysis, we would need to incorporate time information for each individual instruction, something that seems to us a quite natural and an easy extension to our specification.

The theorem 'gcd-t-ubound' is just an immediate consequence of 'gcd-t-ub'.

THEOREM: gcd-t-ub
$\text{gcd-t}(a, b) \leq (22 + (9 * (\log(2, a) + \log(2, b))))$

5.3 Integer Square Root

In this section, we study the correctness of the object code of the following Ada program `isqrt` that computes the integer square root of a given nonnegative integer using the Newton's method. The binary was provided by Dr. Steve Zeigler of Verdix, and was generated by the Verdix Ada compiler.

```
function isqrt (i:integer) return integer is
  j : integer := (i / 2);
 begin
  while ((i / j) < j) loop
    j := (j + (i / j)) / 2;
  end loop;
  return j;
 end isqrt;
```

```
The MC68020 assembly code generated by Verdix Ada Compiler.

   1  function isqrt (i:integer) return integer is
      00000: link.w      a6, #-04
   2     j : integer := (i / 2);
      00004: move.l      d2, d1
      00006: bge.b       06    -> 0e
      00008: addi.l      #01, d1
      0000e: asr.l       #01, d1
   3  begin
   4     while not ((i / j) >= j) loop
      00010: move.l      d2, d0
      00012: divsl.l     d1, d0:d0
      00016: trapv
      00018: cmp.l       d0, d1
      0001a: ble.b       01c    -> 038
   5        j := (j + (i / j)) / 2;
      0001c: add.l       d1, d0
```

```
   0001e: trapv
   00020: move.l      d0, d1
   00022: bge.b       06    -> 02a
   00024: addi.l      #01, d1
   0002a: asr.l       #01, d1
4     while not ((i / j) >= j) loop
   0002c: move.l      d2, d0
   0002e: divsl.l     d1, d0:d0
   00032: trapv
6  end loop;
   00034: cmp.l       d0, d1
   00036: bgt.b       -01c    -> 01c
7  return j;
   00038: move.l      d1, d0
   0003a: unlk        a6
   0003c: rts
8 end isqrt;
```

5.3.1 The Formalization

According to our formulation, we need to formalize in the Nqthm logic the preconditions, the time function, and the functional behavior of this ISQRT program.

The function ISQRT-CODE defines the machine code of `isqrt` as a list of unsigned integers. The function isqrt-statep (s, i) characterizes the preconditions of the initial state s.

```
DEFINITION:
ISQRT-CODE
=  '(78 86 255 252 34 2 108 6 6 129 0 0 0 1 226 129 32
     2 76 65 8 0 78 118 178 128 111 28 208 129 78 118
     34 0 108 6 6 129 0 0 0 1 226 129 32 2 76 65 8 0
     78 118 178 128 110 228 32 1 78 94 78 117)
```

```
DEFINITION:
isqrt-statep (s, i)
=  ((mc-status (s) = 'running)
    ∧  evenp (mc-pc (s))
    ∧  rom-addrp (mc-pc (s), mc-mem (s), 70)
    ∧  mcode-addrp (mc-pc (s), mc-mem (s), ISQRT-CODE)
    ∧  ram-addrp (sub (32, 8, read-sp (s)), mc-mem (s), 12)
    ∧  (i = iread-dn (32, 2, s))
    ∧  ilessp (1, i))
```

The function isqrt-t (i) specifies the number of instructions needed to complete this program ISQRT.

```
DEFINITION:
```

```
isqrt1-t (i, j)
= if j ≃ 0 then 0
  elseif (i ÷ j) < j
  then 10 + isqrt1-t (i, (j + (i ÷ j)) ÷ 2)
  else 8 endif
```

```
DEFINITION:
isqrt-t (i)
= let j1 be ((i ÷ 2) + (i ÷ (i ÷ 2))) ÷ 2
  in
  if i < sq (i ÷ 2) then 14 + isqrt1-t (i, j1)
  else 12 endif endlet
```

The functional behavior of the program is specified by the following function isqrt (i), which is just a formalization in the Nqthm logic of the algorithm employed.

```
DEFINITION:
isqrt1 (i, j)
= if j ≃ 0 then fix (i)
  elseif (i ÷ j) < j then isqrt1 (i, (j + (i ÷ j)) ÷ 2)
  else fix (j) endif
```

```
DEFINITION:
isqrt (i)
= let j1 be ((i ÷ 2) + (i ÷ (i ÷ 2))) ÷ 2
  in
  if i < sq (i ÷ 2) then isqrt1 (i, j1)
  else i ÷ 2 endif endlet
```

5.3.2 The Proof

We follow strictly the two-step proof outlined in Section 1. In the first step, we prove the following theorem that corresponds to the theorems **P-1** to **P-7**.

```
THEOREM: isqrt-correctness
let sn be stepn (s, isqrt-t (i))
in
isqrt-statep (s, i)
⇒ ((mc-status (sn) = 'running)
    ∧ (mc-pc (sn) = rts-addr (s))
    ∧ (read-rn (32, 14, mc-rfile (sn)) = read-an (32, 6, s))
    ∧ (read-rn (32, 15, mc-rfile (sn))
        = add (32, read-an (32, 7, s), 4))
    ∧ (d2-7a2-5p (rn)
        ⇒ (read-rn (oplen, rn, mc-rfile (sn))
            = read-rn (oplen, rn, mc-rfile (s))))
    ∧ (disjoint (x, k, sub (32, 12, read-sp (s)), 20)
```

$$\begin{array}{rl} \Rightarrow & (\text{read-mem}\,(x,\ \text{mc-mem}\,(sn),\ k) \\ & \quad = \ \text{read-mem}\,(x,\ \text{mc-mem}\,(s),\ k))) \\ \wedge & (\text{iread-dn}\,(32,\ 0,\ sn) = \text{isqrt}\,(i))) \ \textbf{endlet} \end{array}$$

In particular, the theorem above establishes that the content of data register D0 is equal to isqrt (i) after executing isqrt-t (i) instructions. In the second step, we only need to show that the Nqthm function isqrt (i) does compute the square root of an integer greater than 1, which is stated formally as follows.

THEOREM: isqrt-logic-correctness
$(1 < i) \Rightarrow ((i < \text{sq}\,(1 + \text{isqrt}\,(i))) \wedge (i \not< \text{sq}\,(\text{isqrt}\,(i))))$

5.3.3 A Simple Timing Analysis

In the same vein as the GCD example, we have proved that isqrt-t (i) is at most 322, which tells us that this program ISQRT would terminate within 322 instructions. We here assume that i is less than 2^{31}.

THEOREM: isqrt-t-ubound
$((i < \text{exp}\,(2,\ 31)) \wedge (1 < i)) \Rightarrow (\text{isqrt-t}\,(i) \leq 322)$

5.4 Binary Search

In our third example, we study binary search. The following C function `bsearch` taken from page 58 of [32] with some minor modification searches for the occurrence of a given integer in a sorted integer array. In this section, we describe the correctness proof of the object code of this C function.

```
/* bsearch:  find x in a[0] <= a[1] <= ... <= a[n-1] */
int bsearch (int x, int a[], int n)
{
  int low, high, mid;

  low = 0;
  high = n;
  while (low < high) {
    mid = (low + high) / 2;
    if (x < a[mid])
      high = mid;
    else if (x > a[mid])
      low = mid + 1;
    else return mid;
  }
  return -1;
```

```
}
```

The MC68020 assembly code generated by the Gnu C compiler with optimization.

```
<bsearch>:       linkw a6,#0
<bsearch+4>:     moveml d2-d3,sp@-
<bsearch+8>:     movel a6@(8),d3
<bsearch+12>:    moveal a6@(12),a0
<bsearch+16>:    clrl d1
<bsearch+18>:    movel a6@(16),d2
<bsearch+22>:    cmpl d1,d2
<bsearch+24>:    ble 0x232a <bsearch+58>
<bsearch+26>:    movel d1,d0
<bsearch+28>:    addl d2,d0
<bsearch+30>:    bpl 0x2312 <bsearch+34>
<bsearch+32>:    addql #1,d0
<bsearch+34>:    asrl #1,d0
<bsearch+36>:    cmpl 0(a0)[d0.l*4],d3
<bsearch+40>:    bge 0x231e <bsearch+46>
<bsearch+42>:    movel d0,d2
<bsearch+44>:    bra 0x2306 <bsearch+22>
<bsearch+46>:    cmpl 0(a0)[d0.l*4],d3
<bsearch+50>:    ble 0x232c <bsearch+60>
<bsearch+52>:    movel d0,d1
<bsearch+54>:    addql #1,d1
<bsearch+56>:    bra 0x2306 <bsearch+22>
<bsearch+58>:    movel #-1,d0
<bsearch+60>:    moveml a6@(-8),d2-d3
<bsearch+66>:    unlk a6
<bsearch+68>:    rts
```

5.4.1 The Formalization

As desribed in Section 1, we need to formalize in the Nqthm logic the preconditions, the time function, and the functional behavior of this program `bsearch`.

The function BSEARCH-CODE defines the machine code of `bsearch` as a list of unsigned integers. The function bsearch-statep (s, x, a, n, lst) characterizes the preconditions on the initial state s.

```
DEFINITION:
BSEARCH-CODE
=  '(78 86 0 0 72 231 48 0 38 46 0 8 32 110 0 12 66
     129 36 46 0 16 180 129 111 32 32 1 208 130 106 2
     82 128 226 128 182 176 12 0 108 4 36 0 96 232 182
     176 12 0 111 8 34 0 82 129 96 220 112 255 76 238
     0 12 255 248 78 94 78 117)
```

DEFINITION:
bsearch-statep (s, x, a, n, lst)

```
= ((mc-status(s) = 'running)
   ∧ evenp(mc-pc(s))
   ∧ rom-addrp(mc-pc(s), mc-mem(s), 70)
   ∧ mcode-addrp(mc-pc(s), mc-mem(s), BSEARCH-CODE)
   ∧ ram-addrp(sub(32, 12, read-sp(s)), mc-mem(s), 28)
   ∧ ram-addrp(a, mc-mem(s), 4 * n)
   ∧ mem-ilst(4, a, mc-mem(s), n, lst)
   ∧ disjoint(sub(32, 12, read-sp(s)), 28, a, 4 * n)
   ∧ (a = read-mem(add(32, read-sp(s), 8), mc-mem(s), 4))
   ∧ (n = iread-mem(add(32, read-sp(s), 12), mc-mem(s), 4))
   ∧ (x = iread-mem(add(32, read-sp(s), 4), mc-mem(s), 4))
   ∧ int-rangep(2 * n, 32)
   ∧ (n ∈ N))
```

The function bsearch-t$(x,\ n,\ lst)$ specifies the number of instructions needed to complete the execution of this program.

```
DEFINITION:
bsearch1-t(x, lst, i, j)
= let k be (i + j) ÷ 2
  in
  if i < j
  then if ilessp(x, get-nth(k, lst))
          then 10 + bsearch1-t(x, lst, i, k)
          elseif ilessp(get-nth(k, lst), x)
          then 13 + bsearch1-t(x, lst, 1 + k, j)
          else 13 endif
  else 6 endif endlet

DEFINITION:
bsearch-t(x, n, lst) = (6 + bsearch1-t(x, lst, 0, n))
```

The functional behavior of the program is specified by the following function bsearch$(x,\ n,\ lst)$, which is just a formalization in Nqthm logic of the algorithm employed.

```
DEFINITION:
bsearch1(x, lst, i, j)
= let k be (i + j) ÷ 2
  in
  if i < j
  then if ilessp(x, get-nth(k, lst)) then bsearch1(x, lst, i, k)
          elseif ilessp(get-nth(k, lst), x)
          then bsearch1(x, lst, 1 + k, j)
          else k endif
  else -1 endif endlet

DEFINITION: bsearch(x, n, lst) = bsearch1(x, lst, 0, n)
```

5.4.2 The Proof

We strictly follow the two-step proof outlined in Section 1. In the first step, we prove the following theorem that corresponds to the theorems **P-1** to **P-7**.

```
THEOREM: bsearch-correctness
let sn be stepn (s, bsearch-t (x, n, lst))
in
bsearch-statep (s, x, a, n, lst)
⇒ ((mc-status (sn) = 'running)
    ∧ (mc-pc (sn) = rts-addr (s))
    ∧ (read-rn (32, 14, mc-rfile (sn))
        = read-rn (32, 14, mc-rfile (s)))
    ∧ (read-rn (32, 15, mc-rfile (sn))
        = add (32, read-sp (s), 4))
    ∧ ((d2-7a2-5p (rn) ∧ (oplen ≤ 32))
        ⇒ (read-rn (oplen, rn, mc-rfile (sn))
            = read-rn (oplen, rn, mc-rfile (s))))
    ∧ (disjoint (x, k, sub (32, 12, read-sp (s)), 28)
        ⇒ (read-mem (x, mc-mem (sn), k)
            = read-mem (x, mc-mem (s), k)))
    ∧ (iread-dn (32, 0, sn) = bsearch (x, n, lst))) endlet
```

In particular, the theorem above has established that the content of data register D0 is equal to bsearch (x, n, lst) after executing bsearch-t (x, n, lst) instructions. In the second step, we need to show that the Nqthm function bsearch (x, n, lst) is correct with respect to the following specification:

1. If bsearch (x, n, lst) returns other than -1, then it returns an (nonnegative) integer k such that the kth element of lst is equal to the integer x.

2. If bsearch (x, n, lst) returns -1 and lst is ordered, then the integer x is not in lst.

which is stated formally and proved mechanically as the following two theorems.

```
THEOREM: bsearch-found
((bsearch (x, n, lst) ≠ -1) ∧ lst-integerp (lst) ∧ integerp (x))
⇒ (get-nth (bsearch (x, n, lst), lst) = x)
```

```
THEOREM: bsearch-not-found
((bsearch (x, len (lst), lst) = -1)
 ∧ orderedp (lst)
 ∧ lst-integerp (lst)
 ∧ integerp (x))
⇒ (x ∉ lst)
```

5.4.3 A Simple Timing Analysis

Similar to the GCD and ISQRT examples, we have proved that bsearch-t$(x,\ n,\ lst)$ is at most 435, which gives us an upper bound of the number of instructions executed by this program BSEARCH. We assume that n is less than 2^{31}.

THEOREM: bsearch-t-ubound
$(n < \exp(2, 31)) \Rightarrow (\text{bsearch-t}(x,\ n,\ lst) \leq 435)$

5.5 Quicksort

Quick Sort was our first example to consider recursion. The following C program `qsort` taken from page 87 of [32] with some minor modification sorts an integer array into ascending order. The correctness proof of the object code of this program was rather complicated. It took us a couple of weeks to come up with a proof. It seemed that our life would be much easier if we would have first studied some simpler example, something like Fibonacci numbers.

```
/* slightly modified from K&R. */
/* qsort: sort a[left]...a[right] into increasing order.  We use the middle */
/* element of each subarray for partitioning.   */
void qsort (int a[], int left, int right)
{
  int i, last, temp;

  if (left >= right)
    return;
  last = (left + right) / 2;
  temp = a[left];
  a[left] = a[last];
  a[last] = temp;
  last = left;
  for (i = left + 1; i<= right; i++)
    if (a[i] < a[left]){
      temp = a[++last];
      a[last] = a[i];
      a[i] = temp;
    };
  temp = a[left];
  a[left] = a[last];
  a[last] = temp;
  qsort(a, left, last-1);
  qsort(a, last+1, right);
}

The MC68020 assembly code generated by the Gnu C compiler with optimization.

0x22b8 <qsort>:         linkw fp,#0
```

```
0x22bc <qsort+4>:       moveml d2-d4/a2-a3,sp@-
0x22c0 <qsort+8>:       moveal fp@(8),a3
0x22c4 <qsort+12>:      movel fp@(12),d3
0x22c8 <qsort+16>:      movel fp@(16),d4
0x22cc <qsort+20>:      cmpl d3,d4
0x22ce <qsort+22>:      ble 0x2338 <qsort+128>
0x22d0 <qsort+24>:      movel d3,d2
0x22d2 <qsort+26>:      addl d4,d2
0x22d4 <qsort+28>:      bpl 0x22d8 <qsort+32>
0x22d6 <qsort+30>:      addql #1,d2
0x22d8 <qsort+32>:      asrl #1,d2
0x22da <qsort+34>:      movel 0(a3)[d3.l*4],d1
0x22de <qsort+38>:      movel 0(a3)[d2.l*4],0(a3)[d3.l*4]
0x22e4 <qsort+44>:      movel d1,0(a3)[d2.l*4]
0x22e8 <qsort+48>:      movel d3,d2
0x22ea <qsort+50>:      movel d2,d0
0x22ec <qsort+52>:      bra 0x2308 <qsort+80>
0x22ee <qsort+54>:      moveal 0(a3)[d0.l*4],a0
0x22f2 <qsort+58>:      cmpal 0(a3)[d3.l*4],a0
0x22f6 <qsort+62>:      bge 0x2308 <qsort+80>
0x22f8 <qsort+64>:      addql #1,d2
0x22fa <qsort+66>:      movel 0(a3)[d2.l*4],d1
0x22fe <qsort+70>:      movel 0(a3)[d0.l*4],0(a3)[d2.l*4]
0x2304 <qsort+76>:      movel d1,0(a3)[d0.l*4]
0x2308 <qsort+80>:      addql #1,d0
0x230a <qsort+82>:      cmpl d0,d4
0x230c <qsort+84>:      bge 0x22ee <qsort+54>
0x230e <qsort+86>:      movel 0(a3)[d3.l*4],d1
0x2312 <qsort+90>:      movel 0(a3)[d2.l*4],0(a3)[d3.l*4]
0x2318 <qsort+96>:      movel d1,0(a3)[d2.l*4]
0x231c <qsort+100>:     moveal d2,a0
0x231e <qsort+102>:     pea a0@(-1)
0x2322 <qsort+106>:     movel d3,sp@-
0x2324 <qsort+108>:     movel a3,sp@-
0x2326 <qsort+110>:     lea 0x22b8 <qsort>,a2
0x232a <qsort+114>:     jsr a2@
0x232c <qsort+116>:     movel d4,sp@-
0x232e <qsort+118>:     moveal d2,a0
0x2330 <qsort+120>:     pea a0@(1)
0x2334 <qsort+124>:     movel a3,sp@-
0x2336 <qsort+126>:     jsr a2@
0x2338 <qsort+128>:     moveml fp@(-20),d2-d4/a2-a3
0x233e <qsort+134>:     unlk fp
0x2340 <qsort+136>:     rts
```

5.5.1 The Formalization

According to our formulation, we need to formalize in the Nqthm logic the preconditions, the time function, and the functional behavior of this program `qsort`.

The function QSORT-CODE represents the machine code of `qsort` as a list

of unsigned integers. The function qstack $(l,\ r,\ lst)$ specifies the stack space needed for the program. The function qsort-statep $(s,\ a,\ l,\ r,\ n,\ lst)$ characterizes the preconditions of the initial state s.

```
DEFINITION:
QSORT-CODE
= '(78 86 0 0 72 231 56 48 38 110 0 8 38 46 0 12 40
    46 0 16 184 131 111 104 36 3 212 132 106 2 82 130
    226 130 34 51 60 0 39 179 44 0 60 0 39 129 44 0
    36 3 32 2 96 26 32 115 12 0 177 243 60 0 108 16
    82 130 34 51 44 0 39 179 12 0 44 0 39 129 12 0 82
    128 184 128 108 224 34 51 60 0 39 179 44 0 60 0
    39 129 44 0 32 66 72 104 255 255 47 3 47 11 69
    250 255 144 78 146 47 4 32 66 72 104 0 1 47 11 78
    146 76 238 12 28 255 236 78 94 78 117)
```

```
DEFINITION:
qstack (l, r, lst)
= let last be qlast (l, r, lst),
      lst1 be qpart (l, r, lst)
  in
  if l < r
  then max (40 + qstack (l, last − 1, lst1),
            52 + qstack (1 + last, r, qsort (l, last − 1, lst1)))
  else 68 endif endlet
```

```
DEFINITION:
qsort-statep (s, a, l, r, n, lst)
= let sp be sub (32, qstack (l, r, lst) − 16, read-sp (s))
  in
  (mc-status (s) = 'running)
  ∧ evenp (mc-pc (s))
  ∧ rom-addrp (mc-pc (s), mc-mem (s), 138)
  ∧ mcode-addrp (mc-pc (s), mc-mem (s), QSORT-CODE)
  ∧ ram-addrp (a, mc-mem (s), 4 * n)
  ∧ mem-ilst (4, a, mc-mem (s), n, lst)
  ∧ ram-addrp (sp, mc-mem (s), qstack (l, r, lst))
  ∧ disjoint (a, 4 * n, sp, qstack (l, r, lst))
  ∧ (a = read-mem (add (32, read-sp (s), 4), mc-mem (s), 4))
  ∧ (l = iread-mem (add (32, read-sp (s), 8), mc-mem (s), 4))
  ∧ (r = iread-mem (add (32, read-sp (s), 12), mc-mem (s), 4))
  ∧ (qstack (l, r, lst) < exp (2, 32))
  ∧ (l ∈ N)
  ∧ (r < n)
  ∧ uint-rangep (4 * n, 32) endlet
```

The function qsort-t $(l,\ r,\ lst)$ specifies the number of instructions needed to complete the execution of this program `qsort`.

```
DEFINITION:
```

```
qpart-aux-t (a, l, r, n, lst, last, i)
= if r < i then 11
  elseif ilessp (get-nth (i, lst), get-nth (l, lst))
  then 10 + qpart-aux-t (a,
                         l,
                         r,
                         n,
                         swap (1 + last, i, lst),
                         1 + last,
                         1 + i)
  else 6 + qpart-aux-t (a, l, r, n, lst, last, 1 + i) endif

Definition:
qpart-t (a, l, r, n, lst)
= let lst1 be swap (l, (l + r) ÷ 2, lst)
  in
  18 + qpart-aux-t (a, l, r, n, lst1, l, 1 + l) endlet

Definition: qsort-10 (a, l, r, n, lst) = 10

Definition: qsort-5 (a, l, r, n, lst) = 5

Definition: qsort-3 (a, l, r, n, lst) = 3

Definition:
qsort-t (a, l, r, n, lst)
= let last be qlast (l, r, lst),
      qlst be qpart (l, r, lst)
  in
  if l < r
  then qpart-t (a, l, r, n, lst)
       + (qsort-t (a, l, last - 1, n, qlst)
          + (qsort-5 (a, l, r, n, lst)
             + (qsort-t (a,
                         1 + last,
                         r,
                         n,
                         qsort (l, last - 1, qlst))
                + qsort-3 (a, l, r, n, lst))))
  else qsort-10 (a, l, r, n, lst) endif endlet
```

The functional behavior of this program is specified by the following function qsort (*a*, *l*, *r*, *n*, *lst*), which is just a formalization in Nqthm logic of the algorithm employed.

```
Definition:
qpart-aux (l, r, lst, last, i)
= if r < i then swap (l, last, lst)
  elseif ilessp (get-nth (i, lst), get-nth (l, lst))
  then qpart-aux (l, r, swap (1 + last, i, lst), 1 + last, 1 + i)
  else qpart-aux (l, r, lst, last, 1 + i) endif
```

```
DEFINITION:
qpart (l, r, lst) = qpart-aux (l, r, swap (l, (l + r) ÷ 2, lst), l, 1 + l)

DEFINITION:
qlast-aux (l, r, lst, last, i)
= if r < i then fix (last)
  elseif ilessp (get-nth (i, lst), get-nth (l, lst))
  then qlast-aux (l, r, swap (1 + last, i, lst), 1 + last, 1 + i)
  else qlast-aux (l, r, lst, last, 1 + i) endif

DEFINITION:
qlast (l, r, lst) = qlast-aux (l, r, swap (l, (l + r) ÷ 2, lst), l, 1 + l)

DEFINITION:
qsort (l, r, lst)
= if l < r
  then qsort (1 + qlast (l, r, lst),
              r,
              qsort (l, qlast (l, r, lst) − 1, qpart (l, r, lst)))
  else lst endif
```

5.5.2 The Proof

We follow strictly the two-step proof outlined in Section 1. In the first step, we prove the following theorem that corresponds to the theorems **P-1** to **P-7**.

```
THEOREM: qsort-correctness
let sn be stepn (s, qsort-t (a, l, r, n, lst)),
    sp be sub (32, qstack (l, r, lst) − 16, read-sp (s))
in
qsort-statep (s, a, l, r, n, lst)
⇒ ((mc-status (sn) = 'running)
   ∧ (mc-pc (sn) = rts-addr (s))
   ∧ (read-rn (32, 14, mc-rfile (sn))
        = read-rn (32, 14, mc-rfile (s)))
   ∧ (read-rn (32, 15, mc-rfile (sn))
        = add (32, read-rn (32, 15, mc-rfile (s)), 4))
   ∧ (((oplen ≤ 32) ∧ d2-7a2-5p (rn))
        ⇒ (read-rn (oplen, rn, mc-rfile (sn))
             = read-rn (oplen, rn, mc-rfile (s))))
   ∧ ((disjoint (sp, qstack (l, r, lst), x, k)
         ∧ disjoint (a, 4 * n, x, k))
        ⇒ (read-mem (x, mc-mem (sn), k)
             = read-mem (x, mc-mem (s), k)))
   ∧ mem-ilst (4, a, mc-mem (sn), n, qsort (l, r, lst))) endlet
```

In particular, the above theorem has established that the content of the array a is equal to qsort (l, r, lst) after executing qsort-t (a, l, r, n, lst) instructions. In the second step, we need to show that the Nqthm function qsort (l, r, lst) does sort the given integer list lst, which is stated formally as follows.

DEFINITION:
orderedp1 (l, r, lst)
= **if** $r \leq l$ **then t**
else ileq (get-nth (l, lst), get-nth ($1 + l$, lst))
$\wedge$ orderedp1 ($1 + l$, r, lst) **endif**

THEOREM: qsort-orderedp1
orderedp1 ($left$, $right$, qsort ($left$, $right$, lst))

DEFINITION:
count-lst (x, l, r, lst)
= **if** $r < l$ **then** 0
elseif x = get-nth (l, lst) **then** 1 + count-lst (x, $1 + l$, r, lst)
else count-lst (x, $1 + l$, r, lst) **endif**

THEOREM: count-lst-qsort
count-lst (x, l, r, qsort (l, r, lst)) = count-lst (x, l, r, lst)

Roughly speaking, the theorem 'qsort-orderedp1' asserts that the list qsort ($left$, $right$, lst) is in ascending order; the theorem 'count-lst-qsort' asserts that qsort (l, r, lst) is a permutation of lst. The proof of these two theorems required many supporting lemmas. We refer the interested readers to Appendix C.4.

5.5.3 A Simple Stack Space Analysis

We have seen, in the preceding examples, how to prove time bounds for machine code programs. Another very important issue addressed explicitly in machine code program proving but not in high-level program proving is the memory space requirement. While this has been quite simple in the other examples in this chapter, the stack space required by `qsort` is given as the recursive function qstack (l, r, lst), and some sort of formal analysis is desirable.

We have mechanically proved the following theorem, which asserts that the size of the stack needed for any correct execution of `qsort` is at most $52(r - l) + 52$ bytes, where l and r are the lower and upper bounds of the array, respectively.

THEOREM: qstack-ubound
qstack (l, r, lst) $\leq$ (68 + (52 * (r − l)))

The proof of the above theorem is by induction, and Nqthm automatically finds the right induction schema. We need to prove two key lemmas for each of the two inductive cases in the proof.

THEOREM: qstack-ubound-la-1
$(l < r)$
$\Rightarrow \ ((52 * (r - l))$
$\not< \ (52 + (52 * ((\text{qlast}\,(l,\ r,\ lst) - 1) - l))))$

THEOREM: qstack-ubound-la-2
$(l < r)$
$\Rightarrow \ ((52 * (r - l))$
$\not< \ (52 + (52 * (r - (1 + \text{qlast}\,(l,\ r,\ lst))))))$

5.6 The Boyer-Moore Majority Voting Algorithm

The last example in this chapter is the correctness proof of the object code of the following C program `mjrty` that implements the majority voting algorithm invented and mechanically proved correct by Boyer and Moore [10]. This small program can be used to determine if there is a candidate who has received a majority of votes cast in an election.

```
/* a majority voting algorithm by Boyer and Moore */
#define YES 1
#define NO  0

struct winner {
  int x;
  int y;
};

struct winner mjrty (int a[], int n)
{
  int cand, i, k;
  struct winner temp;

  k = 0;
  for (i = 0; i < n; i++)
    if (k == 0) {
      cand = a[i];
      k = 1;
    }
  else {
    if (cand == a[i])
      k++;
    else
      k--;
  };
  temp.x = cand;
  if (k == 0) {
    temp.y = NO;
    return temp;
  };
```

```
  if (k > n/2) {
    temp.y = YES;
    return temp;
  };
  k = 0;
  for (i = 0; i < n; i++)
    if (a[i] == cand)
      k++;
  if (k > n/2)
    temp.y = YES;
  else temp.y = NO;
  return temp;
}
```

The MC68020 assembly code generated by Gnu C compiler with optimization.

```
0x2310 <mjrty>:         linkw a6,#0
0x2314 <mjrty+4>:       moveml d2-d5,sp@-
0x2318 <mjrty+8>:       moveal a6@(8),a0
0x231c <mjrty+12>:      movel a6@(12),d2
0x2320 <mjrty+16>:      clrl d1
0x2322 <mjrty+18>:      clrl d0
0x2324 <mjrty+20>:      cmpl d0,d2
0x2326 <mjrty+22>:      ble 0x2346 <mjrty+54>
0x2328 <mjrty+24>:      tstl d1
0x232a <mjrty+26>:      bne 0x2334 <mjrty+36>
0x232c <mjrty+28>:      movel 0(a0)[d0.l*4],d3
0x2330 <mjrty+32>:      movel #1,d1
0x2332 <mjrty+34>:      bra 0x2340 <mjrty+48>
0x2334 <mjrty+36>:      cmpl 0(a0)[d0.l*4],d3
0x2338 <mjrty+40>:      bne 0x233e <mjrty+46>
0x233a <mjrty+42>:      addql #1,d1
0x233c <mjrty+44>:      bra 0x2340 <mjrty+48>
0x233e <mjrty+46>:      subl #1,d1
0x2340 <mjrty+48>:      addql #1,d0
0x2342 <mjrty+50>:      cmpl d0,d2
0x2344 <mjrty+52>:      bgt 0x2328 <mjrty+24>
0x2346 <mjrty+54>:      movel d3,d4
0x2348 <mjrty+56>:      tstl d1
0x234a <mjrty+58>:      beq 0x2382 <mjrty+114>
0x234c <mjrty+60>:      movel d2,d0
0x234e <mjrty+62>:      bge 0x2352 <mjrty+66>
0x2350 <mjrty+64>:      addql #1,d0
0x2352 <mjrty+66>:      asrl #1,d0
0x2354 <mjrty+68>:      cmpl d1,d0
0x2356 <mjrty+70>:      bge 0x235c <mjrty+76>
0x2358 <mjrty+72>:      movel #1,d5
0x235a <mjrty+74>:      bra 0x2384 <mjrty+116>
0x235c <mjrty+76>:      clrl d1
0x235e <mjrty+78>:      clrl d0
0x2360 <mjrty+80>:      cmpl d0,d2
0x2362 <mjrty+82>:      ble 0x2372 <mjrty+98>
0x2364 <mjrty+84>:      cmpl 0(a0)[d0.l*4],d3
0x2368 <mjrty+88>:      bne 0x236c <mjrty+92>
```

```
0x236a <mjrty+90>:      addql #1,d1
0x236c <mjrty+92>:      addql #1,d0
0x236e <mjrty+94>:      cmpl d0,d2
0x2370 <mjrty+96>:      bgt 0x2364 <mjrty+84>
0x2372 <mjrty+98>:      movel d2,d0
0x2374 <mjrty+100>:     bge 0x2378 <mjrty+104>
0x2376 <mjrty+102>:     addql #1,d0
0x2378 <mjrty+104>:     asrl #1,d0
0x237a <mjrty+106>:     cmpl d1,d0
0x237c <mjrty+108>:     bge 0x2382 <mjrty+114>
0x237e <mjrty+110>:     movel #1,d5
0x2380 <mjrty+112>:     bra 0x2384 <mjrty+116>
0x2382 <mjrty+114>:     clrl d5
0x2384 <mjrty+116>:     movel d4,d0
0x2386 <mjrty+118>:     movel d5,d1
0x2388 <mjrty+120>:     moveml a6@(-16),d2-d5
0x238e <mjrty+126>:     unlk a6
0x2390 <mjrty+128>:     rts
```

5.6.1 The Formalization

According to our formulation, we need to formalize in the Nqthm logic the preconditions, the time function, and the functional behavior of this `mjrty` program.

The function MJRTY-CODE defines the machine code of `mjrty` as a list of unsigned integers. The function mjrty-statep(s, a, n, lst) characterizes the preconditions of the initial state s.

```
Definition:
mjrty-code
= '(78 86 0 0 72 231 60 0 32 110 0 8 36 46 0 12 66
    129 66 128 180 128 111 30 74 129 102 8 38 48 12 0
    114 1 96 12 182 176 12 0 102 4 82 129 96 2 83 129
    82 128 180 128 110 226 40 3 74 129 103 54 32 2
    108 2 82 128 226 128 176 129 108 4 122 1 96 40 66
    129 66 128 180 128 111 14 182 176 12 0 102 2 82
    129 82 128 180 128 110 242 32 2 108 2 82 128 226
    128 176 129 108 4 122 1 96 2 66 133 32 4 34 5 76
    238 0 60 255 240 78 94 78 117)
```

```
Definition:
mjrty-statep(s, a, n, lst)
= ((mc-status(s) = 'running)
   ∧ evenp(mc-pc(s))
   ∧ rom-addrp(mc-pc(s), mc-mem(s), 130)
   ∧ mcode-addrp(mc-pc(s), mc-mem(s), MJRTY-CODE)
   ∧ ram-addrp(sub(32, 20, read-sp(s)), mc-mem(s), 32)
   ∧ ram-addrp(a, mc-mem(s), 4 * n)
   ∧ mem-ilst(4, a, mc-mem(s), n, lst)
   ∧ disjoint(a, 4 * n, sub(32, 20, read-sp(s)), 32)
```

$\wedge$ $(a = \text{read-mem}(\text{add}(32, \text{read-sp}(s), 4), \text{mc-mem}(s), 4))$
$\wedge$ $(n = \text{iread-mem}(\text{add}(32, \text{read-sp}(s), 8), \text{mc-mem}(s), 4))$
$\wedge$ $(n \not\simeq 0))$

The function mjrty-t(a, n, lst) specifies the number of instructions needed to complete the execution of this program.

```
DEFINITION:
mjrty-cand-t(a, n, lst, cand, i, k)
= if i < n
  then if k ≃ 0
       then let cand1 be get-nth(i, lst)
            in
            8 + mjrty-cand-t(a, n, lst, cand1, 1 + i, 1) endlet
       elseif cand = get-nth(i, lst)
       then 9 + mjrty-cand-t(a, n, lst, cand, 1 + i, 1 + k)
       else 8 + mjrty-cand-t(a, n, lst, cand, 1 + i, k − 1) endif
  elseif cand = get-nth(0, lst) then 18
  else 17 endif

DEFINITION:
mjrty-sn-t(a, n, lst, cand, i, k)
= if i < n
  then if k ≃ 0
       then let cand1 be get-nth(i, lst)
            in
            8 + mjrty-sn-t(a, n, lst, cand1, 1 + i, 1) endlet
       elseif cand = get-nth(i, lst)
       then 9 + mjrty-sn-t(a, n, lst, cand, 1 + i, 1 + k)
       else 8 + mjrty-sn-t(a, n, lst, cand, 1 + i, k − 1) endif
  elseif k ≃ 0 then 11
  else 17 endif

DEFINITION:
cand-cnt-t(a, n, lst, cand, i, k)
= if i < n
  then if cand = get-nth(i, lst)
       then 6 + cand-cnt-t(a, n, lst, cand, 1 + i, 1 + k)
       else 5 + cand-cnt-t(a, n, lst, cand, 1 + i, k) endif
  elseif (n ÷ 2) < k then 14
  else 13 endif

DEFINITION:
mjrty-t(a, n, lst)
= let cand be get-nth(0, lst)
  in
  14 + if (mjrty-k(n, lst, cand, 1, 1) ≃ 0)
          ∨ ((n ÷ 2) < mjrty-k(n, lst, cand, 1, 1))
       then mjrty-sn-t(a, n, lst, cand, 1, 1)
       else mjrty-cand-t(a, n, lst, cand, 1, 1)
            + if cand = mjrty-cand(n,
```

```
                                lst,
                                cand,
                                1,
                                1)
then cand-cnt-t (a,
                 n,
                 lst,
                 mjrty-cand (n,
                             lst,
                             cand,
                             1,
                             1),
                 1,
                 1)
else cand-cnt-t (a,
                 n,
                 lst,
                 mjrty-cand (n,
                             lst,
                             cand,
                             1,
                             1),
                 1,
                 0) endif endif endlet
```

The functional behavior of the program is specified by the following functions mjrty-cand $(n,\ lst,\ cand,\ i,\ k)$ and mjrty-p $(n,\ lst,\ cand,\ i,\ k)$, which are just a formalization in the Nqthm logic of the algorithm employed.

```
Definition:
mjrty-cand (n, lst, cand, i, k)
= if i < n
    then if k ≃ 0 then mjrty-cand (n, lst, get-nth (i, lst), 1 + i, 1)
           elseif cand = get-nth (i, lst)
           then mjrty-cand (n, lst, cand, 1 + i, 1 + k)
           else mjrty-cand (n, lst, cand, 1 + i, k − 1) endif
    else cand endif
```

```
Definition:
mjrty-k (n, lst, cand, i, k)
= if i < n
    then if k ≃ 0 then mjrty-k (n, lst, get-nth (i, lst), 1 + i, 1)
           elseif cand = get-nth (i, lst)
           then mjrty-k (n, lst, cand, 1 + i, 1 + k)
           else mjrty-k (n, lst, cand, 1 + i, k − 1) endif
    else k endif
```

```
Definition:
cand-cnt (n, lst, cand, i, k)
= if i < n
    then if cand = get-nth (i, lst)
```

```
        then cand-cnt(n, lst, cand, 1 + i, 1 + k)
        else cand-cnt(n, lst, cand, 1 + i, k) endif
    else k endif

DEFINITION:
mjrty-p(n, lst, cand, i, k)
= if mjrty-k(n, lst, cand, i, k) ≃ 0 then f
  elseif (n ÷ 2) < mjrty-k(n, lst, cand, i, k) then t
  else (n ÷ 2)
       < cand-cnt(n, lst, mjrty-cand(n, lst, cand, i, k), i, k) endif
```

5.6.2 The Proof

We follow strictly the two-step proof outlined in Section 1. In the first step, we prove the following theorem that corresponds to the theorems **P-1** to **P-7**.

```
THEOREM: mjrty-correctness
let sn be stepn(s, mjrty-t(a, n, lst))
in
mjrty-statep(s, a, n, lst)
⇒ ((mc-status(sn) = 'running)
   ∧ (mc-pc(sn) = rts-addr(s))
   ∧ (read-rn(32, 14, mc-rfile(sn))
      = read-rn(32, 14, mc-rfile(s)))
   ∧ (read-rn(32, 15, mc-rfile(sn))
      = add(32, read-sp(s), 4))
   ∧ ((d2-7a2-5p(rn) ∧ (oplen ≤ 32))
      ⇒ (read-rn(oplen, rn, mc-rfile(sn))
         = read-rn(oplen, rn, mc-rfile(s))))
   ∧ (disjoint(sub(32, 20, read-sp(s)), 32, x, k)
      ⇒ (read-mem(x, mc-mem(sn), k)
         = read-mem(x, mc-mem(s), k)))
   ∧ (iread-dn(32, 0, sn) = mjrty-cand(n, lst, 0, 0, 0))
   ∧ (iread-dn(32, 1, sn)
      = if mjrty-p(n, lst, 0, 0, 0) then 1
        else 0 endif)) endlet
```

In particular, the above theorem has established that the content of data register D0 is equal to mjrty-cand$(n, lst, 0, 0, 0)$ and the content of data register D1 is equivalent to mjrty-p$(n, lst, 0, 0, 0)$ after executing mjrty-t(a, n, lst) instructions from an initial state s satisfying mjrty-statep(s, a, n, lst).

In the second step, we need to prove the correctness of the Nqthm function 'mjrty-cand' and 'mjrty-p' according to the following specification.

1. If the function mjrty-p$(n, lst, 0, 0, 0)$ is true, then the function mjrty-cand$(n, lst, 0, 0, 0)$ returns the candidate who wins the majority.

2. If the function mjrty-p$(n,\ lst,\ 0,\ 0,\ 0)$ is false, then no one wins the majority.

which is given formally as the following two theorems.

THEOREM: mjrty-thm1
mjrty-p$(n,\ lst,\ 0,\ 0,\ 0)$
$\Rightarrow\ \ ((n \div 2) <$ cand-cnt$(n,\ lst,$ mjrty-cand$(n,\ lst,\ 0,\ 0,\ 0),\ 0,\ 0))$

THEOREM: mjrty-thm2
$(\neg$ mjrty-p$(n,\ lst,\ 0,\ 0,\ 0)) \Rightarrow ((n \div 2) \not< $ cand-cnt$(n,\ lst,\ x,\ 0,\ 0))$

5.6.3 A Simple Timing Analysis

We now return to the sort of timing analysis we have done in the previous few examples. Intuitively, the following theorem says that the program `mjrty` terminates within $46 + (15 * (n - 1))$ instructions, where n is, say, the number of votes cast in an election.

THEOREM: mjrty-t-ubound
mjrty-t$(a,\ n,\ lst) \leq (46 + (15 * (n - 1)))$

The proof of the above theorem `mjrty-t-ubound` is quite simple. We need to prove three lemmas that establish the upper bounds of the three time functions cand-cnt-t$(a,\ n,\ lst,\ cand,\ i,\ k)$, mjrty-cand-t$(a,\ n,\ lst,\ cand,\ i,\ k)$, mjrty-sn-t$(a,\ n,\ lst,\ cand,\ i,\ k)$ used in the definition of mjrty-t$(a,\ n,\ lst)$.

THEOREM: cand-cnt-t-ubound
$(14 + (6 * (n - i))) \not< $ cand-cnt-t$(a,\ n,\ lst,\ cand,\ i,\ k)$

THEOREM: mjrty-cand-t-ubound
$(18 + (9 * (n - i))) \not< $ mjrty-cand-t$(a,\ n,\ lst,\ cand,\ i,\ k)$

THEOREM: mjrty-sn-t-ubound
$(17 + (9 * (n - i))) \not< $ mjrty-sn-t$(a,\ n,\ lst,\ cand,\ i,\ k)$

The proofs of the above three lemmas are straightforward. We do not elaborate on their proofs.

Chapter 6

Issues in Machine Code Program Proving

The idea of verifying the object code produced by high-level programming language compilers effectively eliminates the need to give useful mathematical semantics for high-level programming languages, since the behavior of a given program is directly determined by the processor model on which the program executes, and hence can be analyzed at the processor level. By recasting every high-level language construct into the clearly understood world of machine integers in a single addressing space, we certainly simplify many subtle semantics issues, such as evaluation orders, pointers and aliasing, that have long perplexed the formal specification and verification community. But, on the other hand, using a computing model at the machine-code level seems dramatically to increase the complexity of program proving because of the loss of some abstractions. The question is, therefore, what have we actually gained by adopting this machine code approach. In attempting to address this question, we focus on some specific issues in program semantics and program proving, and study them at the machine-code level using some simple examples.

There are four sections in this chapter, each of which addresses one program proving problem that we feel is important and interesting. In these sections, we discuss the verification of some simple programs that illustrate how we handle those program proving problems at the machine-code level. The examples used in this chapter are toy programs designed just for the purpose of exposition. The full proof script of these examples is given in Appendix C.

6.1 Subroutine Calling

Composing proofs in program proving is as essential as composing programs in programming. Handling subroutine calling in machine code program proving has been one of the main considerations in our formulation of correctness for machine code programs. The correctness theorem of a subroutine should characterize the behavior of the subroutine well enough so that the same theorem can be used

repeatedly in the proof of a large class of programs that call the subroutine, just as the same subroutine can be used repeatedly in many programs. In this section, we use an extremely simple example to illustrate how to handle subroutine calling in our formalization. To some extent, we have encountered this problem in the `qsort` example of Chapter 5. But we have avoided discussing it there.

Let us consider the following program GCD3 that computes the greatest common divisor of three nonnegative integers by calling the already proved GCD twice. We here want to prove the correctness of GCD3 using the correctness theorem of GCD, given in Chapter 5.

```
/* Compute the GCD of the three nonnegative integers. */
gcd3(a, b, c)
long int a, b, c;
{
  gcd(gcd(a, b), c);
}
```

```
The MC68020 assembly code generated by the Gnu C compiler with optimization.

0x2324 <gcd3>:          linkw a6,#0
0x2328 <gcd3+4>:        movel a2,sp@-
0x232a <gcd3+6>:        movel a6@(16),sp@-
0x232e <gcd3+10>:       movel a6@(12),sp@-
0x2332 <gcd3+14>:       movel a6@(8),sp@-
0x2336 <gcd3+18>:       lea @#0x2350 <gcd>,a2
0x233c <gcd3+24>:       jsr a2@
0x233e <gcd3+26>:       addqw #8,sp
0x2340 <gcd3+28>:       movel d0,sp@-
0x2342 <gcd3+30>:       jsr a2@
0x2344 <gcd3+32>:       moveal a6@(-4),a2
0x2348 <gcd3+36>:       unlk a6
0x234a <gcd3+38>:       rts
```

We follow the formulation we have discussed in Chapter 5. 'gcd3-code' formalizes the machine code of GCD3, but with a 'hole' that is represented by the four -1's. The 'hole' is intended for the location of the function GCD, and is specified elsewhere in the function 'gcd3-statep'. The functions 'gcd3-load' and 'gcd3-statep' together formalize the preconditions on the initial state. In particular, we have specified that the longword at location (GCD3-ADDR $+20$) be GCD-ADDR.

```
DEFINITION:
GCD3-CODE
=  '(78 86 0 0 47 10 47 46 0 16 47 46 0 12 47 46 0 8
     69 249 -1 -1 -1 -1 78 146 80 79 47 0 78 146 36
     110 255 252 78 94 78 117)
```

```
CONSERVATIVE AXIOM: gcd3-load
 gcd3-loadp (s)
 =  (evenp (GCD3-ADDR)
     ∧  (GCD3-ADDR ∈ N)
     ∧  nat-rangep (GCD3-ADDR, 32)
     ∧  rom-addrp (GCD3-ADDR, mc-mem (s), 40)
     ∧  mcode-addrp (GCD3-ADDR, mc-mem (s), GCD3-CODE)
     ∧  gcd-loadp (s)
     ∧  (pc-read-mem (add (32, GCD3-ADDR, 20), mc-mem (s), 4)
         =  GCD-ADDR))
```

Simultaneously, we introduce the new function symbols 'gcd3-loadp' and 'gcd3-addr'.

```
DEFINITION:
gcd3-statep (s, a, b, c)
=  ((mc-status (s) = 'running)
    ∧  gcd3-loadp (s)
    ∧  (mc-pc (s) = GCD3-ADDR)
    ∧  ram-addrp (sub (32, 36, read-sp (s)), mc-mem (s), 52)
    ∧  (a = iread-mem (add (32, read-sp (s), 4), mc-mem (s), 4))
    ∧  (b = iread-mem (add (32, read-sp (s), 8), mc-mem (s), 4))
    ∧  (c = iread-mem (add (32, read-sp (s), 12), mc-mem (s), 4))
    ∧  (0 < a)
    ∧  (0 < b)
    ∧  (0 < c))
```

The time function of GCD3 is defined as follows. The function gcd3-t (a, b, c) gives the total number of instructions executed by GCD3. The functions gcd3-t1 (a, b, c) and gcd3-t3 (a, b, c) reflect the two subroutine calls to GCD in GCD3.

```
DEFINITION:  gcd3-t0 (a, b, c) = 7

DEFINITION:  gcd3-t1 (a, b, c) = gcd-t (a, b)

DEFINITION:  gcd3-t2 (a, b, c) = 3

DEFINITION:  gcd3-t3 (a, b, c) = gcd-t (gcd (a, b), c)

DEFINITION:  gcd3-t4 (a, b, c) = 3

DEFINITION:
gcd3-t (a, b, c)
=  (gcd3-t0 (a, b, c)
    +  (gcd3-t1 (a, b, c)
        +  (gcd3-t2 (a, b, c)
            +  (gcd3-t3 (a, b, c) + gcd3-t4 (a, b, c)))))
```

The functional behavior of GCD3 is specified by the following function 'gcd3'.

DEFINITION: gcd3 (a, b, c) = gcd (gcd (a, b), c)

The correctness of GCD3 is then given by the following three theorems.

```
THEOREM: gcd3-correctness
let sn be stepn (s, gcd3-t (a, b, c))
in
gcd3-statep (s, a, b, c)
⇒ ((mc-status (sn) = 'running)
    ∧ (mc-pc (sn) = rts-addr (s))
    ∧ (read-rn (32, 14, mc-rfile (sn))
        = read-rn (32, 14, mc-rfile (s)))
    ∧ (read-rn (32, 15, mc-rfile (sn))
        = add (32, read-rn (32, 15, mc-rfile (s)), 4))
    ∧ (((oplen ≤ 32) ∧ d2-7a2-5p (rn))
        ⇒ (read-rn (oplen, rn, mc-rfile (sn))
            = read-rn (oplen, rn, mc-rfile (s))))
    ∧ (disjoint (x, k, sub (32, 36, read-sp (s)), 52)
        ⇒ (read-mem (x, mc-mem (sn), k)
            = read-mem (x, mc-mem (s), k)))
    ∧ (iread-dn (32, 0, sn) = gcd (gcd (a, b), c))) endlet

THEOREM: gcd3-is-cd
((a mod gcd3 (a, b, c)) = 0)
∧ ((b mod gcd3 (a, b, c)) = 0)
∧ ((c mod gcd3 (a, b, c)) = 0)

THEOREM: gcd3-the-greatest
((a ≠ 0)
 ∧ (b ≠ 0)
 ∧ (c ≠ 0)
 ∧ ((a mod x) = 0)
 ∧ ((b mod x) = 0)
 ∧ ((c mod x) = 0))
⇒ (gcd3 (a, b, c) ≮ x)
```

The theorem 'gcd3-correctness' proved that the content of the data register D0 is equal to gcd (gcd (a, b), c). The theorems 'gcd3-is-cd' and 'gcd3-the-greatest' proved further that gcd3 (a, b, c) does compute the greatest common divisor of three nonnegative integers.

To explain the use of the theorem 'gcd-correctness' in the proof of the theorem 'gcd3-correctness', let us look at the first subroutine call to GCD in GCD3. As shown in Figure 6.1, we introduce a pair of intermediate states $s0$ and $s1$: $s0$ denotes stepn (s, gcd3-t0 (a, b, c)), the machine state right before the execution of the subprogram GCD; $s1$ denotes stepn ($s0$, gcd3-t1 (a, b, c)), the machine state right after the execution of the subprogram GCD. The properties of these two intermediate states are characterized by gcd3-s0p (s, a, b, c) and gcd3-s1p (s, a, b, c), respectively.

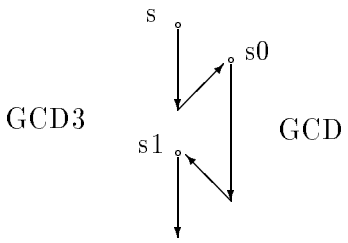

Figure 6.1: How to Use the Correctness of GCD in GCD3

DEFINITION:
gcd3-s0p (s, a, b, c)
= ((mc-status (s) = 'running)
∧ gcd3-loadp (s)
∧ (mc-pc (s) = GCD-ADDR)
∧ (read-an (32, 2, s) = GCD-ADDR)
∧ (rts-addr (s) = add (32, GCD3-ADDR, 26))
∧ ram-addrp (sub (32, 12, read-sp (s)), mc-mem (s), 52)
∧ equal* (read-an (32, 6, s), add (32, read-sp (s), 20))
∧ (a = iread-mem (add (32, read-sp (s), 4), mc-mem (s), 4))
∧ (b = iread-mem (add (32, read-sp (s), 8), mc-mem (s), 4))
∧ (c = iread-mem (add (32, read-sp (s), 12), mc-mem (s), 4))
∧ (0 < a)
∧ (0 < b)
∧ (0 < c))

DEFINITION:
gcd3-s1p (s, a, b, c)
= ((mc-status (s) = 'running)
∧ gcd3-loadp (s)
∧ (read-an (32, 2, s) = GCD-ADDR)
∧ (mc-pc (s) = add (32, GCD3-ADDR, 26))
∧ ram-addrp (sub (32, 16, read-sp (s)), mc-mem (s), 52)
∧ equal* (read-an (32, 6, s), add (32, read-sp (s), 16))
∧ (iread-dn (32, 0, s) = gcd (a, b))
∧ (c = iread-mem (add (32, read-sp (s), 8), mc-mem (s), 4))
∧ (0 < a)
∧ (0 < b)
∧ (0 < c))

Therefore, if we want to prove, for example, gcd3-statep(s, a, b, c) ⇒ gcd3-s1p($s1$, a, b, c), we can first prove two lemmas gcd3-statep(s, a, b, c) ⇒ gcd3-s0p($s0$, a, b, c) and gcd3-s0p($s0$, a, b, c) ⇒ gcd3-s1p($s1$, a, b, c), and then the goal is proved by composing these two lemmas. The second lemma is merely something about the subprogram GCD, and therefore can be proved automatically by 'gcd-correctness'.

6.2 Functional Parameters

Taking functions as arguments has long perplexed the programming language community, and the current theoretical solutions to its semantics are subtle. Many formal program verification systems have deliberately avoided considering this issue by simply working on a language subset with this functional parameter feature excluded.[5, 16] As far as we can tell, handling functional parameter in machine code program proving could be at least as difficult as program proving at higher levels. In this section, we address this important issue in the context of machine code program proving.

Our solution is quite intuitive. At the machine-code level, functional parameters can be simply viewed as pointers to programs in the memory. To verify a program that takes functions as arguments, we first assert the correctness of the functional parameters using constraint definitions. Under the constrains introduced by those constraint definitions, the correctness of the program can be proved. To verify the correctness of specific functional instances of the program, we can repeatedly use the correctness theorem of the program by substituting the functional parameters of that program by *specific* functions as long as these functions meet the constrains imposed by the constraint definitions of the functional parameters.

But the mechanization of the above idea is extremely difficult. To explain it, let us look at a very simple example. The following C function `max` compares two integers `a` and `b` using the functional parameter `comp`, and returns the "larger" one accordingly. Our aim is to prove the correctness of its binary.

```
max(int a, int b, int (*comp)(int, int))
{
  if ((*comp)(a, b) < 0)
    return b;
  else return a;
}
```

```
The MC68020 assembly code of the C function max on SUN-3 is given as
follows.  This binary is generated by "gcc -O".
```

```
0x2320 <max>:           linkw fp,#0
0x2324 <max+4>:         moveml d2-d3,sp@-
0x2328 <max+8>:         movel fp@(8),d3
0x232c <max+12>:        movel fp@(12),d2
0x2330 <max+16>:        movel d2,sp@-
0x2332 <max+18>:        movel d3,sp@-
0x2334 <max+20>:        moveal fp@(16),a0
0x2338 <max+24>:        jsr a0@
0x233a <max+26>:        tstl d0
```

```
0x233c <max+28>:        bge 0x2342 <max+34>
0x233e <max+30>:        movel d2,d0
0x2340 <max+32>:        bra 0x2344 <max+36>
0x2342 <max+34>:        movel d3,d0
0x2344 <max+36>:        moveml fp@(-8),d2-d3
0x234a <max+42>:        unlk fp
0x234c <max+44>:        rts
```

First, the correctness of the functional parameter is formalized by the following constraint definition 'comp-correctness'. There are three new "undefined" functions comp-statep $(s,\ a,\ b)$, comp-t $(a,\ b)$, and comp $(a,\ b)$ introduced into the logic by 'comp-correctness', each of which has its intended meaning as the precondition on the initial state, the time function, and the behavior function, respectively. The correctness statement is the standard one we have been using throughout this work.

CONSERVATIVE AXIOM: p-disjointness
(p-disjoint $(x,\ n,\ s) \wedge ((j + \text{index-n}\ (y,\ x)) \leq n)$)
$\Rightarrow$ p-disjoint $(y,\ j,\ s)$

Simultaneously, we introduce the new function symbol 'p-disjoint'.

CONSERVATIVE AXIOM: comp-correctness
comp-statep $(s,\ a,\ b)$
$\Rightarrow$ **let** sn **be** stepn $(s,$ comp-t $(a,\ b))$
in
(mc-status (sn) = 'running)
$\wedge$ (mc-pc (sn) = rts-addr (s))
$\wedge$ (read-rn (32, 14, mc-rfile (sn))
= read-rn (32, 14, mc-rfile (s)))
$\wedge$ (read-rn (32, 15, mc-rfile (sn))
= add (32, read-sp (s), 4))
$\wedge$ (((*oplen* $\leq$ 32) $\wedge$ d2-7a2-5p (rn))
$\Rightarrow$ (read-rn $(oplen,\ rn,$ mc-rfile (sn))
= read-rn $(oplen,\ rn,$ mc-rfile (s)))))
$\wedge$ (p-disjoint $(x,\ k,\ s)$
$\Rightarrow$ (read-mem $(x,$ mc-mem $(sn),\ k)$
= read-mem $(x,$ mc-mem $(s),\ k)$)))
$\wedge$ (iread-dn (32, 0, sn) = comp $(a,\ b)$) **endlet**

Simultaneously, we introduce the new function symbols 'comp-statep', 'comp-t', and 'comp'.

Assuming the correctness of its functional parameter, we can prove the correctness of the binary of `max`. As shown below, max-comp $(a,\ b)$ is the behavior function; max-t $(a,\ b)$ is the time function, max-statep $(s,\ a,\ b)$ is the precondition on the initial state; finally, 'max-correctness' gives the correctness of this program.

```
DEFINITION:
max-comp(a, b)
= if negativep(comp(a, b)) then b
  else a endif

DEFINITION: max-t0(a, b) = 8

DEFINITION:
max-t(a, b)
= (max-t0(a, b)
   + (comp-t(a, b)
      + if negativep(comp(a, b)) then 7
        else 6 endif))

DEFINITION:
MAX-CODE
= '(78 86 0 0 72 231 48 0 38 46 0 8 36 46 0 12 47 2
    47 3 32 110 0 16 78 144 74 128 108 4 32 2 96 2 32
    3 76 238 0 12 255 248 78 94 78 117)

DEFINITION:
max-sp(s, a, b)
= ((mc-status(s) = 'running)
   ∧ evenp(mc-pc(s))
   ∧ rom-addrp(mc-pc(s), mc-mem(s), 46)
   ∧ mcode-addrp(mc-pc(s), mc-mem(s), MAX-CODE)
   ∧ (a = iread-mem(add(32, read-sp(s), 4), mc-mem(s), 4))
   ∧ (b = iread-mem(add(32, read-sp(s), 8), mc-mem(s), 4))
   ∧ ram-addrp(sub(32, 24, read-sp(s)), mc-mem(s), 40))

CONSERVATIVE AXIOM: max-state
(max-statep(s, a, b) ⇒ comp-statep(stepn(s, max-t0(a, b)), a, b))
∧ (max-statep(s, a, b)
   ⇒ p-disjoint(add(32,
                    read-rn(32, 15, mc-rfile(s)),
                    4294967272),
                40,
                stepn(s, max-t0(a, b))))
∧ (max-statep(s, a, b) = (max-sp(s, a, b) ∧ comp-loadp(s, a, b)))
```

Simultaneously, we introduce the new function symbols 'max-statep' and 'comp-loadp'.

```
THEOREM: max-correctness
let sn be stepn(s, max-t(a, b))
in
max-statep(s, a, b)
⇒ ((mc-status(sn) = 'running)
   ∧ (mc-pc(sn) = rts-addr(s))
   ∧ (read-rn(32, 14, mc-rfile(sn))
      = read-rn(32, 14, mc-rfile(s)))
   ∧ (read-rn(32, 15, mc-rfile(sn))
```

$$
\begin{array}{ll}
 & = \ \text{add}\,(32,\ \text{read-sp}\,(s),\ 4)) \\
\wedge & (((\mathit{oplen} \leq 32) \wedge \text{d2-7a2-5p}\,(\mathit{rn})) \\
 & \Rightarrow \ (\text{read-rn}\,(\mathit{oplen},\ \mathit{rn},\ \text{mc-rfile}\,(\mathit{sn})) \\
 & \qquad = \ \text{read-rn}\,(\mathit{oplen},\ \mathit{rn},\ \text{mc-rfile}\,(s)))) \\
\wedge & ((\text{disjoint}\,(x,\ k,\ \text{sub}\,(32,\ 24,\ \text{read-sp}\,(s)),\ 40) \\
 & \quad \wedge \ \text{max-disjoint}\,(x,\ k,\ s)) \\
 & \Rightarrow \ (\text{read-mem}\,(x,\ \text{mc-mem}\,(\mathit{sn}),\ k) \\
 & \qquad = \ \text{read-mem}\,(x,\ \text{mc-mem}\,(s),\ k))) \\
\wedge & (\text{iread-dn}\,(32,\ 0,\ \mathit{sn}) = \text{max-comp}\,(a,\ b))) \ \textbf{endlet}
\end{array}
$$

The most interesting feature of the above theorem 'max-correctness' is that it can be used to prove the correctness of multiple functional instances of MAX. To see how this works, let us try to prove the correctness of the binary of `max(a, b, gt)` by an instantiation of the above theorem. The C function `gt` is given below.

```
gt(int a, int b)
{
  if (a == b)
    return 0;
  else if (a > b)
    return 1;
  else return -1;
}
```

```
The MC68020 assembly code of the above GT program.  The code is generated
by "gcc -O".

0x22de <gt>:    linkw fp,#0
0x22e2 <gt+4>:  movel fp@(8),d1
0x22e6 <gt+8>:  movel fp@(12),d0
0x22ea <gt+12>: cmpl d1,d0
0x22ec <gt+14>: bne 0x22f2 <gt+20>
0x22ee <gt+16>: clrl d0
0x22f0 <gt+18>: bra 0x22fc <gt+30>
0x22f2 <gt+20>: cmpl d1,d0
0x22f4 <gt+22>: bge 0x22fa <gt+28>
0x22f6 <gt+24>: movel #1,d0
0x22f8 <gt+26>: bra 0x22fc <gt+30>
0x22fa <gt+28>: movel #-1,d0
0x22fc <gt+30>: unlk fp
0x22fe <gt+32>: rts
```

There are two steps in the proof. The first step is to establish the correctness of the machine code for `gt`, since we must discharge the constrains introduced by 'comp-correctness' when any instantiation of that theorem with the substitution

$$\{\text{gt-statep/comp-statep}, \text{gt-t/comp-t}, \text{gt/comp}\}$$

is performed. The formalization and correctness theorem of the binary of `gt` is given as follows.

```
DEFINITION:
gt(a, b)
= if a = b then 0
  elseif ilessp(b, a) then 1
  else -1 endif

DEFINITION:
gt-t(a, b)
= if a = b then 9
  elseif ilessp(b, a) then 11
  else 10 endif

DEFINITION:
GT-CODE
= '(78 86 0 0 34 46 0 8 32 46 0 12 176 129 102 4 66
    128 96 10 176 129 108 4 112 1 96 2 112 255 78 94
    78 117)

DEFINITION:
gt-statep(s, a, b)
= ((mc-status(s) = 'running)
   ∧ evenp(mc-pc(s))
   ∧ rom-addrp(mc-pc(s), mc-mem(s), 34)
   ∧ mcode-addrp(mc-pc(s), mc-mem(s), GT-CODE)
   ∧ ram-addrp(sub(32, 4, read-sp(s)), mc-mem(s), 16)
   ∧ (a = iread-mem(add(32, read-sp(s), 4), mc-mem(s), 4))
   ∧ (b = iread-mem(add(32, read-sp(s), 8), mc-mem(s), 4)))

THEOREM: gt-correctness
let sn be stepn(s, gt-t(a, b))
in
gt-statep(s, a, b)
⇒ ((mc-status(sn) = 'running)
   ∧ (mc-pc(sn) = rts-addr(s))
   ∧ (read-rn(32, 14, mc-rfile(sn))
      = read-rn(32, 14, mc-rfile(s)))
   ∧ (read-rn(32, 15, mc-rfile(sn))
      = add(32, read-sp(s), 4))
   ∧ (d2-7a2-5p(rn)
      ⇒ (read-rn(oplen, rn, mc-rfile(sn))
         = read-rn(oplen, rn, mc-rfile(s))))
   ∧ (disjoint(x, k, sub(32, 4, read-sp(s)), 4)
      ⇒ (read-mem(x, mc-mem(sn), k)
         = read-mem(x, mc-mem(s), k)))
   ∧ (iread-dn(32, 0, sn) = gt(a, b))) endlet
```

We then, in the second step, prove the correctness of the binary of *max*(a, b, gt) by instantiating the theorem 'max-correctness'. The functions max-gt-statep(s, a, b), max-gt-t(a, b), max-gt(a, b) formalize the precondition, the time function, and the functional behavior of this program, respectively. Finally, the theorem 'max-gt-correctness' shows the correctness of the program.

```
DEFINITION:
max-gt-statep(s, a, b)
= let comp be read-mem(add(32, read-sp(s), 12), mc-mem(s), 4)
  in
  (mc-status(s) = 'running)
  ∧ evenp(mc-pc(s))
  ∧ rom-addrp(mc-pc(s), mc-mem(s), 46)
  ∧ mcode-addrp(mc-pc(s), mc-mem(s), MAX-CODE)
  ∧ (a = iread-mem(add(32, read-sp(s), 4), mc-mem(s), 4))
  ∧ (b = iread-mem(add(32, read-sp(s), 8), mc-mem(s), 4))
  ∧ evenp(comp)
  ∧ rom-addrp(comp, mc-mem(s), len(GT-CODE))
  ∧ mcode-addrp(comp, mc-mem(s), GT-CODE)
  ∧ ram-addrp(sub(32, 28, read-sp(s)), mc-mem(s), 44) endlet

DEFINITION:
max-gt-t(a, b)
= (max-t0(a, b)
   + (gt-t(a, b)
      + if negativep(gt(a, b)) then 7
        else 6 endif))

DEFINITION:
max-gt(a, b)
= if negativep(gt(a, b)) then b
  else a endif

THEOREM: max-gt-correctness
 max-gt-statep(s, a, b)
⇒ let sn be stepn(s, max-gt-t(a, b))
  in
  (mc-status(sn) = 'running)
  ∧ (mc-pc(sn) = rts-addr(s))
  ∧ (read-rn(32, 14, mc-rfile(sn))
      = read-rn(32, 14, mc-rfile(s)))
  ∧ (read-rn(32, 15, mc-rfile(sn))
      = add(32, read-sp(s), 4))
  ∧ (((oplen ≤ 32) ∧ d2-7a2-5p(rn))
      ⇒ (read-rn(oplen, rn, mc-rfile(sn))
          = read-rn(oplen, rn, mc-rfile(s))))
  ∧ ((disjoint(x, k, sub(32, 24, read-sp(s)), 40)
       ∧ disjoint(x, k, sub(32, 28, read-sp(s)), 4))
      ⇒ (read-mem(x, mc-mem(sn), k)
          = read-mem(x, mc-mem(s), k)))
  ∧ (iread-dn(32, 0, sn) = max-gt(a, b)) endlet
```

The above theorem 'max-gt-correctness' is simply an intantiation of the theorem 'max-correctness' by substituting 'max-gt-statep' for 'max-statep', max-gt-t for max-t, max-t for max-comp, and etc. We recommend the interested reader study the complete proof script in Appendix C.

6.3 Switch Statement

The switch statement has posed no problems in high-level language semantics, as it can be simply treated as a bunch of nested if statements. But, at the machine-code level, the matter gets a bit complicated since it may involve a tranfer of control to a computed location. We now examine how to deal with the optimized binary of C's switch statement, produced by the Gnu C compiler. In this relatively simple setting, our limited experience indicates that there are no major obstacles in dealing with computed jumps in our approach to machine code program proving. But we suspect this would pose some very serious problems for any low-level code verification work that abstracts away programs from the memory. At the present, we have not considered some perhaps much more difficult transfer issues, such as the set-jump/long-jump pair in the standard C library.

We have provided some program proving support for the computed transfer construct in our lemma library. Since the Gnu C compiler utilizes a very standard technique to handle the switch statement, we believe our treatment is probably applicable to many other languages and compilers using the same technique.

To make our discussion concrete, we here present a very simple example to demonstrate the problem we have dealt with. Reading the assembly code of the following C function `foo`, the instruction at line `foo+14` computes the address of an entry in a table embedded in the program that stores the offset for jumping, and the instruction at line `foo+18` jumps according to the offset.

```
int foo(int n)
{
  int i;

  switch(n) {
  case 0: i = 0; break;
  case 1: i = 1; break;
  case 2: i = 4; break;
  case 3: i = 9; break;
  case 4: i = 16; break;
  default: i = n; break;
  };
  return i;
}
```

The MC68020 assembly code generated by the Gnu C compiler with optimization.

```
0x23b2 <foo>:      linkw a6,#0
0x23b6 <foo+4>:    movel a6@(8),d0
0x23ba <foo+8>:    movel #4,d1
```

```
0x23bc <foo+10>:   cmpl d1,d0
0x23be <foo+12>:   bhi 0x23e4 <foo+50>
0x23c0 <foo+14>:   movew 0x23c8[d0.l*2],d1
0x23c4 <foo+18>:   jmp 0x23c8[d1.w]
0x23c8 <foo+22>:   orb #14,a2
0x23cc <foo+26>:   orb #22,a2@
0x23d0 <foo+30>:   orb #-128,a2@+
0x23d4 <foo+34>:   bra 0x23e4 <foo+50>
0x23d6 <foo+36>:   movel #1,d0
0x23d8 <foo+38>:   bra 0x23e4 <foo+50>
0x23da <foo+40>:   movel #4,d0
0x23dc <foo+42>:   bra 0x23e4 <foo+50>
0x23de <foo+44>:   movel #9,d0
0x23e0 <foo+46>:   bra 0x23e4 <foo+50>
0x23e2 <foo+48>:   movel #16,d0
0x23e4 <foo+50>:   unlk a6
0x23e6 <foo+52>:   rts
```

The correctness proof of this toy program is trivial, and completely automatic with the help of the special purpose lemmas we have added into the lemma library. The formalization is no different from the other examples: FOO-CODE is the machine code of the program `foo`; foo-statep $(s,\ n)$ formalizes the preconditions on the initial state; foo-t (n) defines the exact number of instructions to complete this program; and foo (n) characterizes the functional behavior of this program. Finally, the theorem 'foo-correctness' asserts the correctness of this program.

```
DEFINITION:
FOO-CODE
=  '(78 86 0 0 32 46 0 8 114 4 176 129 98 36 50 59 10
     6 78 251 16 2 0 10 0 14 0 18 0 22 0 26 66 128 96
     14 112 1 96 10 112 4 96 6 112 9 96 2 112 16 78 94
     78 117)
```

```
DEFINITION:
foo-statep (s, n)
=  ((mc-status (s) = 'running)
    ∧ evenp (mc-pc (s))
    ∧ rom-addrp (mc-pc (s), mc-mem (s), 54)
    ∧ mcode-addrp (mc-pc (s), mc-mem (s), FOO-CODE)
    ∧ ram-addrp (sub (32, 4, read-sp (s)), mc-mem (s), 12)
    ∧ disjoint (mc-pc (s), 54, sub (32, 4, read-sp (s)), 12)
    ∧ (n = iread-mem (add (32, read-sp (s), 4), mc-mem (s), 4)))
```

```
DEFINITION:
foo-t (n)
=  if (n = 0) ∨ (n = 1) ∨ (n = 2) ∨ (n = 3) then 11
   elseif n = 4 then 10
   else 7 endif
```

```
DEFINITION:
foo (n)
= if between-ileq (0, n, 4) then n * n
  else n endif

THEOREM: foo-correctness
 let sn be stepn (s, foo-t (n))
 in
 foo-statep (s, n)
 ⇒ ((mc-status (sn) = 'running)
    ∧ (mc-pc (sn) = rts-addr (s))
    ∧ (read-rn (32, 14, mc-rfile (sn))
       = read-rn (32, 14, mc-rfile (s)))
    ∧ (read-rn (32, 15, mc-rfile (sn))
       = add (32, read-an (32, 7, s), 4))
    ∧ (d2-7a2-5p (rn)
       ⇒ (read-rn (oplen, rn, mc-rfile (sn))
          = read-rn (oplen, rn, mc-rfile (s))))
    ∧ (disjoint (x, k, sub (32, 4, read-sp (s)), 12)
       ⇒ (read-mem (x, mc-mem (sn), k)
          = read-mem (x, mc-mem (s), k)))
    ∧ (iread-dn (32, 0, sn) = foo (n))) endlet
```

6.4 Embedded Assembly Code

We all know that no high-level programming language semantics can make clear the meaning of embedded assembly code in programs, simply because assembly code is intrinsically machine dependent. By considering directly the binary code of high-level programs after compilation, we do not need to address this semantics issue. Programs and embedded assembly codes are all translated into the formalized world of machine instructions, and their correctness can be studied on the basis of a formal processor semantics. To make our discussion concrete, let us study a very simple example in this section. Our example also demonstrates how easily we can handle embedded assembly code. All we need to know is what the programmer should know when he writes the embedded assembly code.

Our example is the following trivial C function `foo` which returns `a` if the longword at location 10000 is equal to 0 and returns `b` otherwise.

```
int foo (int a, int b)
{
  asm("tstl 10000:w ");
  asm("beq l1 ");
  asm("movl a6@(12), d0 ");
  asm("jmp end ");
  asm("l1: movl a6@(8), d0 ");
```

```
  asm("end: nop ");
}
```

```
The MC68020 assembly code generated by the Gnu C compiler with optimization.
```

```
0x243a <foo>:           linkw fp,#0
0x243e <foo+4>:         tstl @#0x2710
0x2442 <foo+8>:         beq 0x244e <foo+20>
0x2446 <foo+12>:        movel fp@(12),d0
0x244a <foo+16>:        jmp 0x2452 <foo+24>
0x244e <foo+20>:        movel fp@(8),d0
0x2452 <foo+24>:        nop
0x2454 <foo+26>:        unlk fp
0x2456 <foo+28>:        rts
```

As always, we formalize the preconditions of the initial state, the time function, and the functional behavior of the program, which are given below as the functions 'foo-statep', 'foo-t', and 'foo', respectively.

DEFINITION:
FOO-CODE
= '(78 86 0 0 74 184 39 16 103 0 0 10 32 46 0 12 78
250 0 6 32 46 0 8 78 113 78 94 78 117)

DEFINITION:
foo-statep(s, a, b)
= ((mc-status(s) = 'running)
∧ evenp(mc-pc(s))
∧ rom-addrp(mc-pc(s), mc-mem(s), 30)
∧ mcode-addrp(mc-pc(s), mc-mem(s), FOO-CODE)
∧ ram-addrp(sub(32, 4, read-sp(s)), mc-mem(s), 16)
∧ ram-addrp(10000, mc-mem(s), 4)
∧ disjoint(10000, 4, sub(32, 4, read-sp(s)), 16)
∧ (a = iread-mem(add(32, read-sp(s), 4), mc-mem(s), 4))
∧ (b = iread-mem(add(32, read-sp(s), 8), mc-mem(s), 4)))

DEFINITION:
foo-t(x)
= **if** x = 0 **then** 7
else 8 **endif**

DEFINITION:
foo(a, b, x)
= **if** x = 0 **then** a
else b **endif**

Note that we need to specify in 'foo-statep' that the memory locations 10000 to 10003 are readable and do not overlap with a certain part of the stack that will be modified by the program.

The correctness theorem of this program, given below, strictly follows our formulation in Chapter 5. The proof of this theorem is quite straightforward.

```
THEOREM: foo-correctness
let x be iread-mem(10000, mc-mem(s), 4)
in
foo-statep(s, a, b)
⇒ ((mc-status(stepn(s, foo-t(x))) = 'running)
    ∧ (mc-pc(stepn(s, foo-t(x))) = rts-addr(s))
    ∧ (read-rn(32, 14, mc-rfile(stepn(s, foo-t(x))))
        = read-rn(32, 14, mc-rfile(s)))
    ∧ (read-rn(32, 15, mc-rfile(stepn(s, foo-t(x))))
        = add(32, read-an(32, 7, s), 4))
    ∧ (d2-7a2-5p(rn)
        ⇒ (read-rn(oplen,
                   rn,
                   mc-rfile(stepn(s, foo-t(x))))
            = read-rn(oplen, rn, mc-rfile(s))))
    ∧ (disjoint(x, k, sub(32, 4, read-sp(s)), 16)
        ⇒ (read-mem(x, mc-mem(stepn(s, foo-t(x))), k)
            = read-mem(x, mc-mem(s), k)))
    ∧ (iread-dn(32, 0, stepn(s, foo-t(x))) = foo(a, b, x))) endlet
```

The last conjunct in the above theorem proves, that after executing foo-t(x) instructions, the content of data register D0 is equal to foo(a, b, x), where a and b are the two inputs, and x is the longword at location 10000 in the memory.

Chapter 7

Proving Theorems about the Berkeley Unix C String Library

One of our main goals in defining a formal model for a widely used processor was to study the correctness of *real* programs executed on that particular processor. The results reported in the preceding chapters have demonstrated the *potential* to apply our verification methodology to some small programs that are in *real* use. As the final stage of this project, we chose to investigate some small, but real applications of our verification system.

After studying carefully several possible application candidates, we decided to study the Berkeley Unix C String Library – an implementation of the C string library of ANSI/ISO standard. The reasons for this choice were very simple: the library has been widely used and publicly released as part of the Berkeley Unix Operating System; and the string functions are quite simple and self-contained, and hence a good target for experimentation. We are quite pleased by the results of this small verification project. Twenty one out of twenty-two functions specified in the ISO standard have been mechanically verified. The function `strerror`, though mathematically trivial, is the only one left out because of the need of formalizing IO, to which we have not attended in this thesis. There were three programming errors revealed in the process of the verification. The machine code for these string functions was generated by the Gnu C compiler.

This chapter reports our work on proving mathematical theorems about the Berkeley Unix C String Library. We first give a very brief and informal introduction to the functions in the Berkeley Unix C String Library that we have considered and the mathematical theorems about these functions that we have proved. This should give the reader an overview of this small verification project. To formalize our discussion, we next look into the formal verification of the Berkeley C String Library. We only present the mechanical proofs of a couple of the most interesting and tricky functions in the library: `memmove` and `strstr`. Finally, we discuss the two programming errors we have discovered in studying this C string library. The complete proof script of all the string functions is given in Appendix C.

Copyright Notice. All the C code, taken from the Berkeley C String library, presented in this chapter and Appendix C, is subject to the following copyright terms.


```
/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms are permitted provided
 * that: (1) source distributions retain this entire copyright notice and
 * comment, and (2) distributions including binaries display the following
 * acknowledgement:  ``This product includes software developed by the
 * University of California, Berkeley and its contributors'' in the
 * documentation or other materials provided with the distribution and in
 * all advertising materials mentioning features or use of this software.
 * Neither the name of the University nor the names of its contributors may
 * be used to endorse or promote products derived from this software without
 * specific prior written permission.
 * THIS SOFTWARE IS PROVIDED ``AS IS'' AND WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF
 * MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 */
```


7.1 The Berkeley Unix C String Library

The Berkeley Unix C string library is intended to be an implementation of the C string library of the ANSI/ISO standard, and is publicly released as part of the Berkeley Unix Operating System.[1] There are twenty-two string functions specified in the ANSI/ISO standard, and we have verified the binary of the Berkeley implementation of twenty one of them. The binary was generated by the Gnu C Compiler for the MC68020. In this section, we give an informal description of this small verification project. For each of the string functions verified, we provide the formal syntax of the function, a paraphrase of the informal English specification of the ISO standard [28, 44], and an informal description of the theorems we proved about the function.

Notation. We adopt an informal, conventional notation to describe the theorems we have proved about these C string functions. We use $s, s1$, and $s2$ to denote strings, $s[i]$ to denote the ith character in the string s, and x' to denote the value of x in the

[1] The copy of the Berkeley C string library used in this work was obtained by anonymous ftp from `ftp.uu.net`

post state. We also informally introduce an predicate $disjoint(s1, s2)$ to assert that the strings $s1$ and $s2$ do not overlap.

Our presentation below of the C string library is highly informal but follows closely the ISO standard [28], where the reader may find a more accurate and verbose English description of these functions. Still more formal is the treatment in the Appendix C.

7.1.1 The `memcpy` Function

Synopsis. `void *memcpy (void *s1, const void *s2, size_t n)`

Description. The `memcpy` function copies `n` characters from the object `s2` into the object `s1`, and returns the value of `s1`. The behavior of the function is undefined if `s1` and `s2` overlap.

Theorem. We have: $i < n \Rightarrow s1'[i] = s2[i]$.[2]

7.1.2 The `memmove` Function

Synopsis. `void *memmove (void *s1, const void *s2, size_t n)`

Description. The `memmove` function copies `n` characters from the object `s2` into the object `s1`, and returns the value of `s1`. The `memmove` function works correctly on any two objects.

Theorem. We have: $i < n \Rightarrow s1'[i] = s2[i]$.

7.1.3 The `strcpy` Function

Synopsis. `char *strcpy (char *s1, const char *s2)`

Description. The `strcpy` function copies the string `s2` into the array `s1`, and returns the value of `s1`. The behavior of the function is undefined if the strings `s1` and `s2` overlap.

Theorem. Assuming $disjoint(s1, s2)$, we have:

$$j \leq |s2| \Rightarrow s1'[j] = s2[j].$$

[2] The Berkeley implementation of `memcpy` works correctly on any two objects.

7.1.4 The `strncpy` Function

Synopsis. `char *strncpy (char *s1, const char *s2, size_t n)`

Description. The `strncpy` function copies at most `n` characters from the array `s2` to the array `s1`, and returns the value of `s1`. The behavior of the function is undefined if the strings `s1` and `s2` overlap.

Theorem. Assuming $disjoint(s1, s2)$, we have:

1. $j < min(n, |s2|) \Rightarrow s1'[j] = s2[j]$.
2. $|s2| \leq j < n \Rightarrow s1'[j] = 0$.

7.1.5 The `strcat` Function

Synopsis. `char *strcat (char *s1, const char *s2)`

Description. The `strcat` function appends a copy of the string `s2` to the end of the string `s1`, and returns the value of `s1`. The behavior of the function is undefined if `s1` and `s2` overlap.

Theorem. Assume $disjoint(s1, s2)$, we have:

1. $j < |s1| \Rightarrow s1'[j] = s1[j]$.
2. $|s1| \leq j < |s1| + |s2| \Rightarrow (s1'[j] = s2[j - |s1|]$.

7.1.6 The `strncat` Function

Synopsis. `char *strncat (char *s1, const char *s2, size_t n)`

Description. The `strncat` function appends at most `n` characters from the array `s2` to the end of the string `s1`, and returns the value of `s1`. The behavior of the function is undefined if `s1` and `s2` overlap.

Theorem. Assuming $disjoint(s1, s2)$, we have:

1. $j < |s1| \Rightarrow s1'[j] = s1[j]$.
2. $|s1| \leq j < |s1| + min(|s2|, n) \Rightarrow s1'[j] = s2[j - |s1|]$.

7.1.7 The `memcmp` Function

Synopsis. `int memcmp (const void *s1, const void *s2, size_t n)`

Description. The `memcmp` function compares the first `n` characters of the objects `s1` and `s2`, and returns an integer greater than, equal to, or less than zero, according to the lexical order of the objects `s1` and `s2`.

Theorem. We have:

1. $memcmp(s1, s2, n) = 0 \Rightarrow \forall j < n(s1[j] = s2[j])$.
2. $memcmp(s1, s2, n) \neq 0 \Rightarrow \exists i < n(memcmp(s1, s2, n) = s1[i] - s2[i] \wedge \forall j < i(s1[j] = s2[j]))$
3. $memcmp(s2, s1, n) < 0 \leftrightarrow memcmp(s1, s2, n) > 0$

7.1.8 The `strcmp` Function

Synopsis. `int strcmp (const char *s1, const char *s2)`

Description. The `strcmp` function compares the string `s1` to the string `s2`, and returns an integer greater than, equal to, or less than zero, according to the lexical order of the strings `s1` and `s2`.

Theorem. We have:

1. $strcmp(s1, s2) = 0 \Rightarrow \forall j \leq |s1|(s1[j] = s2[j])$.
2. $strcmp(s1, s2) \neq 0 \Rightarrow \exists i < |s1|(strcmp(s1, s2) = s1[i] - s2[i] \wedge \forall j < i(s1[j] = s2[j]))$
3. $strcmp(s2, s1) < 0 \leftrightarrow strcmp(s1, s2) > 0$

7.1.9 The `strcoll` Function

Synopsis. `int strcoll (const char *s1, const char *s2)`

Description. Since `LC_COLLATE` is not implemented, the function `strcoll` is equivalent to `strcmp`.

Theorem. We have: $strcoll(s1, s2) = strcmp(s1, s2)$.

7.1.10 The `strncmp` Function

Synopsis. `int strncmp (const char *s1, const char *s2, size_t n)`

Description. The `strncmp` function compares at most `n` characters of the arrays `s1` and `s2`, and returns an integer greater than, equal to, or less than zero, according to the lexical order of the arrays `s1` and `s2`.

Theorem. We have:

1. $strncmp(s1, s2, n) = 0 \Rightarrow \forall j < min(|s1|, n)(s1[j] = s2[j])$.
2. $strncmp(s1, s2, n) \neq 0 \Rightarrow \exists i < min(|s1|, n)(strncmp(s1, s2, n) = s1[i]-$

$s2[i] \wedge \forall j < i(s1[j] = s2[j]))$

3. $strncmp(s2, s1, n) < 0 \leftrightarrow strncmp(s1, s2, n) > 0$

7.1.11 The `strxfrm` Function

Synopsis. `size_t strxfrm (char *s1, const char *s2, size_t n)`

Description. Since `LC_COLLATE` is not implemented, the `strxfrm` function simply copies the string `s2` to the array `s1`, and returns the length of the string `s2`. At most `n` characters are copied to the array `s1`. If `n` is zero, `s1` is permitted to be a null pointer.

Theorem. Assuming $disjoint(s1, s2)$, we have:

1. $j < min(n, |s2|) \Rightarrow s1'[j] = s2[j]$.
2. $strxfrm(s1, s2, n) = |s2|$. [3]

7.1.12 The `memchr` Function

Synopsis. `void *memchr (const void *s, int c, size_t n)`

Description. The `memchr` function returns a pointer to the first occurrence of `c` in the initial `n` characters of the object `s`, or a null pointer if `c` is not found.

Theorem. We have:

1. $memchr(s, c, n) \neq 0 \Rightarrow s[memchr(s, c, n) - s] = c$.
2. $memchr(s, c, n) = 0 \Rightarrow \forall j < n(s[j] \neq c)$.
3. $j < (memchr(s, c, n) - s) \Rightarrow s[j] \neq c$.

7.1.13 The `strchr` Function

Synopsis. `char *strchr (const char *s, int c)`

Description. The `strchr` function returns a pointer to the first occurrence of `c` in the string `s`, or a null pointer if `c` is not found.

Theorem. We have:

1. $strchr(s, c) \neq 0 \Rightarrow s[strchr(s, c) - s] = c$.
2. $strchr(s, c) = 0 \Rightarrow \forall j < |s|, s[j] \neq c$.
3. $j < (strchr(s, c) - s) \Rightarrow s[j] \neq c$.

[3] The Berkeley `strxfrm` function contains a bug that falsifies this theorem.

7.1.14 The `strcspn` Function

Synopsis. `size_t strcspn (const char *s1, const char *s2)`

Description. The `strcspn` function returns the length of the maximum initial segment of the string `s1` which consists entirely of characters not from the string `s2`.

Theorem. We have:

1. $strchr(s2, s1[strcspn(s1, s2)]) \neq 0$.
2. $j < strcspn(s1, s2) \Rightarrow strchr(s2, s1[j]) = 0$

7.1.15 The `strpbrk` Function

Synopsis. `char *strpbrk (const char *s1, const char *s2)`

Description. The `strpbrk` function returns a pointer to the first occurrence in the string `s1` of any character from the string `s2`, or a null pointer if no character from `s2` occurs in `s1`.

Theorem. We have:

1. $strpbrk(s1, s2) \neq 0 \Rightarrow strchr1(s2, s1[strpbrk(s1, s2) - s1]) \neq 0$.
2. $j < (strpbrk(s1, s2) - s1) \Rightarrow strchr1(s2, s1[j]) = 0$.

7.1.16 The `strrchr` Function

Synopsis. `char *strrchr (const char *s, int c)`

Description. The `strrchr` function returns a pointer to the last occurrence of `c` in the string `s`, or a null pointer if `c` is not found.

Theorem. We have:

1. $strrchr(s, c) \neq 0 \Rightarrow s[strrchr(s, c) - s] = c$.
2. $strrchr(s, c) = 0 \Rightarrow \forall j < |s|(s[j] \neq c)$.
3. $(strrchr(s, c) - s) < j < |s| \Rightarrow s[j] \neq c$.

7.1.17 The `strspn` Function

Synopsis. `size_t strspn (const char *s1, const char *s2)`

Description. The `strspn` function returns the length of the maximum initial segment of the string `s1` which consists entirely of characters from the string `s2`.

Theorem. We have:

1. $strspn(s1, s2) < |s1| \Rightarrow strchr1(s2, s1[strspn(s1, s2)]) = 0$.
2. $j < strspn(s1, s2) \Rightarrow strchr1(s2, s1[j]) \neq 0$.

7.1.18 The `strstr` Function

Synopsis. `char *strstr (const char *s1, const char *s2)`

Description. The `strstr` function returns a pointer to the first occurrence in the string `s1` of the string `s2`, or a null pointer if the string `s2` is not found.

Theorem. We have:

1. $strstr(s1, s2) \neq 0 \Rightarrow strncmp(strstr(s1, s2), s2, |s2|) = 0$.
2. $strstr(s1, s2) = 0 \Rightarrow \forall j < |s1|(strncmp(s1 + j, s2, |s2|) \neq 0)$.
3. $s1 \leq s < strstr(s1, s2) \Rightarrow strncmp(s, s2, |s2|) \neq 0$.

7.1.19 The `strtok` Function

Synopsis. `char *strtok (char *str1, const char *str2)`

Description. A sequence of calls to the `strtok` function breaks the string `s1` into a sequence of tokens, each of which is delimited by a character from the separator string `s2`. The `strtok` function returns a pointer to the first character of the current token, or a null pointer if there is no token found in the token string. Please see [53, 28] for more detailed descriptions.

Theorem. Let $i(s1)$ be $strspn(s1, s2)$, $j(s1)$ be $strcspn(s1 + i(s1), s2)$, and $last$ be the static variable, we have:

1. $((s1 \neq 0) \wedge (s1[i(s1)] = 0)) \Rightarrow (strtok(s1, s2) = 0) \wedge (last' = 0))$
2. $((s1 \neq 0) \wedge (s1[i(s1)] \neq 0) \wedge (s1[j(s1)] = 0)) \Rightarrow ((strtok(s1, s2) = s1 + i(s1)) \wedge (last' = 0))$
3. $((s1 \neq 0) \wedge (s1[i(s1)] \neq 0) \wedge (s1[j(s1)] \neq 0)) \Rightarrow ((strtok(s1, s2) = s1 + i(s1)) \wedge (last' = s1 + j(s1) + 1) \wedge (s1'[j(s1)] = 0))$
4. $(last = 0) \Rightarrow ((strtok(0, s2) = 0) \wedge (last' = 0))$
5. $((last \neq 0) \wedge (last[i(last)] \neq 0) \wedge (last[j(last)] = 0)) \Rightarrow ((strtok(0, s2) = last + i(last)) \wedge (last' = 0))$
6. $((last \neq 0) \wedge (last[i(last)] \neq 0) \wedge (last[j(last)] \neq 0)) \Rightarrow ((strtok(0, s2) = last + i(last)) \wedge (last' = last + j(last) + 1) \wedge (last'[j(last)] = 0))$

7.1.20 The `memset` Function

Synopsis. `void *memset (void *s, const int c, size_t n)`

Description. The `memset` function copies the value of `c` into each of the first `n` characters of the object `s`.

Theorem. We have:

1. $i \leq j < n \Rightarrow s'[j] = ch$.
2. $n \leq j < |s| \Rightarrow s'[j] = s[j]$.

7.1.21 The `strlen` Function

Synopsis. `size_t strlen (const char *s)`

Description. The `strlen` function returns the length of the string `s`.

Theorem. We have:

1. $j < strlen(s) \Rightarrow s'[j] \neq 0$.
2. $s'[strlen(s)] = 0$.

7.2 Proving the String Functions Correct

The descriptions given in the preceding section are very informative, but rather informal. To remedy that, we describe in this section the formalization and verification of two functions `memmove` and `strstr` of the Berkeley C string library.

7.2.1 Proving the `memmove` Function Correct

The first example is the `memmove` function. As one of the copying functions, the interesting feature of this function is that it works even when the copying takes place between objects that overlap.

As always, we first give the C and the assembly code of this function to be studied.

```
/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 */

typedef int word;               /* "word" used for optimal copy speed */

#define wsize   sizeof(word)
```

```
#define wmask   (wsize - 1)

/*
 * Copy a block of memory, handling overlap.
 * This is the routine that actually implements
 * (the portable versions of) bcopy, memcpy, and memmove.
 */
void *
memmove(dst0, src0, length)
        void *dst0;
        const void *src0;
        register size_t length;
{
        register char *dst = dst0;
        register const char *src = src0;
        register size_t t;

        if (length == 0 || dst == src)          /* nothing to do */
                goto done;

        /*
         * Macros: loop-t-times; and loop-t-times, t>0
         */
#define TLOOP(s) if (t) TLOOP1(s)
#define TLOOP1(s) do { s; } while (--t)

        if ((unsigned long)dst < (unsigned long)src) {
                /*
                 * Copy forward.
                 */
                t = (int)src;   /* only need low bits */
                if ((t | (int)dst) & wmask) {
                        /*
                         * Try to align operands.  This cannot be done
                         * unless the low bits match.
                         */
                        if ((t ^ (int)dst) & wmask || length < wsize)
                                t = length;
                        else
                                t = wsize - (t & wmask);
                        length -= t;
                        TLOOP1(*dst++ = *src++);
                }
                /*
                 * Copy whole words, then mop up any trailing bytes.
                 */
                t = length / wsize;
                TLOOP(*(word *)dst = *(word *)src; src += wsize; dst += wsize);
                t = length & wmask;
                TLOOP(*dst++ = *src++);
        } else {
                /*
                 * Copy backwards.  Otherwise essentially the same.
                 * Alignment works as before, except that it takes
```

```
                 * (t&wmask) bytes to align, not wsize-(t&wmask).
                 */
                src += length;
                dst += length;
                t = (int)src;
                if ((t | (int)dst) & wmask) {
                        if ((t ^ (int)dst) & wmask || length <= wsize)
                                t = length;
                        else
                                t &= wmask;
                        length -= t;
                        TLOOP1(*--dst = *--src);
                }
                t = length / wsize;
                TLOOP(src -= wsize; dst -= wsize; *(word *)dst = *(word *)src);
                t = length & wmask;
                TLOOP(*--dst = *--src);
        }
done:
        return (dst0);
}
```

The MC68020 assembly code generated by the Gnu C compiler with optimization.

```
0x2550 <memmove>:       linkw fp,#0
0x2554 <memmove+4>:     moveml d2-d4,sp@-
0x2558 <memmove+8>:     movel fp@(8),d3
0x255c <memmove+12>:    movel fp@(16),d2
0x2560 <memmove+16>:    moveal d3,a1
0x2562 <memmove+18>:    moveal fp@(12),a0
0x2566 <memmove+22>:    beq 0x2604 <memmove+180>
0x256a <memmove+26>:    cmpal d3,a0
0x256c <memmove+28>:    beq 0x2604 <memmove+180>
0x2570 <memmove+32>:    bls 0x25bc <memmove+108>
0x2572 <memmove+34>:    movel a0,d1
0x2574 <memmove+36>:    movel d1,d0
0x2576 <memmove+38>:    orl d3,d0
0x2578 <memmove+40>:    movel #3,d4
0x257a <memmove+42>:    andl d4,d0
0x257c <memmove+44>:    beq 0x25a2 <memmove+82>
0x257e <memmove+46>:    movel d1,d0
0x2580 <memmove+48>:    eorl d3,d0
0x2582 <memmove+50>:    movel #3,d4
0x2584 <memmove+52>:    andl d4,d0
0x2586 <memmove+54>:    bne 0x258e <memmove+62>
0x2588 <memmove+56>:    movel #3,d4
0x258a <memmove+58>:    cmpl d2,d4
0x258c <memmove+60>:    bcs 0x2592 <memmove+66>
0x258e <memmove+62>:    movel d2,d1
0x2590 <memmove+64>:    bra 0x259a <memmove+74>
0x2592 <memmove+66>:    movel #3,d0
0x2594 <memmove+68>:    andl d1,d0
0x2596 <memmove+70>:    movel #4,d1
0x2598 <memmove+72>:    subl d0,d1
```

```
0x259a <memmove+74>:    subl d1,d2
0x259c <memmove+76>:    moveb a0@+,a1@+
0x259e <memmove+78>:    subl #1,d1
0x25a0 <memmove+80>:    bne 0x259c <memmove+76>
0x25a2 <memmove+82>:    movel d2,d1
0x25a4 <memmove+84>:    lsrl #2,d1
0x25a6 <memmove+86>:    beq 0x25ae <memmove+94>
0x25a8 <memmove+88>:    movel a0@+,a1@+
0x25aa <memmove+90>:    subl #1,d1
0x25ac <memmove+92>:    bne 0x25a8 <memmove+88>
0x25ae <memmove+94>:    movel #3,d1
0x25b0 <memmove+96>:    andl d2,d1
0x25b2 <memmove+98>:    beq 0x2604 <memmove+180>
0x25b4 <memmove+100>:   moveb a0@+,a1@+
0x25b6 <memmove+102>:   subl #1,d1
0x25b8 <memmove+104>:   bne 0x25b4 <memmove+100>
0x25ba <memmove+106>:   bra 0x2604 <memmove+180>
0x25bc <memmove+108>:   addal d2,a0
0x25be <memmove+110>:   addal d2,a1
0x25c0 <memmove+112>:   movel a0,d1
0x25c2 <memmove+114>:   movel a1,d0
0x25c4 <memmove+116>:   orl d1,d0
0x25c6 <memmove+118>:   movel #3,d4
0x25c8 <memmove+120>:   andl d4,d0
0x25ca <memmove+122>:   beq 0x25ec <memmove+156>
0x25cc <memmove+124>:   movel a1,d0
0x25ce <memmove+126>:   eorl d1,d0
0x25d0 <memmove+128>:   movel #3,d4
0x25d2 <memmove+130>:   andl d4,d0
0x25d4 <memmove+132>:   bne 0x25dc <memmove+140>
0x25d6 <memmove+134>:   movel #4,d4
0x25d8 <memmove+136>:   cmpl d2,d4
0x25da <memmove+138>:   bcs 0x25e0 <memmove+144>
0x25dc <memmove+140>:   movel d2,d1
0x25de <memmove+142>:   bra 0x25e4 <memmove+148>
0x25e0 <memmove+144>:   movel #3,d4
0x25e2 <memmove+146>:   andl d4,d1
0x25e4 <memmove+148>:   subl d1,d2
0x25e6 <memmove+150>:   moveb a0@-,a1@-
0x25e8 <memmove+152>:   subl #1,d1
0x25ea <memmove+154>:   bne 0x25e6 <memmove+150>
0x25ec <memmove+156>:   movel d2,d1
0x25ee <memmove+158>:   lsrl #2,d1
0x25f0 <memmove+160>:   beq 0x25f8 <memmove+168>
0x25f2 <memmove+162>:   movel a0@-,a1@-
0x25f4 <memmove+164>:   subl #1,d1
0x25f6 <memmove+166>:   bne 0x25f2 <memmove+162>
0x25f8 <memmove+168>:   movel #3,d1
0x25fa <memmove+170>:   andl d2,d1
0x25fc <memmove+172>:   beq 0x2604 <memmove+180>
0x25fe <memmove+174>:   moveb a0@-,a1@-
0x2600 <memmove+176>:   subl #1,d1
0x2602 <memmove+178>:   bne 0x25fe <memmove+174>
0x2604 <memmove+180>:   movel d3,d0
```

```
0x2606 <memmove+182>:   moveml fp@(-12),d2-d4
0x260c <memmove+188>:   unlk fp
0x260e <memmove+190>:   rts
```

We follow our formulation described in Chapter 5. The first step is therefore to formalize the precondition on the initial state memmove-statep(s, *str1*, n, *lst1*, *str2*, *lst2*), the time function memmove-t(*str1*, *str2*, n, *lst1*, *lst2*), and the behavior function memmove(*str1*, *str2*, n, *lst1*, *lst2*). Only 'memmove-statep' is given here. The definition of the other two functions, though quite lengthy, are straightforward.

DEFINITION:
memmove-statep(s, *str1*, n, *lst1*, *str2*, *lst2*)
= ((mc-status(s) = 'running)
  ∧ evenp(mc-pc(s))
  ∧ rom-addrp(mc-pc(s), mc-mem(s), 192)
  ∧ mcode-addrp(mc-pc(s), mc-mem(s), MEMMOVE-CODE)
  ∧ ram-addrp(sub(32, 16, read-sp(s)), mc-mem(s), 32)
  ∧ ram-addrp(*str1*, mc-mem(s), n)
  ∧ mem-lst(1, *str1*, mc-mem(s), n, *lst1*)
  ∧ ram-addrp(*str2*, mc-mem(s), n)
  ∧ mem-lst(1, *str2*, mc-mem(s), n, *lst2*)
  ∧ disjoint(sub(32, 16, read-sp(s)), 32, *str1*, n)
  ∧ disjoint(sub(32, 16, read-sp(s)), 32, *str2*, n)
  ∧ (*str1* = read-mem(add(32, read-sp(s), 4), mc-mem(s), 4))
  ∧ (*str2* = read-mem(add(32, read-sp(s), 8), mc-mem(s), 4))
  ∧ (n = uread-mem(add(32, read-sp(s), 12), mc-mem(s), 4))
  ∧ uint-rangep(nat-to-uint(*str1*) + n, 32)
  ∧ uint-rangep(nat-to-uint(*str2*) + n, 32))

In the definition of 'memmove-statep', we have not asserted that the objects pointed by $str1$ and $str2$ do not overlap. But we do have to assert that a certain portion of the stack must not overlap with the objects pointed by $str1$ and $str2$.

The following theorem 'memmove-correctness' gives the correctness of this function.

THEOREM: memmove-correctness
let sn **be** stepn(s, memmove-t(*str1*, *str2*, n, *lst1*, *lst2*))
in
memmove-statep(s, *str1*, n, *lst1*, *str2*, *lst2*)
⇒ ((mc-status(sn) = 'running)
  ∧ (mc-pc(sn) = rts-addr(s))
  ∧ (read-rn(32, 14, mc-rfile(sn))
    = read-rn(32, 14, mc-rfile(s)))
  ∧ (read-rn(32, 15, mc-rfile(sn))
    = add(32, read-sp(s), 4))
  ∧ ((d2-7a2-5p(rn) ∧ (*oplen* ≤ 32))

$\Rightarrow$ (read-rn (*oplen*, *rn*, mc-rfile (*sn*))
= read-rn (*oplen*, *rn*, mc-rfile (*s*))))
$\wedge$ ((disjoint (*x*, *k*, sub (32, 16, read-sp (*s*)), 32)
$\wedge$ disjoint (*x*, *k*, *str1*, *n*)
$\wedge$ disjoint (*x*, *k*, *str2*, *n*))
$\Rightarrow$ (read-mem (*x*, mc-mem (*sn*), *k*)
= read-mem (*x*, mc-mem (*s*), *k*)))
$\wedge$ (read-dn (32, 0, *sn*) = *str1*)
$\wedge$ mem-lst (1,
str1,
mc-mem (*sn*),
n,
memmove (*str1*, *str2*, *n*, *lst1*, *lst2*))) **endlet**

While the other conjuncts are standard, the last two conjuncts give us the functional behavior of this function: after the execution of this program, the content of data register D0 is equal to *str*1, and the object pointed to by *str*1 is equal to memmove (*str1*, *str2*, *n*, *lst1*, *lst2*). The following theorem further proves that the new object pointed to by *str*1 is correct, according to the standard.

THEOREM: memmove-thm1
($j < n$)
$\Rightarrow$ (get-nth (j, memmove (*str1*, *str2*, *n*, *lst1*, *lst2*)) = get-nth (j, *lst2*))

7.2.2 Proving the `strstr` Function Correct

The second example is the `strstr` function, which is one of the most complicated search functions in the library. The interesting feature of this function is that it calls the string functions `strlen` and `strncmp` in the Berkeley implementation, which provides us a rather realistic suite to test our ability to handle subroutine calling.

```
/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 */

/* find pointer to first occurrence of find[] in s[] */
char *
strstr(s, find)
        register const char *s, *find;
{
        register char c, sc;
        register size_t len;

        if ((c = *find++) != 0) {
                len = strlen(find);
```

```
                do {
                        do {
                                if ((sc = *s++) == 0)
                                        return (NULL);
                        } while (sc != c);
                } while (strncmp(s, find, len) != 0);
                s--;
        }
        return ((char *)s);
}
```

The MC68020 assembly code generated by the Gnu C compiler with optimization.

```
0x2718 <strstr>:        linkw fp,#0
0x271c <strstr+4>:      moveml d2-d3/a2-a3,sp@-
0x2720 <strstr+8>:      moveal fp@(8),a2
0x2724 <strstr+12>:     moveal fp@(12),a3
0x2728 <strstr+16>:     moveb a3@+,d2
0x272a <strstr+18>:     beq 0x275a <strstr+66>
0x272c <strstr+20>:     movel a3,sp@-
0x272e <strstr+22>:     jsr @#0x25b0 <strlen>
0x2734 <strstr+28>:     movel d0,d3
0x2736 <strstr+30>:     addqw #4,sp
0x2738 <strstr+32>:     moveb a2@+,d0
0x273a <strstr+34>:     bne 0x2740 <strstr+40>
0x273c <strstr+36>:     clrl d0
0x273e <strstr+38>:     bra 0x275c <strstr+68>
0x2740 <strstr+40>:     cmpb d0,d2
0x2742 <strstr+42>:     bne 0x2738 <strstr+32>
0x2744 <strstr+44>:     movel d3,sp@-
0x2746 <strstr+46>:     movel a3,sp@-
0x2748 <strstr+48>:     movel a2,sp@-
0x274a <strstr+50>:     jsr @#0x2608 <strncmp>
0x2750 <strstr+56>:     addaw #12,sp
0x2754 <strstr+60>:     tstl d0
0x2756 <strstr+62>:     bne 0x2738 <strstr+32>
0x2758 <strstr+64>:     subqw #1,a2
0x275a <strstr+66>:     movel a2,d0
0x275c <strstr+68>:     moveml fp@(-16),d2-d3/a2-a3
0x2762 <strstr+74>:     unlk fp
0x2764 <strstr+76>:     rts
```

First, the precondition strstr-statep(s, *str1*, *n1*, *lst1*, *str2*, *n2*, *lst2*), the time function strstr-t(*n1*, *lst1*, *n2*, *lst2*), and the bahavior function strstr(*n1*, *lst1*, *n2*, *lst2*) are defined. Like the preceding example, only 'strstr-statep' is given as follows.

Conservative Axiom: strstr-load

strstr-loadp(s)

$=$ (evenp(STRSTR-ADDR)

$\wedge$ (STRSTR-ADDR $\in$ **N**)

```
    ∧  nat-rangep (STRSTR-ADDR, 32)
    ∧  rom-addrp (STRSTR-ADDR, mc-mem (s), 78)
    ∧  mcode-addrp (STRSTR-ADDR, mc-mem (s), STRSTR-CODE)
    ∧  strlen-loadp (s)
    ∧  strncmp-loadp (s)
    ∧  (pc-read-mem (add (32, STRSTR-ADDR, 24), mc-mem (s), 4)
        =  STRLEN-ADDR)
    ∧  (pc-read-mem (add (32, STRSTR-ADDR, 52), mc-mem (s), 4)
        =  STRNCMP-ADDR))
```

Simultaneously, we introduce the new function symbols 'strstr-loadp' and 'strstr-addr'.

```
DEFINITION:
strstr-statep (s, str1, n1, lst1, str2, n2, lst2)
=  ((mc-status (s) = 'running)
    ∧  strstr-loadp (s)
    ∧  (mc-pc (s) = STRSTR-ADDR)
    ∧  ram-addrp (sub (32, 48, read-sp (s)), mc-mem (s), 60)
    ∧  ram-addrp (str1, mc-mem (s), n1)
    ∧  mem-lst (1, str1, mc-mem (s), n1, lst1)
    ∧  ram-addrp (str2, mc-mem (s), n2)
    ∧  mem-lst (1, str2, mc-mem (s), n2, lst2)
    ∧  disjoint (str1, n1, sub (32, 48, read-sp (s)), 60)
    ∧  disjoint (str2, n2, sub (32, 48, read-sp (s)), 60)
    ∧  (str1 = read-mem (add (32, read-sp (s), 4), mc-mem (s), 4))
    ∧  (str2 = read-mem (add (32, read-sp (s), 8), mc-mem (s), 4))
    ∧  (slen (0, n1, lst1) < n1)
    ∧  (slen (0, n2, lst2) < n2)
    ∧  (n1 ≯ 0)
    ∧  uint-rangep (n1, 32)
    ∧  (n2 ≯ 0)
    ∧  uint-rangep (n2, 32))
```

There are a few interesting things in 'strstr-statep' that deserve some explanation. First, we have specified that the long word at (STRSTR-ADDR +24) be STRLEN-ADDR, which is the address of the function `strlen`, and the long word at (STRSTR-ADDR +52) be STRNCMP-ADDR, which is the address of the function `strncmp`. Second, how to specify a null terminated string has perplexed us for a while. Our current solution is to introduce an upper bound on the number of characters in the string. In 'strstr-statep', we have introduced two new variables $n1$ and $n2$, which are used as bounds for string $str1$ and $str2$, respectively.

The following theorem 'strstr-correctness' gives the correctness of this function.

```
THEOREM: strstr-correctness
let sn  be stepn (s, strstr-t (n1, lst1, n2, lst2))
in
strstr-statep (s, str1, n1, lst1, str2, n2, lst2)
```

```
⇒  ((mc-status (sn) = 'running)
    ∧  (mc-pc (sn) = rts-addr (s))
    ∧  (read-rn (32, 14, mc-rfile (sn))
         =  read-rn (32, 14, mc-rfile (s)))
    ∧  (read-rn (32, 15, mc-rfile (sn))
         =  add (32, read-sp (s), 4))
    ∧  ((d2-7a2-5p (rn) ∧ (oplen ≤ 32))
         ⇒  (read-rn (oplen, rn, mc-rfile (sn))
              =  read-rn (oplen, rn, mc-rfile (s))))
    ∧  (disjoint (x, k, sub (32, 48, read-sp (s)), 60)
         ⇒  (read-mem (x, mc-mem (sn), k)
              =  read-mem (x, mc-mem (s), k)))
    ∧  (read-dn (32, 0, sn)
         =  if strstr (n1, lst1, n2, lst2)
            then add (32, str1, strstr* (n1, lst1, n2, lst2))
            else 0 endif)) endlet
```

In particular, the last conjunct in the above theorem gives us the functional behavior of this function: after the execution of this program, the content of data register D0 is equivalent to strstr(*n1*, *lst1*, *n2*, *lst2*). The next three theorems further proves that this function is correct, according to the standard.

```
Theorem: strstr1-thm1
let j  be strstr1 (i, n1, lst1, n2, lst2, len)
in
((j ∈ N) ∧ (n = (1 + len)))
⇒  (strncmp2 (j, n, lst1, 0, lst2) = 0) endlet

Theorem: strstr-thm2
(lst-of-chrp (lst1)
 ∧  lst-of-chrp (lst2)
 ∧  (j < strstr (n1, lst1, n2, lst2))
 ∧  (n2 ≠ 0))
⇒  (strncmp (strlen (0, n2, lst2), mcdr (j, lst1), lst2) ≠ 0)

Theorem: strstr-thm3
(lst-of-chrp (lst1)
 ∧  lst-of-chrp (lst2)
 ∧  (¬ strstr (n1, lst1, n2, lst2))
 ∧  (j < strlen (0, n1, lst1))
 ∧  (n2 ≠ 0))
⇒  (strncmp (strlen (0, n2, lst2), mcdr (j, lst1), lst2) ≠ 0)
```

7.3 Programming Errors

Generally, people believe that detecting errors in machine code programs is hopelessly hard. But our experience with machine code program proving indicates that finding bugs seems to be no harder than finding proofs in our framework. The

discovery of programming errors comes naturally as a by-product in the process of the proofs. We add this short section to explain the three programming errors we found in the process of verifying the Berkeley Unix C string library and to report our experience in finding them.

7.3.1 The Bug in the Berkeley `strxfrm` Function

The first programming error we found is in the Berkeley C string function `strxfrm`, which went undetected in BSD4.3, and will be corrected for the release of BSD4.4.

According to its specification, the `strxfrm(s1, s2, n)` function returns the length of the string `s2`. But when we attempted to prove that the data register D0 has the length of `s2` after an execution of this function, we found that this was not a true theorem for the Berkeley implementation. And then the error was detected.

The bug can easily be seen in the corresponding Berkeley C code.

```
register size_t r = 0;
register int c;

/*
 * Since locales are unimplemented, this is just a copy.
 */
if (n != 0) {
        while ((c = *src++) != 0) {
                r++;
                if (--n == 0) {
                        while (*src++ != 0)
                                r++;
                        break;
                }
                *dst++ = c;
        }
        *dst = 0;
}
return (r);
```

Evidently, in the case of `n == 0`, this function returns `0`, rather than the length of the string `s2`.

7.3.2 The Bug in the Berkeley `memmove` Function

The second programming error we found is in the Berkeley C string function `memmove`, which was detected on June 21, 1991, and has been corrected in the latest version of BSD4.3.

According to its specification, `memmove(src, dst, length)` returns the value of `src`. But we failed to prove that the data register D0 has the value of `src` after an excution of this function.

The bug, shown in the following two lines from the Berkeley C code, is extremely simple.

```
if (length == 0 || dst == src)          /* nothing to do */
        return;
```

As it shows, in the cases of `length == 0` or `dst == src`, this function does not return the value of `src`. In the latest version of the library, the second line has been corrected to "`goto done;`".

7.3.3 The Bug in Plauger's `strtok` Function

The third programming error we found is not in the Berkeley C string library, but in the `strtok` function of [44].[4] This error had been detected by the author before we reported it to him.

The bug is that the erroneous `strtok` function will dereference a null pointer in some situation. In our proof attempts, the theorem prover kept "complaining" that it could not prove that memory location 0 is readable. Based on this information, we carefully studied the C code again, and detected the error that occurs in the following three lines of code from the `strtok` function.

```
send = strpbrk(sbegin, str2);
if (*send != '\0')
  *send++ = '\0';
```

In the case that `strpbrk(sbegin, str2)` is `NULL`, the first line will assign `send` to be `NULL`, and this will cause an error when `send` is dereferenced in the second line.

[4] After deciding to study the Standard C String Library, we looked into three implementations: the Berkeley, the Plauger, and the Gnu.

Chapter 8

Conclusions

The main goal of this dissertation is to build a powerful proof system on top of Nqthm that can be used to mechanically verify MC68020 machine code programs. Our experimentation with many realistic, though very small, machine code programs demonstrates that we have achieved this goal. We believe that the work reported in this dissertation is the first instance of the formal verification, by an automated reasoning system, of the binary code for "real" software produced by "industrial strength" high-level compilers targeting a widely used microprocessor whose semantics was formalized with an operational semantics in the logic of the reasoning system used. We are optimistic that the verification techniques developed in this work can be applied to many programs on many different microprocessors and for many different higher-level language compilers. In this final chapter, we summarize our main results and contributions, point out possible applications of our methodology, and speculate on future research directions.

8.1 The State of the Project

The work described in this dissertation consists of three major components:

1. We have formally described a substantial subset of the user mode of the MC68020 microprocessor at the instruction-set level. The formal specification is given as an interpreter in the formal logic of Nqthm, and is about 128,000 bytes long, which takes up approximately 80 pages of text.

2. We have developed a mathematical theory for machine code reasoning, which has been mechanized as a lemma library in the automated reasoning system Nqthm. Each of the lemmas in the library is mechanically checked by Nqthm. Our library of lemmas is about 250,000 bytes, or 140 pages, long. It consists of approximately 1500 lemmas.

3. We have mechanically verified several dozen MC68020 machine code programs. Most of the machine code programs are the object code produced by the Gnu C compilers from their C counterparts. Primarily to provide concrete evidence that this work is easily applicable to many other languages than C, we have also mechanically verified the object code produced by the Verdix Ada compiler for an integer square root algorithm. Furthermore, we have mechanically verified the object code produced by the AKCL Common Lisp compiler for a 'fixnum' GCD program. The programs verified include some of the C functions in Kernighan and Ritchie's book [32], in particular binary search and Quick Sort, a majority voting program, and the Berkeley implementation of the ANSI/ISO standard C string library.

We have worked on this project for about three years. The lemma library is the most time consuming (and rewarding) part of this project.

8.2 The Significance of the Project

We believe the main contributions of this work are two-fold.

Scientifically, we have demonstrated the feasibility of mechanically proving the correctness of machine code programs, in particular object code produced by high-level language compilers. The methodologies used and developed in this work provide a general framework for program proving. Our approach to program proving can be characterized simply as symbolic execution and theorem proving with an interpreter semantics in a computational logic. The successful integration of various program proving techniques with a powerful theorem prover makes our approach very promising.

Practically, we have provided a means to verify mechanically MC68020 machine code programs. The main product of this work is a powerful verification system, built on top of Nqthm, consisting of a formal model for the MC68020 microprocessor and a lemma library for machine code reasoning, that can be used to verify a few thousands lines of MC68020 machine code programs, as evidenced by those examples presented in this thesis.

The main objection against machine-code verification is that proofs are too complicated and we can prove nothing but toy programs. We hope this thesis has provided enough evidence that some small but real pieces of software are within our reach. To scale up further, some automated tools must be developed to assist

the user in the proof. At the present time, we have not implemented any such tools. But we have been considering the implementation of something like a verification condition generator for machine code reasoning that can be used to automatically generate proof obligations.

We have learned a great deal through this project. For any one who plans to do some work of this nature, we offer the following advice:

1. It is crucial to work closely with an expert designer who knows the processor architecture very well. On the one hand, such an expert can answer various questions concerning the architecture. On the other hand, we are obliged to convince him of the correctness of the formal specification.

2. It is extremely helpful if we have the access to the designer's architectural simulator. While there is no way of guaranteeing the correctness of the formal specification, some sort of testing and simulation to compare the designer's architectural simulator and the formal specification would help us gain more confidence in our modelling.

3. When the intended application to formal specifications is mechanical mathematical reasoning, formal specifications should be developed together with the intended applications and the mechanical theorem proving tool being used, because the form and details of a specification dramatically influence the resulting proof obligations and the behavior of the theorem prover. While each formal specification language has its style of thinking, each theorem proving system has certainly its way of reasoning.

Most importantly, we have learned how to formalize and mechanize the mathematics for machine code program proving in a computational logic.

8.3 Future Work

There are a number of potentially important areas for future research building upon this work.

First of all, the success of this work directly suggests that we investigate:

- The correctness of some moderate-sized piece of software that is in critical use. One good example is the verification of microcontroller programs, an important

issue that has been largely ignored by the formal verification community due to the lack of formal methods to handle lower level code.

- The analysis of real-time execution bounds of programs. By reasoning at the object-code level, we are able to prove properties about real-time behavior for some programs, which is an advantage over many higher-level language approaches.

- The correctness of high-level programming language compilers. Even though compiler verification may have little practical impact in the near future, it is a research area with many interesting problems.

- The correctness of some lower-level software, e.g., software for cache and memory management. This has been one of our main motivations.

We believe that success in any of these areas would be a major contribution to formal methods.

As a next step, we plan to recast and reapply what we have learned in this work to another computer architecture. We believe we can do it but dealing with some issues, such as the nondeterminism introduced via instructions such as 'delayed branch,' which may leave the program counter in an indeterminate state during some instructions, will be challenging. We have been investigating the idea of doing similar work on the SPARC [49] and Alpha [48] architectures.

Currently, we have left out the supervisor mode of the MC68020 microprocessor in our MC68020 formal model. Specifying supervisor mode is a very challenging, but extremely important research topic. We would certainly consider this supervisor mode issue in any future research along the lines of this work.

Some microprocessor architectures, such as Alpha, support on-chip floating point arithmetic. Specifying floating point instructions and verifying floating point programs would be an adventure we have not attended to. We speculate that formal specification of floating point arithmetic perhaps would not pose too great of a challenge, but the formal verification of floating point programs would be extremely difficult, if not impossible.

Appendix A

Syntax Summary

Here is a summary of the conventional syntax used in this report in terms of the official syntax of the Nqthm logic described in [9]. (`cond` and `let` are recent extensions not described in [9].)

1. Variables. x, y, z, etc. are printed in italics.

2. Function application. For any function symbol for which special syntax is not given below, an application of the symbol is printed with the usual notation; e.g., the term `(fn x y z)` is printed as fn(x, y, z). Note that the function symbol is printed in Roman. In the special case that 'c' is a function symbol of no arguments, i.e., it is a constant, the term `(c)` is printed merely as C, in small caps, with no trailing parentheses. Because variables are printed in italics, there is no confusion between the printing of variables and constants.

3. Other constants. **t**, **f**, and **nil** are printed in bold. Quoted constants are printed in the ordinary fashion of the Nqthm logic, e.g., `'(a b c)` is still printed just that way. `#b001` is printed as 001_2, `#o765` is printed as 765_8, and `#xA9` is printed as $A9_{16}$.

4. `(if x y z)` is printed as

   if x **then** y **else** z **endif**.

5. `(cond (test1 value1) (test2 value2) (t value3))` is printed as

   if *test1* **then** *value1* **elseif** *test2* **then** *value2* **else** *value3* **endif**.

6. `(let ((var1 val1) (var2 val2)) form)` is printed as

   let *var1* **be** *val1*, *var2* **be** *val2* **in** *form* **endlet**.

7. The remaining function symbols that are printed specially are described in the following table.

Nqthm Syntax	Conventional Syntax
`(or x y)`	$x \vee y$
`(and x y)`	$x \wedge y$
`(times x y)`	$x * y$
`(plus x y)`	$x + y$
`(remainder x y)`	$x \bmod y$
`(quotient x y)`	$x \div y$
`(difference x y)`	$x - y$
`(implies x y)`	$x \rightarrow y$
`(member x y)`	$x \in y$
`(geq x y)`	$x \geq y$
`(greaterp x y)`	$x > y$
`(leq x y)`	$x \leq y$
`(lessp x y)`	$x < y$
`(equal x y)`	$x = y$
`(not (member x y))`	$x \notin y$
`(not (geq x y))`	$x \ngeq y$
`(not (greaterp x y))`	$x \ngtr y$
`(not (leq x y))`	$x \nleq y$
`(not (lessp x y))`	$x \nless y$
`(not (equal x y))`	$x \neq y$
`(minus x)`	$-x$
`(add1 x)`	$1 + x$
`(nlistp x)`	$x \simeq \mathbf{nil}$
`(zerop x)`	$x \simeq 0$
`(numberp x)`	$x \in \mathbf{N}$
`(sub1 x)`	$x - 1$
`(not (nlistp x))`	$x \not\simeq \mathbf{nil}$
`(not (zerop x))`	$x \not\simeq 0$
`(not (numberp x))`	$x \notin \mathbf{N}$

Appendix B

The Nqthm Script of The MC68020 Model and Lemma Library

Appendix B provides a complete documentation of our MC68020 formal specification and the lemma library developed to facilitate machine code reasoning.

B.1 An Integer Sublibrary

This section contains some preliminary theories that is useful in our MC68020 specification.

```
#|
------------------------------------------------------------------------
Date:      Jan, 1991
Modified:  May 18, 1992.
File:      mc20-0.events
------------------------------------------------------------------------

                  AN ARITHMETIC SUBLIBRARY
|#

; before we start to write our specification, some preliminary theories
; have to be established.

;              THEOREMS ABOUT SET AND BAG
; subset relation.
(defn subset (x y)
  (if (listp x)
      (if (member (car x) y)
          (subset (cdr x) y)
        f)
    t))

; delete the elements of x from the set y.
(defn delete (x y)
  (if (listp y)
      (if (equal x (car y))
          (cdr y)
        (cons (car y) (delete x (cdr y))))
    y))
```

```
; determines whether x is a subbag of y.
(defn subbagp (x y)
  (if (listp x)
      (if (member (car x) y)
          (subbagp (cdr x) (delete (car x) y))
        f)
    t))

; the difference.
(defn bagdiff (x y)
  (if (listp y)
      (if (member (car y) x)
          (bagdiff (delete (car y) x) (cdr y))
        (bagdiff x (cdr y)))
    x))

; the intersection.
(defn bagint (x y)
  (if (listp x)
      (if (member (car x) y)
          (cons (car x)
                (bagint (cdr x) (delete (car x) y)))
        (bagint (cdr x) y))
    nil))

(prove-lemma delete-non-member (rewrite)
     (implies (not (member x y))
              (equal (delete x y) y)))

(prove-lemma member-delete (rewrite)
     (implies (member x (delete u v))
              (member x v)))

(prove-lemma delete-commutativity (rewrite)
     (equal (delete x (delete y z))
            (delete y (delete x z))))

(prove-lemma subbagp-delete (rewrite)
     (implies (subbagp x (delete u y))
              (subbagp x y)))

(prove-lemma subbagp-cdr1 (rewrite)
     (implies (subbagp x y)
              (subbagp (cdr x) y)))

(prove-lemma subbagp-cdr2 (rewrite)
     (implies (subbagp x (cdr y))
              (subbagp x y)))

(prove-lemma subbagp-bagint1 (rewrite)
     (subbagp (bagint x y) x))

(prove-lemma subbagp-bagint2 (rewrite)
     (subbagp (bagint x y) y))
```

```
;             THEOREMS ABOUT NATURAL NUMBERS
; lemmas about lessp.
(prove-lemma lessp-of-1 (rewrite)
     (equal (lessp x 1) (zerop x)))

(prove-lemma lessp-sub1 (rewrite)
     (equal (lessp (sub1 x) x) (not (zerop x))))

; lemmas about plus.
(prove-lemma plus-add1 (rewrite)
     (equal (plus 1 x) (add1 x)))

(prove-lemma plus-add1-1 (rewrite)
     (equal (plus x (add1 y))
            (add1 (plus x y))))

(prove-lemma plus-commutativity (rewrite)
     (equal (plus x y) (plus y x)))

(prove-lemma plus-commutativity1 (rewrite)
     (equal (plus x (plus y z))
            (plus y (plus x z))))

(prove-lemma plus-associativity (rewrite)
     (equal (plus (plus x y) z)
            (plus x (plus y z))))

(prove-lemma plus-equal-cancel0 (rewrite)
     (equal (equal (plus x y) x)
            (and (numberp x) (zerop y))))

(prove-lemma plus-equal-cancel (rewrite)
     (equal (equal (plus x y) (plus x z))
            (equal (fix y) (fix z))))

(prove-lemma plus-lessp-cancel-0 (rewrite)
     (equal (lessp x (plus x y))
            (not (zerop y))))

(prove-lemma plus-lessp-cancel-1 (rewrite)
     (equal (lessp (plus x y) (plus x z))
            (lessp y z)))

(prove-lemma plus-lessp-cancel-add1 (rewrite)
     (equal (lessp (plus y x) (add1 y))
            (zerop x)))

(prove-lemma plus-equal-0 (rewrite)
     (equal (equal (plus x y) 0)
            (and (zerop x) (zerop y))))

; lemmas about difference.
(prove-lemma sub1-of-1 (rewrite)
```

```
     (equal (equal (sub1 x) 0)
            (or (zerop x) (equal x 1))))

(prove-lemma difference-sub1 (rewrite)
     (equal (difference x 1) (sub1 x)))

(prove-lemma difference-sub1-sub1 (rewrite)
     (equal (sub1 (difference x (sub1 y)))
            (if (zerop y)
                (sub1 x)
              (if (lessp x y) 0 (difference x y)))))

(prove-lemma difference-0 (rewrite)
     (implies (leq x y)
              (equal (difference x y) 0)))

(prove-lemma difference-x-x (rewrite)
     (equal (difference x x) 0))

(prove-lemma difference-plus-cancel0 (rewrite)
     (and (equal (difference (plus x y) x)
                 (fix y))
          (equal (difference (plus y x) x)
                 (fix y))))

(prove-lemma difference-plus1 (rewrite)
     (implies (not (lessp x y))
              (equal (plus (difference x y) z)
                     (difference (plus x z) y))))

(prove-lemma difference-plus2 (rewrite)
     (implies (not (lessp y z))
              (equal (plus x (difference y z))
                     (difference (plus x y) z))))

(prove-lemma difference-difference1 (rewrite)
     (equal (difference (difference x y) z)
            (difference x (plus y z))))

(prove-lemma difference-difference2 (rewrite)
     (implies (not (lessp y z))
              (equal (difference x (difference y z))
                     (difference (plus x z) y)))
     ((induct (difference y z))))

(prove-lemma difference-plus-cancel1 (rewrite)
     (equal (difference (plus x y) (plus x z))
            (difference y z)))

(prove-lemma difference-plus-cancel-add1 (rewrite)
     (equal (difference (plus y x) (add1 y))
            (sub1 x)))

(prove-lemma difference-lessp (rewrite)
```

```
     (equal (lessp (difference m n) m)
            (and (not (zerop m)) (not (zerop n)))))

(prove-lemma difference-lessp1 (rewrite)
     (implies (lessp x z)
              (equal (lessp (difference x y) z) t)))

(prove-lemma difference=0 (rewrite)
     (equal (equal 0 (difference x y))
            (not (lessp y x))))

(prove-lemma difference-equal-cancel-0 (rewrite)
     (equal (equal x (difference x y))
            (and (numberp x)
                 (or (equal x 0) (zerop y)))))

(prove-lemma difference-equal-cancel-1 (rewrite)
     (equal (equal (difference x z) (difference y z))
            (if (lessp x z)
                (not (lessp z y))
              (if (lessp y z)
                  (not (lessp z x))
                (equal (fix x) (fix y))))))

(prove-lemma difference-lessp-cancel (rewrite)
     (equal (lessp (difference a c) (difference b c))
            (if (leq c a) (lessp a b) (lessp c b))))

; meta lemmas for plus and difference.  Stolen from basic.events.
(defn plus-fringe (x)
  (if (and (listp x)
           (equal (car x) 'plus))
      (append (plus-fringe (cadr x))
              (plus-fringe (caddr x)))
    (cons x nil)))

(defn plus-tree (l)
  (if (nlistp l)
      ''0
    (if (nlistp (cdr l))
        (list 'fix (car l))
      (if (nlistp (cddr l))
          (list 'plus (car l) (cadr l))
        (list 'plus (car l) (plus-tree (cdr l)))))))

(prove-lemma numberp-eval$-plus (rewrite)
     (implies (equal (car x) 'plus)
              (numberp (eval$ t x a))))

(prove-lemma numberp-eval$-plus-tree (rewrite)
     (numberp (eval$ t (plus-tree l) a)))

(prove-lemma member-implies-plus-tree-greatereqp (rewrite)
     (implies (member x y)
```

```
             (not (lessp (eval$ t (plus-tree y) a) (eval$ t x a)))))

(prove-lemma plus-tree-delete (rewrite)
     (equal (eval$ t (plus-tree (delete x y)) a)
            (if (member x y)
                (difference (eval$ t (plus-tree y) a) (eval$ t x a))
              (eval$ t (plus-tree y) a))))

(prove-lemma subbagp-implies-plus-tree-geq (rewrite)
     (implies (subbagp x y)
              (not (lessp (eval$ t (plus-tree y) a)
                          (eval$ t (plus-tree x) a)))))

(prove-lemma plus-tree-bagdiff (rewrite)
     (implies (subbagp x y)
              (equal (eval$ t (plus-tree (bagdiff y x)) a)
                     (difference (eval$ t (plus-tree y) a)
                                 (eval$ t (plus-tree x) a)))))

(prove-lemma numberp-eval$-bridge (rewrite)
     (implies (equal (eval$ t z a) (eval$ t (plus-tree x) a))
              (numberp (eval$ t z a))))

(prove-lemma bridge-to-subbagp-implies-plus-tree-geq (rewrite)
     (implies (and (subbagp y (plus-fringe z))
                   (equal (eval$ t z a)
                          (eval$ t (plus-tree (plus-fringe z)) a)))
              (equal (lessp (eval$ t z a) (eval$ t (plus-tree y) a)) f)))

(prove-lemma eval$-plus-tree-append (rewrite)
     (equal (eval$ t (plus-tree (append x y)) a)
            (plus (eval$ t (plus-tree x) a)
                  (eval$ t (plus-tree y) a))))

(prove-lemma plus-tree-plus-fringe (rewrite)
     (equal (eval$ t (plus-tree (plus-fringe x)) a)
            (fix (eval$ t x a)))
     ((induct (plus-fringe x))))

(prove-lemma member-implies-numberp (rewrite)
     (implies (and (member c (plus-fringe x))
                   (numberp (eval$ t c a)))
        (numberp (eval$ t x a)))
     ((induct (plus-fringe x))))

(prove-lemma cadr-eval$-list (rewrite)
     (and (equal (car (eval$ 'list x a))
                 (eval$ t (car x) a))
          (equal (cdr (eval$ 'list x a))
                 (if (listp x) (eval$ 'list (cdr x) a) 0))))

(prove-lemma eval$-quote (rewrite)
     (equal (eval$ t (cons 'quote args) a)
            (car args)))
```

```
(prove-lemma listp-eval$ (rewrite)
     (equal (listp (eval$ 'list x a)) (listp x)))

; the meta lemma to cancel identical plus terms in equality.  For example,
;   (EQUAL (PLUS A B C) (PLUS B D E)) => (EQUAL (PLUS A C) (PLUS D E)).
(defn cancel-equal-plus (x)
  (if (and (listp x) (equal (car x) 'equal))
      (if (and (listp (cadr x)) (equal (caadr x) 'plus)
               (listp (caddr x)) (equal (caaddr x) 'plus))
          (list 'equal
                (plus-tree
                 (bagdiff (plus-fringe (cadr x))
                          (bagint (plus-fringe (cadr x))
                                  (plus-fringe (caddr x)))))
                (plus-tree
                 (bagdiff (plus-fringe (caddr x))
                          (bagint (plus-fringe (cadr x))
                                  (plus-fringe (caddr x))))))
        (if (and (listp (cadr x)) (equal (caadr x) 'plus)
                 (member (caddr x) (plus-fringe (cadr x))))
            (list 'if (list 'numberp (caddr x))
                  (list 'equal
                        (plus-tree
                         (delete (caddr x)
                                 (plus-fringe (cadr x))))
                        ''0)
                  (list 'quote f))
          (if (and (listp (caddr x)) (equal (caaddr x) 'plus)
                   (member (cadr x) (plus-fringe (caddr x))))
              (list 'if (list 'numberp (cadr x))
                    (list 'equal ''0
                          (plus-tree
                           (delete (cadr x)
                                   (plus-fringe (caddr x)))))
                    (list 'quote f))
            x)))
      x))

(prove-lemma correctness-of-cancel-equal-plus ((meta equal))
     (equal (eval$ t x a)
            (eval$ t (cancel-equal-plus x) a))
     ((disable eval$)))

; the meta lemma to cancel identical plus terms in lessp.  For example,
; (DIFFERENCE (PLUS A B C) (PLUS B D E)) => (DIFFERENCE (PLUS A C) (PLUS D E)).
(defn cancel-difference-plus (x)
  (if (and (listp x) (equal (car x) 'difference))
      (if (and (listp (cadr x)) (equal (caadr x) 'plus)
               (listp (caddr x)) (equal (caaddr x) 'plus))
          (list 'difference
                (plus-tree
                 (bagdiff (plus-fringe (cadr x))
                          (bagint (plus-fringe (cadr x))
```

```
                                (plus-fringe (caddr x)))))
                (plus-tree
                 (bagdiff (plus-fringe (caddr x))
                          (bagint (plus-fringe (cadr x))
                                  (plus-fringe (caddr x))))))
          (if (and (listp (cadr x)) (equal (caadr x) 'plus)
                   (member (caddr x) (plus-fringe (cadr x))))
              (plus-tree (delete (caddr x) (plus-fringe (cadr x))))
              (if (and (listp (caddr x)) (equal (caaddr x) 'plus)
                       (member (cadr x) (plus-fringe (caddr x))))
                  ''0
                  x)))
      x))

(prove-lemma correctness-of-cancel-difference-plus ((meta difference))
       (equal (eval$ t x a)
              (eval$ t (cancel-difference-plus x) a))
       ((disable eval$)))

; the meta lemma to cancel identical plus terms in lessp.  For example,
;   (LESSP (PLUS A B C) (PLUS B D E)) => (LESSP (PLUS A C) (PLUS D E)).
(defn cancel-lessp-plus (x)
  (if (and (listp x) (equal (car x) 'lessp))
      (if (and (listp (cadr x)) (equal (caadr x) 'plus)
               (listp (caddr x)) (equal (caaddr x) 'plus))
          (list 'lessp
                (plus-tree
                 (bagdiff (plus-fringe (cadr x))
                          (bagint (plus-fringe (cadr x))
                                  (plus-fringe (caddr x)))))
                (plus-tree
                 (bagdiff (plus-fringe (caddr x))
                          (bagint (plus-fringe (cadr x))
                                  (plus-fringe (caddr x))))))
          (if (and (listp (cadr x)) (equal (caadr x) 'plus)
                   (member (caddr x) (plus-fringe (cadr x))))
              (list 'quote f)
            (if (and (listp (caddr x)) (equal (caaddr x) 'plus)
                     (member (cadr x) (plus-fringe (caddr x))))
                (list 'not
                      (list 'zerop (plus-tree
                                    (delete (cadr x)
                                            (plus-fringe (caddr x))))))
              x)))
    x))

(prove-lemma correctness-of-cancel-lessp-plus ((meta lessp))
     (equal (eval$ t x a)
            (eval$ t (cancel-lessp-plus x) a))
     ((disable eval$)))

(prove-lemma plus-lessp-cancel-2 (rewrite)
     (equal (lessp (plus y x) (plus x z))
            (lessp y z)))
```

```
; lemmas about times.
(prove-lemma times-zero (rewrite)
     (implies (or (zerop x) (zerop y))
              (equal (times x y) 0)))

(prove-lemma times-distributes-plus (rewrite)
     (equal (times x (plus y z))
            (plus (times x y) (times x z))))

(prove-lemma times-add1 (rewrite)
     (equal (times x (add1 y))
            (plus x (times x y))))

(prove-lemma times-commutativity (rewrite)
     (equal (times x z)
            (times z x)))

(prove-lemma times-commutativity1 (rewrite)
     (equal (times x (times y z))
            (times y (times x z))))

(prove-lemma times-equal-0 (rewrite)
     (equal (equal (times x y) 0)
            (or (zerop x) (zerop y))))

(lemma times-equal-1 (rewrite)
       (equal (equal (times x y) 1)
              (and (equal x 1) (equal y 1)))
       ((induct (times x y))))

(prove-lemma times-1 (rewrite)
     (equal (times 1 x) (fix x)))

(prove-lemma times-add1-sub1 (rewrite)
     (equal (add1 (times a (sub1 b)))
            (if (or (zerop a) (zerop b))
                1
              (difference (times a b) (sub1 a)))))

(prove-lemma times-associativity (rewrite)
     (equal (times (times x y) z)
            (times x (times y z))))

; x is a boolean value, iff x is either T or F.
(defn boolean (x)
  (or (truep x) (falsep x)))

(prove-lemma equal-iff (rewrite)
     (implies (and (boolean p) (boolean q))
              (equal (equal p q)
                     (iff p q))))

(prove-lemma times-equal-cancel0 (rewrite)
```

```
     (and (equal (equal (times x y) y)
                 (and (numberp y)
                      (if (equal y 0) t (equal x 1))))
          (equal (equal (times y x) y)
                 (and (numberp y)
                      (if (equal y 0) t (equal x 1))))))

(prove-lemma times-equal-cancel (rewrite)
     (equal (equal (times x y) (times x z))
            (or (zerop x) (equal (fix y) (fix z))))
     ((induct (difference y z))))

(prove-lemma times-lessp-0 (rewrite)
     (equal (lessp 0 (times x y))
            (and (lessp 0 x) (lessp 0 y))))

(prove-lemma times-lessp-1 (rewrite)
     (equal (lessp 1 (times x y))
            (and (not (zerop x))
                 (not (zerop y))
                 (not (and (equal x 1) (equal y 1))))))

(prove-lemma times-lessp-cancel0 (rewrite)
     (and (equal (lessp (times x y) x)
                 (and (not (zerop x)) (zerop y)))
          (equal (lessp x (times x y))
                 (and (not (zerop x)) (not (zerop y)) (not (equal y 1))))))

(prove-lemma times-lessp-cancel (rewrite)
     (equal (lessp (times x y) (times x z))
            (and (not (zerop x)) (lessp y z))))

(disable equal-iff)

(prove-lemma times-lessp-cancel-1 (rewrite)
     (equal (lessp (times x y) (plus x (times x z)))
            (and (not (zerop x)) (leq y z))))

(prove-lemma times-lessp-linear (rewrite)
     (implies (not (lessp i j))
              (not (lessp (times a i) (times a j)))))

(prove-lemma times-distributes-difference (rewrite)
     (equal (difference (times x y) (times x z))
            (times x (difference y z)))
     ((induct (difference y z))))

(prove-lemma times-distributes-difference1 (rewrite)
     (equal (difference (times y x) (times z x))
            (times x (difference y z))))

(prove-lemma times2-add1-lessp-cancel (rewrite)
     (equal (lessp (add1 (times 2 i)) (times 2 j))
            (lessp i j)))
```

```
; meta lemmas for times.  Stolen and modified from naturals.events.
(defn times-fringe (x)
  (if (and (listp x)
           (equal (car x) 'times))
      (append (times-fringe (cadr x))
              (times-fringe (caddr x)))
    (cons x nil)))

(defn times-tree (x)
  (if (nlistp x)
      ''1
      (if (nlistp (cdr x))
          (list 'fix (car x))
        (if (nlistp (cddr x))
            (list 'times (car x) (cadr x))
          (list 'times (car x) (times-tree (cdr x)))))))

(defn and-not-zerop-tree (x)
  (if (nlistp x)
      '(true)
    (if (nlistp (cdr x))
        (list 'not (list 'zerop (car x)))
      (list 'and (list 'not (list 'zerop (car x)))
            (and-not-zerop-tree (cdr x))))))

(prove-lemma numberp-eval$-times (rewrite)
     (implies (equal (car x) 'times)
              (numberp (eval$ t x a))))

(prove-lemma eval$-times-tree-numberp (rewrite)
     (numberp (eval$ t (times-tree x) a)))

(prove-lemma eval$-times-member (rewrite)
     (implies (member e x)
              (equal (eval$ t (times-tree x) a)
                     (times (eval$ t e a)
                            (eval$ t (times-tree (delete e x)) a)))))

(prove-lemma zerop-makes-times-tree-zero (rewrite)
     (implies (and (not (eval$ t (and-not-zerop-tree x) a))
                   (subbagp x y))
              (equal (eval$ t (times-tree y) a) 0)))

(prove-lemma eval$-times-tree-append (rewrite)
     (equal (eval$ t (times-tree (append x y)) a)
            (times (eval$ t (times-tree x) a)
                   (eval$ t (times-tree y) a))))

(prove-lemma times-tree-times-fringe (rewrite)
     (equal (eval$ t (times-tree (times-fringe x)) a)
            (fix (eval$ t x a)))
     ((induct (times-fringe x))))
```

```
(prove-lemma eval$-lessp-times-tree-bagdiff (rewrite)
     (implies (and (eval$ t (and-not-zerop-tree x) a)
                   (subbagp x y)
                   (subbagp x z))
              (equal (lessp (eval$ t (times-tree (bagdiff y x)) a)
                            (eval$ t (times-tree (bagdiff z x)) a))
                     (lessp (eval$ t (times-tree y) a)
                            (eval$ t (times-tree z) a)))))

(prove-lemma zerop-makes-lessp-false-bridge (rewrite)
     (implies (not (eval$ t (and-not-zerop-tree
                             (bagint (times-fringe (cons 'times x))
                                     (times-fringe (cons 'times y))))
                          a))
              (equal (lessp (times (eval$ t (car x) a)
                                   (eval$ t (cadr x) a))
                            (times (eval$ t (car y) a)
                                   (eval$ t (cadr y) a)))
                     f))
     ((use (zerop-makes-times-tree-zero
            (x (bagint (times-fringe (cons 'times x))
                       (times-fringe (cons 'times y))))
            (y (times-fringe (cons 'times x))))
           (zerop-makes-times-tree-zero
            (x (bagint (times-fringe (cons 'times x))
                       (times-fringe (cons 'times y))))
            (y (times-fringe (cons 'times y)))))))

(prove-lemma and-not-zerop-tree-lessp (rewrite)
     (equal (eval$ t (and-not-zerop-tree x) a)
            (not (lessp (eval$ t (times-tree x) a) 1))))

(defn eval$-and-not-zerop-tree-end (w x a)
  (eval$ t (and-not-zerop-tree (delete w x)) a))

(prove-lemma and-not-zerop-tree-delete (rewrite)
     (implies (member w x)
              (equal (eval$ t (and-not-zerop-tree (delete w x)) a)
                     (if (zerop (eval$ t w a))
                         (eval$-and-not-zerop-tree-end w x a)
                       (not (lessp (eval$ t (times-tree x) a)
                                   (eval$ t w a)))))))

(disable and-not-zerop-tree-lessp)

(defn lessp-1-times-tree-delete-end (w x a)
  (lessp 1 (eval$ t (times-tree (delete w x)) a)))

(prove-lemma lessp-1-times-tree-delete (rewrite)
     (implies (member w x)
              (equal (lessp 1 (eval$ t (times-tree (delete w x)) a))
                     (if (zerop (eval$ t w a))
                         (lessp-1-times-tree-delete-end w x a)
                       (lessp (eval$ t w a)
```

```
                            (eval$ t (times-tree x) a))))))

(prove-lemma eval$-times-fringe-member-zero (rewrite)
     (implies (and (member e (times-fringe (cons 'times x)))
                   (zerop (eval$ t e a)))
              (equal (times (eval$ t (car x) a)
                            (eval$ t (cadr x) a))
                     0))
     ((use (eval$-times-member (x (times-fringe (cons 'times x)))))))

(defn cancel-lessp-times (x)
  (if (and (listp x) (equal (car x) 'lessp))
      (if (and (equal (caadr x) 'times)
               (equal (caaddr x) 'times))
          (if (listp (bagint (times-fringe (cadr x))
                             (times-fringe (caddr x))))
              (list 'and
                    (and-not-zerop-tree (bagint (times-fringe (cadr x))
                                                (times-fringe (caddr x))))
                    (list 'lessp
                          (times-tree
                           (bagdiff (times-fringe (cadr x))
                                    (bagint (times-fringe (cadr x))
                                            (times-fringe (caddr x)))))
                          (times-tree
                           (bagdiff (times-fringe (caddr x))
                                    (bagint (times-fringe (cadr x))
                                            (times-fringe (caddr x)))))))
              x)
        (if (and (listp (cadr x)) (equal (caadr x) 'times)
                 (member (caddr x) (times-fringe (cadr x))))
            (list 'and
                  (list 'not (list 'zerop (caddr x)))
                  (list 'not
                        (and-not-zerop-tree (delete (caddr x)
                                                    (times-fringe (cadr x))))))
          (if (and (listp (caddr x)) (equal (caaddr x) 'times)
                   (member (cadr x) (times-fringe (caddr x))))
              (list 'and
                    (list 'not (list 'zerop (cadr x)))
                    (list 'lessp
                          ''1
                          (times-tree (delete (cadr x)
                                              (times-fringe (caddr x))))))
            x)))
    x))

; the meta lemma to cancel identical times terms in lessp. For example,
; (lessp (times b (times c d)) (times b d)) =>
;         (and (and (not (zerop b)) (not (zerop d))) (lessp (fix c) 1))
(prove-lemma correctness-of-cancel-lessp-times ((meta lessp))
     (equal (eval$ t x a)
            (eval$ t (cancel-lessp-times x) a)))
```

```
(disable and-not-zerop-tree-delete)
(disable lessp-1-times-tree-delete)

(prove-lemma eval$-equal-times-tree-bagdiff (rewrite)
     (implies (and (eval$ t (and-not-zerop-tree x) a)
                   (subbagp x y)
                   (subbagp x z))
              (equal (equal (eval$ t (times-tree (bagdiff y x)) a)
                            (eval$ t (times-tree (bagdiff z x)) a))
                     (equal (eval$ t (times-tree y) a)
                            (eval$ t (times-tree z) a)))))

(prove-lemma zerop-makes-equal-true-bridge (rewrite)
     (implies (not (eval$ t (and-not-zerop-tree
                             (bagint (times-fringe (cons 'times x))
                                     (times-fringe (cons 'times y))))
                          a))
              (equal (equal (times (eval$ t (car x) a)
                                   (eval$ t (cadr x) a))
                            (times (eval$ t (car y) a)
                                   (eval$ t (cadr y) a)))
                     t))
     ((use (zerop-makes-times-tree-zero
            (x (bagint (times-fringe (cons 'times x))
                       (times-fringe (cons 'times y))))
            (y (times-fringe (cons 'times x))))
           (zerop-makes-times-tree-zero
            (x (bagint (times-fringe (cons 'times x))
                       (times-fringe (cons 'times y))))
            (y (times-fringe (cons 'times y)))))))

(defn equal-1-eval$-times-tree-delete-end (w x a)
  (equal (eval$ t (times-tree (delete w x)) a) 1))

(prove-lemma equal-1-times-tree-delete (rewrite)
     (implies (member w x)
              (equal (equal (eval$ t (times-tree (delete w x)) a) 1)
                     (if (zerop (eval$ t w a))
                         (equal-1-eval$-times-tree-delete-end w x a)
                       (equal (eval$ t (times-tree x) a)
                              (eval$ t w a))))))

(defn cancel-equal-times (x)
  (if (equal (car x) 'equal)
      (if (and (equal (caadr x) 'times)
               (equal (caaddr x) 'times))
          (if (listp (bagint (times-fringe (cadr x))
                             (times-fringe (caddr x))))
              (list 'if
                    (and-not-zerop-tree (bagint (times-fringe (cadr x))
                                                (times-fringe (caddr x))))
                    (list 'equal
                          (times-tree
                           (bagdiff (times-fringe (cadr x))
```

```
                                  (bagint (times-fringe (cadr x))
                                          (times-fringe (caddr x)))))
                        (times-tree
                         (bagdiff (times-fringe (caddr x))
                                  (bagint (times-fringe (cadr x))
                                          (times-fringe (caddr x))))))
                    '(true))
            x)
        (if (and (listp (cadr x)) (equal (caadr x) 'times)
                 (member (caddr x) (times-fringe (cadr x))))
            (list 'and
                  (list 'numberp (caddr x))
                  (list 'or
                        (list 'equal (caddr x) ''0)
                        (list 'equal
                              (times-tree (delete (caddr x)
                                                  (times-fringe (cadr x))))
                              ''1)))
          (if (and (listp (caddr x)) (equal (caaddr x) 'times)
                   (member (cadr x) (times-fringe (caddr x))))
              (list 'and
                    (list 'numberp (cadr x))
                    (list 'or
                          (list 'equal (cadr x) ''0)
                          (list 'equal
                                (times-tree (delete (cadr x)
                                                    (times-fringe (caddr x))))
                                ''1)))
            x)))
    x))

; the meta lemma to cancel identical times term in equality.  For example,
;     (equal (times b (times c d)) (times b d)) =>
;          (or (or (zerop b) (zerop d)) (equal (fix c) 1))
(prove-lemma correctness-of-cancel-equal-times ((meta equal))
     (equal (eval$ t x a)
            (eval$ t (cancel-equal-times x) a)))

(compile-uncompiled-defns "tmp")

; lemmas about exp.
(defn exp (x y)
  (if (zerop y)
      1
    (times x (exp x (sub1 y)))))

(prove-lemma exp-of-0 (rewrite)
     (equal (exp 0 k) (if (zerop k) 1 0)))

(prove-lemma exp-of-1 (rewrite)
     (equal (exp 1 k) 1))

(prove-lemma exp-plus (rewrite)
     (equal (times (exp x y) (exp x z))
```

```
           (exp x (plus y z))))

(prove-lemma exp-times (rewrite)
     (equal (exp (times x y) z)
            (times (exp x z) (exp y z))))

(prove-lemma exp-exp (rewrite)
     (equal (exp (exp x y) z)
            (exp x (times y z))))

(prove-lemma exp-of-2-0 (rewrite)
     (implies (not (zerop m))
              (not (lessp (exp m n) 1))))

(prove-lemma exp-of-2-1 (rewrite)
     (equal (lessp 1 (exp 2 n))
            (not (zerop n))))

(prove-lemma exp-lessp (rewrite)
     (equal (lessp (exp x y) (exp x z))
            (if (zerop x)
                (and (not (zerop y)) (zerop z))
              (if (equal x 1)
                  f
                (lessp y z)))))

(disable times)

(prove-lemma times-exp2-lessp (rewrite)
     (equal (lessp (times i (exp 2 j)) (exp 2 k))
            (lessp i (exp 2 (difference k j)))))

; lemmas about remainder and quotient.
(prove-lemma remainder-exit (rewrite)
     (implies (lessp i j)
              (equal (remainder i j) (fix i))))

(prove-lemma quotient-exit (rewrite)
     (implies (lessp i j)
              (equal (quotient i j) 0)))

(prove-lemma remainder-0 (rewrite)
     (and (equal (remainder 0 x) 0)
          (equal (remainder x 0) (fix x))))

(prove-lemma quotient-0 (rewrite)
     (and (equal (quotient 0 x) 0)
          (equal (quotient x 0) 0)))

(prove-lemma remainder-1 (rewrite)
     (and (equal (remainder 1 x) (if (equal x 1) 0 1))
          (equal (remainder x 1) 0))
        ((expand (remainder 1 x))))
```

```
(prove-lemma quotient-1 (rewrite)
     (and (equal (quotient 1 x) (if (equal x 1) 1 0))
          (equal (quotient m 1) (fix m)))
     ((expand (quotient 1 x))))

(prove-lemma remainder-x-x (rewrite)
     (equal (remainder x x) 0))

(prove-lemma quotient-x-x (rewrite)
     (equal (quotient x x) (if (zerop x) 0 1)))

(prove-lemma quotient-equal-0 (rewrite)
     (equal (equal (quotient m n) 0)
            (or (zerop m) (zerop n) (lessp m n))))

(prove-lemma remainder-2x (rewrite)
     (equal (remainder (plus x x) 2) 0))

(prove-lemma quotient-2x (rewrite)
     (equal (quotient (plus x x) 2) (fix x)))

(prove-lemma remainder-2x-add1 (rewrite)
     (equal (remainder (plus x (add1 x)) 2) 1))

(prove-lemma quotient-2x-add1 (rewrite)
     (equal (quotient (plus x (add1 x)) 2) (fix x)))

; A generalization lemma about quotient.
(prove-lemma quotient-generalize (generalize)
     (equal (equal (quotient m n) 0)
            (if (or (zerop m) (zerop n))
                t
              (lessp m n))))

(disable quotient-generalize)

(prove-lemma remainder-lessp (rewrite generalize)
     (equal (lessp (remainder x y) y)
            (not (zerop y))))

(prove-lemma quotient-lessp (rewrite)
     (equal (lessp (quotient m n) m)
            (and (not (zerop m))
                 (or (zerop n) (not (equal n 1))))))

(prove-lemma remainder-lessp-linear (rewrite)
     (implies (not (zerop y))
              (lessp (remainder x y) y)))

(prove-lemma quotient-lessp-linear (rewrite)
     (implies (and (not (zerop x)) (lessp 1 y))
              (lessp (quotient x y) x)))

(prove-lemma quotient-leq (rewrite)
```

```
     (not (lessp i (quotient i j))))

(prove-lemma remainder-wrt-2 (rewrite)
     (lessp (remainder n 2) 2))

(prove-lemma remainder-plus1 (rewrite)
     (implies (equal (remainder i j) 0)
              (equal (remainder (plus x i) j) (remainder x j))))

(prove-lemma remainder-plus2 (rewrite)
     (implies (equal (remainder i j) 0)
              (equal (remainder (plus i x) j) (remainder x j))))

(prove-lemma remainder-times (rewrite)
     (and (equal (remainder (times x y) y) 0)
          (equal (remainder (times y x) y) 0))
     ((induct (times x y))))

(prove-lemma remainder-plus-times1 (rewrite)
     (equal (remainder (plus x (times y z)) y)
            (remainder x y)))

(prove-lemma remainder-plus-times2 (rewrite)
     (equal (remainder (plus x (times z y)) y)
            (remainder x y)))

(prove-lemma remainder-plus-plus (rewrite)
     (implies (equal (remainder i j) 0)
              (equal (remainder (plus x y i) j)
                     (remainder (plus x y) j))))

(prove-lemma remainder-plus-add1 (rewrite)
     (implies (equal (remainder i j) 0)
              (equal (remainder (add1 (plus x i)) j)
                     (remainder (add1 x) j)))
     ((use (remainder-plus-plus (x 1) (y x)))))

(prove-lemma remainder-plus-difference1 (rewrite)
     (implies (and (not (lessp x z))
                   (equal (remainder y w) 0))
              (equal (remainder (difference (plus x y) z) w)
                     (remainder (difference x z) w))))

(prove-lemma remainder-plus-difference2 (rewrite)
     (implies (and (not (lessp y z))
                   (equal (remainder x w) 0))
              (equal (remainder (difference (plus x y) z) w)
                     (remainder (difference y z) w))))

(prove-lemma remainder-plus-plus-times1 (rewrite)
     (equal (remainder (plus x w (times y z)) y)
            (remainder (plus x w) y)))

(prove-lemma remainder-plus-plus-times2 (rewrite)
```

```
     (equal (remainder (plus x w (times z y)) y)
            (remainder (plus x w) y)))

(prove-lemma remainder-difference (rewrite)
     (implies (equal (remainder y z) 0)
              (equal (remainder (difference x y) z)
                     (if (lessp x y)
                         0
                       (remainder x z)))))

(prove-lemma remainder-difference-times1 (rewrite)
     (equal (remainder (difference x (times y z)) z)
            (if (lessp x (times y z))
                0
              (remainder x z))))

(prove-lemma remainder-difference-times2 (rewrite)
     (equal (remainder (difference x (times y z)) y)
            (if (lessp x (times y z))
                0
              (remainder x y))))

(prove-lemma quotient-plus1 (rewrite)
     (implies (equal (remainder i j) 0)
              (equal (quotient (plus x i) j)
                     (plus (quotient x j) (quotient i j)))))

(prove-lemma quotient-plus2 (rewrite)
     (implies (equal (remainder i j) 0)
              (equal (quotient (plus i x) j)
                     (plus (quotient i j) (quotient x j))))
     ((use (quotient-plus1))))

(prove-lemma quotient-times (rewrite)
     (and (equal (quotient (times x y) y)
                 (if (zerop y) 0 (fix x)))
          (equal (quotient (times y x) y)
                 (if (zerop y) 0 (fix x))))
     ((induct (times x y))))

(prove-lemma quotient-plus-times1 (rewrite)
     (equal (quotient (plus x (times y z)) y)
            (plus (quotient x y) (if (zerop y) 0 (fix z)))))

(prove-lemma quotient-plus-times2 (rewrite)
     (equal (quotient (plus x (times z y)) y)
            (plus (quotient x y) (if (zerop y) 0 (fix z)))))

(prove-lemma quotient-plus-plus (rewrite)
     (implies (equal (remainder i j) 0)
              (equal (quotient (plus x y i) j)
                     (plus (quotient (plus x y) j) (quotient i j)))))

(prove-lemma quotient-plus-add1 (rewrite)
```

```
     (implies (equal (remainder i j) 0)
              (equal (quotient (add1 (plus x i)) j)
                     (plus (quotient (add1 x) j) (quotient i j))))
     ((use (quotient-plus-plus (x 1) (y x)))))

(prove-lemma quotient-difference-plus1 (rewrite)
     (implies (and (not (lessp x z))
                   (equal (remainder y w) 0))
              (equal (quotient (difference (plus x y) z) w)
                     (plus (quotient (difference x z) w)
                           (quotient y w)))))

(prove-lemma quotient-difference-plus2 (rewrite)
     (implies (and (not (lessp y z))
                   (equal (remainder x w) 0))
              (equal (quotient (difference (plus x y) z) w)
                     (plus (quotient (difference y z) w)
                           (quotient x w))))
     ((use (quotient-difference-plus1 (x y) (y x)))))

(prove-lemma quotient-difference (rewrite)
     (implies (equal (remainder y z) 0)
              (equal (quotient (difference x y) z)
                     (if (lessp x y)
                         0
                       (difference (quotient x z) (quotient y z))))))

(prove-lemma quotient-difference-times1 (rewrite)
     (equal (quotient (difference x (times y z)) z)
            (if (lessp x (times y z))
                0
              (difference (quotient x z) (fix y)))))

(prove-lemma quotient-difference-times2 (rewrite)
     (equal (quotient (difference x (times y z)) y)
            (if (lessp x (times y z))
                0
              (difference (quotient x y) (fix z)))))

(disable remainder-difference)
(disable quotient-difference)

(prove-lemma remainder-sub1 (rewrite)
     (equal (remainder (sub1 m) n)
            (if (zerop n)
                (sub1 m)
              (if (equal (remainder m n) 0)
                  (if (zerop m) 0 (sub1 n))
                (sub1 (remainder m n))))))

(prove-lemma quotient-sub1 (rewrite)
     (equal (quotient (sub1 m) n)
            (if (equal (remainder m n) 0)
                (sub1 (quotient m n))
```

```
             (quotient m n))))

(prove-lemma remainder-quotient (rewrite)
     (equal (plus (times y (quotient x y))
                  (remainder x y))
            (fix x)))

(prove-lemma remainder-quotient-elim (elim)
     (implies (and (not (zerop y)) (numberp x))
              (equal (plus (remainder x y)
                           (times y (quotient x y)))
                     x)))

(prove-lemma remainder-add1 (rewrite)
     (equal (remainder (add1 m) n)
            (if (zerop n)
                (add1 m)
              (if (equal (remainder m n) (sub1 n))
                  0
                (add1 (remainder m n))))))

(prove-lemma quotient-add1 (rewrite)
     (equal (quotient (add1 m) n)
            (if (zerop n)
                0
              (if (equal (remainder m n) (sub1 n))
                  (add1 (quotient m n))
                (quotient m n)))))

(prove-lemma times-plus-lessp (rewrite)
     (implies (lessp x d)
              (equal (lessp (plus x (times b d)) (times c d))
                     (lessp b c)))
     ((enable times)))

(prove-lemma quotient-shrink-fast (rewrite)
     (not (lessp x (times y (quotient x y)))))

(prove-lemma remainder-plus-remainder1 (rewrite)
     (equal (remainder (plus x (remainder y z)) z)
            (remainder (plus x y) z)))

(prove-lemma remainder-difference-remainder1 (rewrite)
     (implies (if (lessp x y) f t)
              (equal (remainder (difference x (remainder y z)) z)
                     (remainder (difference x y) z))))

(prove-lemma remainder-plus-remainder2 (rewrite)
     (equal (remainder (plus x y (remainder z k)) k)
            (remainder (plus x y z) k)))

(prove-lemma remainder-plus-remainder (rewrite)
     (equal (remainder (plus (remainder x z) (remainder y z)) z)
            (remainder (plus x y) z)))
```

```
(prove-lemma remainder-crock (rewrite)
     (implies (lessp y z)
              (equal (remainder (times y x) (times x z))
                     (times y x)))
     ((use (remainder-exit (i (times y x)) (j (times x z))))))

(prove-lemma times-distributes-remainder (rewrite)
     (equal (remainder (times x y) (times x z))
            (times x (remainder y z))))

(prove-lemma quotient-crock (rewrite)
     (implies (lessp y z)
              (equal (quotient (times y x) (times x z)) 0))
     ((use (quotient-exit (i (times x y)) (j (times x z))))))

(prove-lemma quotient-times-cancel (rewrite)
     (equal (quotient (times x y) (times x z))
            (if (zerop x) 0 (quotient y z))))

(disable remainder-crock)
(disable quotient-crock)

(prove-lemma remainder-distributes-times2-add1 (rewrite)
     (equal (remainder (add1 (times 2 y)) (times 2 z))
            (add1 (times 2 (remainder y z)))))

(prove-lemma quotient-distributes-times2-add1 (rewrite)
     (equal (quotient (add1 (times 2 y)) (times 2 z))
            (quotient y z)))

(prove-lemma quotient-exp (rewrite)
     (implies (lessp 1 i)
              (equal (quotient (exp i j) (exp i k))
                     (if (lessp j k) 0 (exp i (difference j k))))))

(prove-lemma remainder-exp (rewrite)
     (implies (lessp 1 i)
              (equal (remainder (exp i j) (exp i k))
                     (if (lessp j k) (exp i j) 0))))

(prove-lemma remainder-plus-cancel0 (rewrite)
     (equal (equal (remainder (plus i j) n) i)
            (if (zerop n)
                (and (numberp i) (zerop j))
              (if (lessp i n)
                  (and (numberp i)
                       (equal (remainder j n) 0))
                f)))
     ((disable remainder-lessp-linear)))

(prove-lemma remainder-plus-cancel (rewrite)
      (equal (equal (remainder (plus i j) n) (remainder (plus i k) n))
             (equal (remainder j n) (remainder k n)))
```

```
      ((induct (plus i j))))

(prove-lemma quotient-times-lessp (rewrite)
     (equal (lessp (quotient x z) y)
            (if (zerop z)
                (not (zerop y))
              (lessp x (times z y)))))

(prove-lemma quotient-quotient (rewrite)
     (equal (quotient (quotient x z) y)
            (quotient x (times z y))))

; we redefine the distribution law of times and plus, and disable the old one.
(prove-lemma times-distributes-plus-new (rewrite)
     (equal (plus (times x y) (times x z))
            (times x (plus y z))))

(disable times-distributes-plus)

; an induction hint for the next event.
(defn quot2-sub12-induct (x y i j)
  (if (zerop i)
      t
    (quot2-sub12-induct (quotient x 2) (quotient y 2) (sub1 i) (sub1 j))))

(prove-lemma lessp-plus-times-exp2 (rewrite)
     (implies (lessp x (exp 2 i))
              (equal (lessp (plus x (times y (exp 2 i)))
                            (exp 2 n))
                     (if (zerop y)
                         (lessp x (exp 2 n))
                       (lessp y (exp 2 (difference n i))))))
     ((induct (quot2-sub12-induct x y* i n))))

(prove-lemma lessp-plus-exp2 (rewrite)
     (implies (lessp x (exp 2 i))
              (equal (lessp (plus x (exp 2 i)) (exp 2 n))
                     (lessp i n)))
     ((use (lessp-plus-times-exp2 (y 1)))))

(prove-lemma remainder-times-exp2-1 (rewrite)
     (equal (remainder (times x (exp 2 i)) (exp 2 j))
            (times (remainder x (exp 2 (difference j i)))
                   (exp 2 i))))

(prove-lemma remainder-times-exp2-2 (rewrite)
     (equal (remainder (times (exp 2 i) x) (exp 2 j))
            (times (remainder x (exp 2 (difference j i)))
                   (exp 2 i)))
     ((use (remainder-times-exp2-1))))

; special cases of remainder-times-exp2.
(prove-lemma remainder-times-exp2-3 (rewrite)
     (and (equal (remainder (times x (exp 2 i)) 2)
```

```
               (if (zerop i) (remainder x 2) 0))
          (equal (remainder (times (exp 2 i) x) 2)
               (if (zerop i) (remainder x 2) 0)))
     ((use (remainder-times-exp2-1 (j 1)))))

(prove-lemma remainder-times-exp2-4 (rewrite)
     (and (equal (remainder (times x (exp 2 i) y) 2)
               (if (zerop i) (remainder (times x y) 2) 0))
          (equal (remainder (times x y (exp 2 i)) 2)
               (if (zerop i) (remainder (times x y) 2) 0)))
     ((use (remainder-times-exp2-3 (x (times x y))))))

(prove-lemma quotient-times-exp2-1 (rewrite)
     (equal (quotient (times x (exp 2 i)) (exp 2 j))
            (if (lessp i j)
                (quotient x (exp 2 (difference j i)))
              (times x (exp 2 (difference i j))))))

(prove-lemma quotient-times-exp2-2 (rewrite)
     (equal (quotient (times (exp 2 i) x) (exp 2 j))
            (if (lessp i j)
                (quotient x (exp 2 (difference j i)))
              (times x (exp 2 (difference i j)))))
     ((use (quotient-times-exp2-1))))

(prove-lemma quotient-times-exp2-3 (rewrite)
     (and (equal (quotient (times x (exp 2 i)) 2)
               (if (zerop i) (quotient x 2) (times x (exp 2 (sub1 i)))))
          (equal (quotient (times (exp 2 i) x) 2)
               (if (zerop i) (quotient x 2) (times x (exp 2 (sub1 i))))))
     ((use (quotient-times-exp2-1 (j 1)))))

(prove-lemma quotient-times-exp2-4 (rewrite)
     (and (equal (quotient (times x (exp 2 i) y) 2)
               (if (zerop i)
                   (quotient (times x y) 2)
                 (times x y (exp 2 (sub1 i)))))
          (equal (quotient (times x y (exp 2 i)) 2)
               (if (zerop i)
                   (quotient (times x y) 2)
                 (times x y (exp 2 (sub1 i))))))
     ((use (quotient-times-exp2-3 (x (times x y))))))

(prove-lemma remainder-remainder-exp2 (rewrite)
     (equal (remainder (remainder x (exp 2 i)) (exp 2 j))
            (if (lessp i j)
                (remainder x (exp 2 i))
              (remainder x (exp 2 j)))))

; a lemma for the event add-evenp.
(prove-lemma remainder-remainder-2 (rewrite)
     (equal (remainder (remainder x (exp 2 i)) 2)
            (if (zerop i) 0 (remainder x 2)))
     ((use (remainder-remainder-exp2 (j 1)))))
```

```
; logarithm is not used in the specification or the lemma library.  It
; comes in when we'd like to reason about the time complexity of programs.
(prove-lemma times-lessp (rewrite)
     (implies (leq x z)
              (equal (lessp x (times y z))
                     (if (or (zerop y) (zerop z))
                         f
                       (if (equal y 1) (lessp x z) t))))
     ((enable times)))

(defn log (b x)
  (if (or (zerop b) (equal b 1))
      0
    (if (lessp x b)
        0
      (add1 (log b (quotient x b))))))

(prove-lemma log-of-0 (rewrite)
     (equal (log b 0) 0))

(prove-lemma log-of-1 (rewrite)
     (implies (lessp 1 b)
              (equal (log b 1) 0)))

(prove-lemma log-equal-0 (rewrite)
     (equal (equal (log b x) 0)
            (or (zerop b) (equal b 1) (lessp x b))))

(prove-lemma log-exp (rewrite)
     (implies (lessp 1 b)
              (equal (log b (exp b n)) (fix n))))

(prove-lemma log-times-exp (rewrite)
     (implies (and (lessp 1 b)
                   (not (zerop x)))
              (and (equal (log b (times x (exp b n)))
                          (plus n (log b x)))
                   (equal (log b (times (exp b n) x))
                          (plus n (log b x))))))

(prove-lemma log-times-exp-1 (rewrite)
     (implies (and (lessp 1 b)
                   (not (zerop x)))
              (and (equal (log b (times x b)) (add1 (log b x)))
                   (equal (log b (times b x)) (add1 (log b x))))))

(prove-lemma log-quotient-exp (rewrite)
     (implies (lessp 1 b)
              (equal (log b (quotient x (exp b i)))
                     (difference (log b x) i))))

(defn quotient2-induct (b x y)
  (if (or (zerop b) (equal b 1))
```

```
      0
    (if (or (zerop x) (zerop y))
        0
      (quotient2-induct b (quotient x b) (quotient y b)))))

(prove-lemma log-leq (rewrite)
     (implies (leq x y)
              (not (lessp (log b y) (log b x))))
     ((induct (quotient2-induct b x y))))

; these two lemmas are useful in time analysis.
(prove-lemma ta-lemma-1 ()
     (implies (leq a a1)
              (leq (plus x (times y (log 2 a)))
                   (plus x (times y (log 2 a1))))))

(prove-lemma ta-lemma-2 ()
     (implies (and (leq a a1)
                   (leq b b1))
              (leq (plus x (times y (plus (log 2 a) (log 2 b))))
                   (plus x (times y (plus (log 2 a1) (log 2 b1)))))))

(make-lib "mc20-0")
```

B.2 A Formal Model of Some MC68020 User-Mode Instructions

This section contains a formal specification of approximately 80% of the user-mode instructions of the Motorola MC68020 microprocessor. The definitions below were written in the logic described in *A Computational Logic Handbook,* [9], with syntactic extensions for 'let' and 'cond'. The definitions have been admitted under the definitional principle described in that book, using the mechanical theorem prover also described in that book. Boyer and Yu [12] presents this MC68020 model in a conventional mathematical syntax. Boyer and Yu [11] describes how we have used this specification to prove mechanically the correctness of several dozen machine code programs, most of them generated by 'industrial strength' compilers for C or Ada. Our specification is based upon the user's manual for the MC68020 microprocessor [41].

```
#|
%--------------------------------------------------------------------------
% Date:      Jan, 1991
% Modified:  December 15, 1992.
% File:      mc20-1.events
%--------------------------------------------------------------------------

Abstract. We present a formal specification of approximately 80% of
```

```
the ‘user mode’ instructions of the Motorola MC68020 microprocessor.
The specification is given in the form of definitions in the logic of
Nqthm, the Boyer-Moore system.  The specification has been used in the
mechanical verification of several dozen machine code programs, whose
binary was generated by ‘industrial strength’ C and Ada compilers.

                      Introduction

The function definitions below are ordered so that a function is
defined before it is referenced by another function.  One of the very
last functions defined, stepn emulates the MC68020.  Like all the
functions in this specification, stepn is a recursive and hence
computable function.  Approximately speaking, if we are given an
MC68020 state s and a positive integer n, we can compute the state s'
that results from executing an MC68020 for n instructions, starting in
state s, by applying stepn to s and n.  If an illegal instruction or
an instruction not among those covered in this specification is
encountered during execution, then s' will exhibit an indication of
the error.  If no such error indication is exhibited, then the
returned state correctly represents the state that a ‘real’ MC68020
would have after running n instructions provided that (i) the caches
are initially consistent with memory, (ii) no interrupts happen during
execution, and, of course, (iii) no externally caused changes to the
state occur during execution.  In Section example is a theorem that
illustrates the use of stepn to emulate an MC68020 on a specific
state, one that contains machine code for Euclid's GCD algorithm.

Disclaimer: The development of this formal specification is part
of a small scientific project aimed at examining the feasibility of
mechanically checking the correctness of machine code programs that
run on widely-used microprocessors.  The accuracy with which the
specification presented here represents a ‘real’ MC68020 is something
we do not know how to ascertain with the certainty of a mathematical
proof.  One can only become increasingly confident by such activities
as critical reading, testing, and bug fixing.  It is in a spirit of
scientific cooperation that we distribute this specification, but we
distribute it without any warranty of any kind, on an ‘as is’ basis.
|#

;                       Start Up
(note-lib "mc20-0")

;                     Some Constants
; In the MC68020, a ``byte'' is 8 bits long.  A ``word'' is 16 bits long.
; A ``long word'' is 32 bits long.  A ``quad word'' is 64 bits long.

(defn b () 8)
(defn w () 16)
(defn l () 32)
(defn q () 64)

(defn bsz () 1)
(defn wsz () 2)
(defn lsz () 4)
```

```
(defn qsz () 8)

; Some error signals.
(defn read-signal () 'read_unavailable_memory)

(defn write-signal () 'write_rom_or_unavailable_memory)

(defn reserved-signal () 'motorola_reserved_for_future_development)

(defn pc-signal () 'pc_outside_rom)

(defn pc-odd-signal () 'pc_at_odd_address)

(defn mode-signal () 'illegal_addressing_mode_in_current_instruction)

; Throughout our specification, we have frequent need to refer to bits and
; bit-vectors. In our model, bit ::= 0 | 1, and bit-vectors ::= nonnegative
; integers.  If the operation is signed, we use the two conversion functions
; nat-to-int and int-to-nat.

; bitp is a function of one argument, x.  bitp returns t or f according to
; whether x is a bit or not.
(defn bitp (x)
  (or (equal x 0) (equal x 1)))

; We frequently use the bits 0 and 1.  For clarity, to identify informally
; when we are using these integers as bits, we use the two constants b1
; and b0.

(defn b1 () 1)

(defn b0 () 0)

; We frequently test a bit to see whether it is 0 or 1.  We define
; the functions b1p and b0p to return t or f according to whether
; their arguments are 0 or non-0 respectively.
(defn b0p (x)
  (equal x (b0)))

(defn b1p (x)
  (not (equal x (b0))))

(defn fix-bit (c)
  (if (b0p c) (b0) (b1)))

; Here are the definitions of some operators for logical arithmetic on bits.

; b-not returns the complement of its argument.
(defn b-not (x)
  (if (b0p x) (b1) (b0)))

; b-and returns the logical and of its two arguments.
(defn b-and (x y)
  (if (b0p x)
```

```
      (b0)
    (if (b0p y) (b0) (b1))))

; b-or returns the logical or of its two arguments.
(defn b-or (x y)
  (if (b0p x)
      (if (b0p y) (b0) (b1))
    (b1)))

; b-nor returns the logical nor of its two arguments.
(defn b-nor (x y)
  (if (b0p x)
      (if (b0p y) (b1) (b0))
    (b0)))

; b-nand returns the logical nand of its two arguments.
(defn b-nand (x y)
  (if (b0p x)
      (b1)
    (if (b0p y) (b1) (b0))))

; b-eor returns the exclusive or of its two arguments.
(defn b-eor (x y)
  (if (b0p x)
      (if (b0p y) (b0) (b1))
    (if (b0p y) (b1) (b0))))

; b-equal returns the logical equal of its two arguments.
(defn b-equal (x y)
  (if (b0p x) (b0p y) (b1p y)))

;            Bit Vector Arithmetic
; bcar returns the first bit of x.
(defn bcar (x)
  (remainder x 2))

; bcdr returns a natural number by cutting off the first bit of x.
; For any natural number x, (equal (plus (bcar x) (times x (bcdr x))) x).

(defn bcdr (x)
  (quotient x 2))

; head is a function of two arguments, x and n.  x and n should be
; nonnegative integers.  head returns the remainder of x divided by
; 2^n.
(defn head (x n)
  (remainder x (exp 2 n)))

; tail is a function of two arguments, x and n.  x and n should be
; nonnegative integers.  tail returns the quotient of x divided by
; 2^n.
(defn tail (x n)
  (quotient x (exp 2 n)))
```

```
; We next define some logical operations on bit-vectors.  lognot takes
; two naturals as its arguments and returns the logical complement of its
; second argument.
(defn lognot (n x)
  (sub1 (difference (exp 2 n) (head x n))))

; logand takes two naturals as arguments and returns their logical and.
(defn logand (x y)
  (if (or (zerop x) (zerop y))
      0
    (plus (b-and (bcar x) (bcar y))
          (times 2 (logand (bcdr x) (bcdr y))))))

; logor takes two naturals as arguments and returns the logical
; (inclusive) or of the two arguments.
(defn logor (x y)
  (if (zerop x)
      (fix y)
    (if (zerop y)
        (fix x)
      (plus (b-or (bcar x) (bcar y))
            (times 2 (logor (bcdr x) (bcdr y)))))))

; logeor takes two naturals as arguments and returns the logical
; exclusive or of the two arguments.
(defn logeor (x y)
  (if (and (zerop x) (zerop y))
      0
    (plus (b-eor (bcar x) (bcar y))
          (times 2 (logeor (bcdr x) (bcdr y))))))
  ((lessp (plus x y))))

; bitn retrieves the nth bit of x.  Indexing is 0-based.
(defn bitn (x n)
  (bcar (tail x n)))

; mbit returns the most significant bit of x, assuming that x is a
; bit vector of n bits.
(defn mbit (x n)
  (bitn x (sub1 n)))

; bits returns bits i through j as a natural number.  bits is a function
; of three arguments, x, i, and j.  x, i, and j should be natural numbers.
; Intuitively, bits extracts bits of x from bit i to bit j.  Normally, i
; should be less than or equal to j.
(defn bits (x i j)
  (head (tail x i) (add1 (difference j i))))

; setn updates the nth bit of x by the given value c.  Indexing is 0-based.
(defn setn (x n c)
  (if (zerop n)
      (plus (fix-bit c) (times 2 (bcdr x)))
    (plus (bcar x) (times 2 (setn (bcdr x) (sub1 n) c)))))
```

```
; adder takes four arguments and returns the addition of x, y, and c modulo
; 2^n.  That is,  (remainder (plus x y c) (exp n 2)).  Typically,  c is
; either 0 or 1.
(defn adder (n c x y)
  (head (plus c x y) n))

; add takes three arguments and returns the addition of x and y modulo
; 2^n.  That is, (remainder (plus x y) (exp n 2)).
(defn add (n x y)
  (head (plus x y) n))

; subtractor takes four arguments and returns the subtraction of y and
; (remainder (plus x c) (exp n 2)). i.e.,
;   (remainder (difference y (plus x c)) (exp n 2)).
; Typically,  c is either 0 or 1.
(defn subtractor (n c x y)
  (adder n (b-not c) y (lognot n x)))

; sub takes three arguments and returns, in the form of 2's complement,
; the subtraction of y and x.  i.e., (remainder (difference y  x) (exp n 2)).
(defn sub (n x y)
  (head (plus y (difference (exp 2 n) (head x n)))
        n))

; app ``appends'' two naturals.  app takes three arguments, n, x, and y.
(defn app (n x y)
  (plus (head x n)
        (times y (exp 2 n))))

; replace replaces x partially by y in the head. replace is a function
; of three arguments, n, x and y, all of which should be naturals. replace
; is frequently used when updating only one byte or one word in a register,
; leaving the other bytes alone.
(defn replace (n x y)
  (app n x (tail y n)))

; ext is a function of three arguments, n, x and size.  ext is used
; frequently to do ``sign-extension''.  For instance, in the MC68020,
; we often extract a byte or word and wish to add it into a 32-bit sum,
; but we first sign-extend the extracted quantity to obtain a meaningful
; sum.
(defn ext (n x size)
  (if (lessp n size)
      (if (b0p (bitn x (sub1 n)))
          (head x n)
        (app n x (sub1 (exp 2 (difference size n)))))
    (head x size)))

; Shift operations.

; Logical shift left.
(defn lsl (len x cnt)
  (head (times x (exp 2 cnt)) len))
```

```
; Arithmetic shift left.
(defn asl (len x cnt)
  (head (times x (exp 2 cnt)) len))

; Logical shift right.
(defn lsr (x cnt)
  (tail x cnt))

; Arithmetic shift right.
(defn asr (n x cnt)
  (if (lessp x (exp 2 (sub1 n)))
      (tail x cnt)
    (if (lessp n cnt)
        (sub1 (exp 2 n))
      (app (difference n cnt) (tail x cnt) (sub1 (exp 2 cnt))))))

;                Integer Arithmetic
; Throughout most of this MC68020 specification, we restrict our attention to
; arithmetic on the nonnegative integers.  However, in the definition of two
; machine instructions, those for signed multiplication and division, we also
; consider all of the integers, both nonnegative and negative.  The Nqthm
; logic adds the negative integers almost as an afterthought, and the basic,
; built-in arithmetic operations of the Nqthm logic work only for nonnegative
; integers.  To do arithmetic on all the integers, we must define appropriate
; operations explicitly, as we do below.
;
; The Nqthm logic has the peculiarity that (minus 0) is not the same as 0 .
; However, we will restrict our domain so that (minus 0) is not considered.
; A negative integer is defined to be of the form (minus x) with x nonzero.
(defn negp (i)
  (and (negativep i)
       (not (equal i (minus 0)))))

; x is an integer iff x is either a nonnegative number or a negative number.
(defn integerp (x)
  (or (numberp x)
      (negp x)))

(defn fix-int (x)
  (if (integerp x) x 0))

(defn izerop (x)
  (equal (fix-int x) 0))

(defn abs (x)
  (if (negp x)
      (negative-guts x)
    (fix x)))

(defn ilessp (i j)
  (if (negp i)
      (if (negp j)
          (lessp (negative-guts j) (negative-guts i))
        t)
```

```
    (if (negp j)
        f
      (lessp i j))))

(defn ileq (i j)
  (not (ilessp j i)))

(defn iplus (x y)
  (if (negp x)
      (if (negp y)
          (minus (plus (negative-guts x)
                       (negative-guts y)))
        (if (lessp y (negative-guts x))
            (minus (difference (negative-guts x) y))
          (difference y (negative-guts x))))
    (if (negp y)
        (if (lessp x (negative-guts y))
            (minus (difference (negative-guts y) x))
          (difference x (negative-guts y)))
      (plus x y))))

(defn ineg (x)
  (if (izerop x)
      0
    (if (negp x)
        (negative-guts x)
      (minus x))))

(defn idifference (x y)
  (iplus x (ineg y)))

(defn itimes (x y)
  (if (negp x)
      (if (negp y)
          (times (negative-guts x) (negative-guts y))
        (fix-int (minus (times (negative-guts x) y))))
    (if (negp y)
        (fix-int (minus (times x (negative-guts y))))
      (times x y))))

(defn iremainder (x y)
  (if (negp x)
      (fix-int (minus (remainder (negative-guts x) (abs y))))
    (remainder x (abs y))))

(defn iquotient (x y)
  (if (negp x)
      (if (negp y)
          (quotient (negative-guts x) (negative-guts y))
        (fix-int (minus (quotient (negative-guts x) y))))
    (if (negp y)
        (fix-int (minus (quotient x (negative-guts y))))
      (quotient x y))))
```

```
; The size of bit vectors.
; nat-rangep returns T, if (lessp nat (exp n 2)), but returns F, otherwise.
(defn nat-rangep (nat n)
  (lessp nat (exp 2 n)))

; The size of an unsigned integer.
; uint-rangep returns T, if (leq 0 x (exp n 2)), and returns F, otherwise.
(defn uint-rangep (x n)
  (lessp x (exp 2 n)))

; Two conversion functions for unsigned integer interpretation.
(defn nat-to-uint (x) (fix x))

(defn uint-to-nat (x) (fix x))

; The size of an integer.
; int-rangep returns T, if -2^(n-1) <= int < 2^(n-1), and returns F, otherwise.
(defn int-rangep (int n)
  (if (zerop n)
      (equal (fix-int int) 0)
    (if (negativep int)
        (leq (negative-guts int) (exp 2 (sub1 n)))
      (lessp int (exp 2 (sub1 n))))))

; Two conversion functions for signed integer interpretation. nat-to-int
; converts natural numbers to integers, int-to-nat converts integers to
; natural numbers.
(defn nat-to-int (x n)
  (if (lessp x (exp 2 (sub1 n)))
      (fix x)
    (minus (difference (exp 2 n) x))))

(defn int-to-nat (x size)
  (if (negativep x)
      (difference (exp 2 size)
                  (negative-guts x))
    (fix x)))

;              Binary Trees for Memory
; A binary tree is either nil or an object of the form (value bt0 . bt1),
; where bt0 and bt1 are binary trees and value is any object stored at
; that node.

; value-field is a function of one argument.  value-field returns the
; object stored at the current node, i.e., the car.
(defn value-field (bt)
  (if (listp bt) (car bt) 0))

; branch0 is a function of one argument, which should be a non-nil binary
; tree. branch0 returns the left branch, i.e., the cadr.
(defn branch0 (bt)
  (if (listp bt) (cadr bt) nil))

; branch1 is a function of one argument, which should be a non-nil bin-tree.
```

```
; branch1 returns the right branch, i.e., the cddr.
(defn branch1 (bt)
  (if (and (listp bt) (listp (cdr bt)))
      (cddr bt)
    nil))

; Construct a binary tree (value br0 . br1).
(defn make-bt (value br0 br1)
  (cons value (cons br0 br1)))

; In order to execute MC68020 instructions reasonably efficiently in an
; applicative programming language, we implement memory using binary trees
; rather than simple linear lists or association lists.  Binary trees give us
; logarithmic access and change times.

; A memory state in this specification is actually given by a cons of two
; binary trees, one that tells us 'protection' information about each byte of
; the memory and one that is the 'physical' memory, i.e., the byte of data
; stored at each 32-bit address.

; A completely 'full' binary tree would contain 2^32 tips, and the explicit
; representation of such a tree would vastly exceed the memory capacity of any
; known implementation of Nqthm.  Therefore, we assign meaning to non-full,
; i.e., partially full, binary trees, both for protection and for data.

; To characterize, informally, the content and protection of an address in
; memory, let us momentarily view an address as a sequence of 32 bits, most
; significant bit on the left.  By an 'initial sequence' of an address x,
; we mean a sequence to which one can append another possibly empty sequence
; on the right to obtain x.  Thus 001 is an initial sequence of 0010011.
; For a given memory data tree bt and address x, what is the content of bt
; at x?  Answer: if the subtree of bt obtained by taking the path through
; bt determined by any initial sequence of x is nil, then the content of
; bt at x is 0.  Otherwise, the content is the value field at the subtree
; of bt determined by x.  In other words, if bt is not sufficiently
; deep along the path x, then the content of bt at x is 0.

; A memory protection tree map is a binary tree which has stored at each
; node, in the value cell, either nil, '(unavailable), '(rom), or
; '(unavailable rom).  (The last of these has the same meaning as
; '(unavailable).) For a given memory protection tree map and address x,
; what is the protection status of map at x?  Answer: if 'unavailable is
; a member of the value cell at any subtree of map obtained by taking the
; path through map determined by any initial subsequence of x, then the
; address x is said to be unavailable, and it may not be read or written
; (even as part of a word or long word operation) by any instruction.
; Moreover, if an address x is not unavailable by the preceding rule, but
; 'rom is a member of any such value cell, then the address is said to be
; ROM and may not be written by any instruction.  Instructions must come
; entirely from such ROM addresses.  Finally, if an address is not unavailable
; or ROM by the preceding rules, we say that it is RAM, and it may be read or
; written by any instruction.
;
```

```
; readp is a function of three arguments, x, map, and n.  map should
; be a memory protection binary tree.  x should be a natural number.  n is
; the index of the 'next bit' to select upon in x while walking the x path
; through map.  Typically readp is called with n initially equal to 32
; and map equal to the current memory protection map.  readp returns
; f if it encounters an 'unavailable at a node on the x path through
; map (considering only the least n significant bits of x), and otherwise
; readp returns t .
(defn readp (x n map)
  (if (member 'unavailable (value-field map))
      f
    (if (or (nlistp map) (zerop n))
        t
      (if (b0p (bitn x (sub1 n)))
          (readp x (sub1 n) (branch0 map))
        (readp x (sub1 n) (branch1 map)))))
  ((lessp (count n))))

; In our specification, programs can only be stored in ROM.  The function
; pc-readp returns t only when it hits a 'rom at a node on the path x
; through map and only if there is no 'unavailable at each node on the
; path x.  n serves the same role it does in readp, as an index into
; x.
(defn pc-readp (x n map)
  (if (member 'unavailable (value-field map))
      f
    (if (member 'rom (value-field map))
        (readp x n map)
      (if (or (nlistp map) (zerop n))
          f
        (if (b0p (bitn x (sub1 n)))
            (pc-readp x (sub1 n) (branch0 map))
          (pc-readp x (sub1 n) (branch1 map))))))
  ((lessp (count n))))

; writep is a function of three arguments, x, n, and map.  map
; should be a memory protection binary tree.  x should be a natural number.
; writep returns t if it never encounters 'unavailable or 'rom at a
; node on the path x through map, otherwise f .  n serves the same role
; it does in readp, as an index into x.
(defn writep (x n map)
  (if (or (member 'unavailable (value-field map))
          (member 'rom (value-field map)))
      f
    (if (or (nlistp map) (zerop n))
        t
      (if (b0p (bitn x (sub1 n)))
          (writep x (sub1 n) (branch0 map))
        (writep x (sub1 n) (branch1 map)))))
  ((lessp (count n))))

; read is a function of three arguments, x, n, and bt.  bt should be
; a binary tree, x and n should be natural numbers.  read returns the
; value component at the node reached by taking the path x through bt. n
```

```
; serves the same role it does in readp, as an index into x.

(defn read (x n bt)
  (if (zerop n)
      (value-field bt)
    (if (b0p (bitn x (sub1 n)))
        (read x (sub1 n) (branch0 bt))
      (read x (sub1 n) (branch1 bt)))))

; pc-read acts the same as read. But it is used in a  quite different
; sense.  So we introduce this dummy function.
(defn pc-read (x n bt)
  (read x n bt))

; write is a function of four arguments, value, x, n, and bt.
; value, x, and n should be nonnegative integers, and bt should be a
; binary tree.  write returns the binary tree obtained by updating bt at
; the address x.  n serves the same role it does in readp, as an index
; into x.

(defn write (value x n bt)
  (if (zerop n)
      (make-bt value (branch0 bt) (branch1 bt))
    (if (b0p (bitn x (sub1 n)))
        (make-bt (value-field bt)
                 (write value x (sub1 n) (branch0 bt))
                 (branch1 bt))
      (make-bt (value-field bt)
               (branch0 bt)
               (write value x (sub1 n) (branch1 bt))))))

; get-nth is a function of two arguments.  The first should be a
; nonnegative integer and the second should be a list.  get-nth
; returns the nth element of lst.  Indexing is 0-based.
; For example, (equal (get-nth 0 (list a b c)) a).
(defn get-nth (n lst)
  (if (zerop n)
      (car lst)
    (get-nth (sub1 n) (cdr lst))))

; put-nth is a function of three arguments: value, n, and lst.
; value and n should be natural numbers, and lst should be
; a list.  put-nth returns a list like lst except that the nth
; element has been changed to be value.  Indexing is 0-based,
; e.g., (equal (put-nth d 1 (list a b c)) (list a d c)).
(defn put-nth (value n lst)
  (if (zerop n)
      (cons value (cdr lst))
    (cons (car lst) (put-nth value (sub1 n) (cdr lst)))))

; The size of the operand, given the operation length.
(defn op-sz (oplen)
  (quotient oplen (b)))
```

```
; read-rn and write-rn are two functions used to fetch and modify
; the register rn in the register file regs.
(defn read-rn (oplen rn regs)
  (head (get-nth rn regs) oplen))

(defn write-rn (oplen value rn regs)
  (put-nth (replace oplen value (get-nth rn regs)) rn regs))

; A machine state is defined to be a list of length 5, say (status regs pc ccr
; mem), whose components have the following purposes: status, if it is not
; 'running, is the reason that execution was stopped; regs holds the data
; registers and the address registers; pc is the program counter; ccr is the
; 16-bit condition code register; and mem is the memory, including protection
; information. The status field is set when we encounter an instruction
; which we do not choose to handle for some reason.  Among the many reasons
; that might arise for setting the status field are (1) an illegal instruction,
; (2) a legal MC68020 instruction (e.g., CALLM) that this specification does
; not handle, and (3) an illegal addressing mode.  To construct a state one
; uses the 5 argument function mc-state, giving it as arguments, in order,
; the halt-reason, the data and address registers, the pc, the ccr, and
; the memory.  The five fields of a state can be accessed with the five
; accessor functions mc-status, mc-rfile, mc-pc, mc-ccr, and mc-mem.

(defn mc-state (status regs pc ccr mem)
  (list status regs pc ccr mem))

(defn mc-status (s) (car s))

(defn mc-rfile (s) (cadr s))

(defn mc-pc (s) (head (caddr s) (l)))

(defn mc-ccr (s) (head (cadddr s) (b)))

(defn mc-mem (s) (caddddr s))

; len is a function of one argument, lst, which should be a proper list.
; len returns the length of lst, i.e., the number of elements in lst.
(defn len (lst)
  (if (nlistp lst)
      0
    (add1 (len (cdr lst)))))

; mc-haltp returns T if some halting condition has been satisfied.
(defn mc-haltp (s)
  (not (equal (mc-status s) 'running)))

;               Operands from Memory
; Everything in this section is machine dependent.  We assume the memory
; capacity is 2^32.  In our specification, the memory is a binary tree
; with depth 32.

(defn byte-readp (x mem)
  (readp x 32 (car mem)))
```

```
; read-memp returns t if the k consecutive bytes in memory starting
; at x are readable, but returns f otherwise.
(defn read-memp (x mem k)
  (if (zerop k)
      t
    (and (byte-readp (add 32 x (sub1 k)) mem)
         (read-memp x mem (sub1 k)))))

; word-readp determines whether both bytes of the word at the memory
; address x are readable.
(defn word-readp (x mem)
  (read-memp x mem (wsz)))

; long-readp determines whether all four bytes of the longword at the
; memory address x are readable.
(defn long-readp (x mem)
  (read-memp x mem (lsz)))

; Programs can only be stored in ROM.  Assume that x is a pointer
; in some program segment.  pc-read-memp returns t if the next k
; consecutive bytes are ROM.
(defn pc-byte-readp (x mem)
  (pc-readp x 32 (car mem)))

(defn pc-read-memp (x mem k)
  (if (zerop k)
      t
    (and (pc-byte-readp (add 32 x (sub1 k)) mem)
         (pc-read-memp x mem (sub1 k)))))

(defn pc-word-readp (x mem)
  (pc-read-memp x mem (wsz)))

(defn pc-long-readp (x mem)
  (pc-read-memp x mem (lsz)))

; Read from the memory.
; byte-read reads a byte from the memory.
(defn byte-read (x mem)
  (head (read x 32 (cdr mem)) (b)))

; Read k consecutive bytes from the memory at x to form a natural number.
; read-mem is a function of three arguments, x, mem, and k.
; read-mem
; returns the natural number obtained by 'appending' together the n bytes
; that are obtained by reading from mem at locations addr, ..., addr+n-1.
; The most significant byte is the one with the lowest memory address, and
; conversely, the least significant byte is the one with the highest memory
; address.  This is known as the 'Big Endian' scheme of memory.
;
(defn read-mem (x mem k)
   (if (zerop k)
       0
```

```
     (app (b)
          (byte-read (add 32 x (sub1 k)) mem)
          (read-mem x mem (sub1 k)))))

; The two functions word-read and long-read use the function
; read-mem to obtain a word or a long word from the memory.
(defn word-read (x mem)
  (read-mem x mem (wsz)))

(defn long-read (x mem)
  (read-mem x mem (lsz)))

; Fetch instructions, by fetching bytes pointed to by the pc.  This is
; the same as reading from memory.  But we define a separate set of functions
; because we use them in a very different sense in our specification.
; pc-byte-read reads a byte from the memory at pc.
(defn pc-byte-read (pc mem)
  (head (pc-read pc 32 (cdr mem)) (b)))

(defn pc-read-mem (pc mem k)
  (if (zerop k)
      0
    (app (b)
         (pc-byte-read (add 32 pc (sub1 k)) mem)
         (pc-read-mem pc mem (sub1 k)))))

; pc-word-read reads a word from the memory at pc.
(defn pc-word-read (pc mem)
  (pc-read-mem pc mem (wsz)))

; pc-long-read reads a longword from the memory at pc.
(defn pc-long-read (pc mem)
  (pc-read-mem pc mem (lsz)))

; We define some bit field extractors.  The function names reflect the
; meanings of the fields for MC68020 instructions.

; The source register field.  s_rn is a function of one argument, ins,
; which should be a word, i.e., a 16-bit bit-vector.
; Nonnegative integer value of bits 0..2 of ins.
(defn s_rn (ins)
  (bits ins 0 2))

; The source mode field.  Integer value of bits 3..5 of ins.
(defn s_mode (ins)
  (bits ins 3 5))

; The destination mode field.  Integer value of bits 6..8 of ins.
(defn d_mode (ins)
  (bits ins 6 8))

; The destination register field.  Integer value of bits 9..11 of ins.
(defn d_rn (ins)
  (bits ins 9 11))
```

```
; The op-mode field.  Integer value of bits 6..8 of ins.
(defn opmode-field (ins)
  (bits ins 6 8))

; The condition field.  Integer value of bits 8..11 of ins.
(defn cond-field (ins)
  (bits ins 8 11))

; By the ``oplen'' of an instruction we mean whether an instruction
; deals with a byte, word, long word, or quad word operation.

; The oplen of the operation is normally determined by bits 6 and 7.
; op-len is a function of one argument, ins, which normally is the
; first word of an instruction.
;
;  67 & (common bit numbers)
;  00 & byte
;  10 & word
;  01 & long word
;  11 & illegal, but we return (qsz).
;
(defn op-len (ins)
  (times (b) (exp 2 (bits ins 6 7))))

;              Storing the Result
; byte-writep determines whether the location x is writable with respect
; to the current memory.
(defn byte-writep (x mem)
  (writep x 32 (car mem)))

; write-memp determines whether the k consecutive bytes starting at address
; x in the memory are writable.
(defn write-memp (x mem k)
  (if (zerop k)
      t
    (and (byte-writep (add 32 x (sub1 k)) mem)
         (write-memp x mem (sub1 k)))))

; write-mem is a function of four arguments, value, x, mem, and k.
; value should be a natural number, namely the thing we are storing;
; x should be a natural number, namely the address at which to store
; value;  mem is the memory;  k is the number of bytes to store.  We
; store the bytes one byte at a time, storing the most significant
; byte of value first, at location x, and storing subsequently,
; decreasingly significant bytes at increasing addresses.
(defn byte-write (value x mem)
  (cons (car mem)
        (write (head value (b)) x 32 (cdr mem))))

(defn write-mem (value x mem k)
      (if (zerop k)
          mem
        (write-mem (tail value (b))
```

```
                  x
                  (byte-write value (add 32 x (sub1 k)) mem)
                  (sub1 k))))

; Obtain c, v, z, n, and x from CCR.  The following five functions ccr-c,
; ccr-v, ccr-z, ccr-n, and ccr-x simply access the five
; correspondingly named bits of the CCR.  We use them to
; specify the condition cc in the bcc instruction.
(defn ccr-c (ccr) (bitn ccr 0))

(defn ccr-v (ccr) (bitn ccr 1))

(defn ccr-z (ccr) (bitn ccr 2))

(defn ccr-n (ccr) (bitn ccr 3))

(defn ccr-x (ccr) (bitn ccr 4))

; Whenever instructions update the CCR, cvznx simply generates a new partial
; CCR consisting of the new cvznx-flags.
(defn cvznx (c v z n x)
  (plus (fix-bit c)
        (plus (times 2 (fix-bit v))
              (plus (times 4 (fix-bit z))
                    (plus (times 8 (fix-bit n))
                          (times 16 (fix-bit x)))))))

; set-cvznx replaces the old flags in CCR by the given flags.
(defn set-cvznx (cvznx ccr)
  (replace 5 cvznx ccr))

(defn set-c (c ccr)
  (set-cvznx (cvznx c (ccr-v ccr) (ccr-z ccr) (ccr-n ccr) (ccr-x ccr))
             ccr))

(defn set-v (v ccr)
  (set-cvznx (cvznx (ccr-c ccr) v (ccr-z ccr) (ccr-n ccr) (ccr-x ccr))
             ccr))

(defn set-z (z ccr)
  (set-cvznx (cvznx (ccr-c ccr) (ccr-v ccr) z (ccr-n ccr) (ccr-x ccr))
             ccr))

(defn set-n (n ccr)
  (set-cvznx (cvznx (ccr-c ccr) (ccr-v ccr) (ccr-z ccr) n (ccr-x ccr))
             ccr))

(defn set-x (x ccr)
  (set-cvznx (cvznx (ccr-c ccr) (ccr-v ccr) (ccr-z ccr) (ccr-n ccr) x)
             ccr))

; To halt the machine, we simply put the halting reason ``signal'' in
; the machine state.
(defn halt (signal s)
```

```
  (mc-state signal
            (mc-rfile s)
            (mc-pc s)
            (mc-ccr s)
            (mc-mem s)))

; To update the register file in the state s.
(defn update-rfile (new-rfile s)
  (mc-state (mc-status s)
            new-rfile
            (mc-pc s)
            (mc-ccr s)
            (mc-mem s)))

; To update the program counter in the state s.
(defn update-pc (new-pc s)
  (mc-state (mc-status s)
            (mc-rfile s)
            new-pc
            (mc-ccr s)
            (mc-mem s)))

; To update the condition code in the state s.
(defn update-ccr (new-ccr s)
  (mc-state (mc-status s)
            (mc-rfile s)
            (mc-pc s)
            (set-cvznx new-ccr (mc-ccr s))
            (mc-mem s)))

; To update the memory in the state s.
(defn update-mem (new-mem s)
  (mc-state (mc-status s)
            (mc-rfile s)
            (mc-pc s)
            (mc-ccr s)
            new-mem))

; read-dn and read-an are used to fetch data and address registers in
; the machine state s.
(defn read-dn (oplen dn s)
  (read-rn oplen dn (mc-rfile s)))

(defn read-an (oplen an s)
  (read-rn oplen (plus 8 an) (mc-rfile s)))

; write-dn and write-an are used to modify data and address registers in
; the machine state s. They return the modified machine state.
(defn write-dn (oplen value dn s)
  (update-rfile (write-rn oplen value dn (mc-rfile s))
                s))

(defn write-an (oplen value an s)
  (update-rfile (write-rn oplen value (plus 8 an) (mc-rfile s))
```

```
              s))

; sp is the constant 7, which refers to the stack pointer sp(a7) in the
; address register file.
(defn sp () 7)

; read-sp is a function that fetches the stack pointer in the given
; state s.
(defn read-sp (s)
  (read-an (l) (sp) s))

; write-sp is a function of two arguments, value and s.  It returns
; a new machine state with the stack pointer updated to value.
(defn write-sp (value s)
  (write-an (l) value (sp) s))

; push-up pushes value onto the sp stack and increments sp.
(defn push-sp (opsz value s)
  (let ((sp (sub (l) opsz (read-sp s))))
    (if (write-memp sp (mc-mem s) opsz)
        (update-mem (write-mem value sp (mc-mem s) opsz)
                    (write-sp sp s))
      (halt (write-signal) s))))

;           Retrieving the Operand According to Oplen

; The function operand returns the operand based on the given addr.
; addr should be a cons; the car tells us where to retrieve the operand,
; the cdr provides the real address.
(defn operand (oplen addr s)
  (if (equal (car addr) 'd)
      (read-dn oplen (cdr addr) s)
    (if (equal (car addr) 'a)
        (read-an oplen (cdr addr) s)
      (if (equal (car addr) 'm)
          (read-mem (cdr addr) (mc-mem s) (op-sz oplen))
        (cdr addr)))))

;              Effective Address Calculation

; We now begin the definition of a collection of functions culminating
; in the function effec-addr, which computes ``the effective
; address'' for MC68020 instructions.  (Actually, some instructions,
; e.g., the MOVE instruction, compute two effective addresses.)

; In his Ph.D. thesis, Warren Hunt specified the FM8502 microprocessor in
; the Nqthm logic [3].  In Hunt's FM8502 there is only one
; instruction format.  Therefore in the FM8502 ``soft-machine'' specification
; one can compute the effective addresses before looking at the op-code.
; But in the MC68020, there are several instruction formats, and the
; algorithm for computing effective addresses depends upon what the
; op-code is.  So we cannot handle instructions as uniformly as in
; FM8502.  We have to know what the op-code is at a very early stage
; in the implementation.
```

```
; Pre-effect and post-effect are two functions used in address
; register predecrement and postincrement.
(defn post-effect (oplen rn addr)
  (if (and (equal rn (sp))
           (equal oplen (b)))
      (add (l) addr (wsz))
    (add (l) addr (op-sz oplen))))

(defn pre-effect (oplen rn addr)
  (if (and (equal rn (sp))
           (equal oplen (b)))
      (sub (l) (wsz) addr)
    (sub (l) (op-sz oplen) addr)))

; For each of the different effective addressing modes, we define a
; function that ``does the work.''  In each case, the function takes
; as its argument the current value of the state, s.  Some may
; take other parameters.  In each case a cons is returned, consisting
; of (a) an internal state with possible an and pc updates after the
; effective address calculation; (b) the effective address, normally
; another cons indicating where to look and where to get the operands.
;
; Register direct modes.  Data register direct (000) and address
; register direct (001).
; Number of extension words: 0.

; dn-direct is a function of two arguments, rn and s.  rn should be
; a natural number and s should be an mc-state.  Mode 000.
(defn dn-direct (rn s)
  (cons s (cons 'd rn)))

; an-direct is a function of two arguments, rn and s.  rn should be
; a natural number and s should be an mc-state.  Mode 001.
(defn an-direct (rn s)
  (cons s (cons 'a rn)))

; Memory address modes.
; The pc argument to these effective address subroutines need not be
; the actual pc of the instruction.  In the case of the MOVE instruction,
; which involves two effective address calculations, the pc will point
; to the word before the ``next'' possible byte in the
; memory which is to be used as an extension word.  For example, the
; instruction
;        i:    move (1,a0) (2,a2)
; i.e., move the word at 1 + (a0) to 2 + (a2), requires altogether 3
; words because two extension words are required, one for each of the
; displacements (1 and 2).  When we invoke the function addr-disp for
; the calculation of the first effective address, the pc will be i.
; But when we again invoke the function addr-disp for the calculation
; of the second effective address, the pc will be i+2.

; A subtlety about pc displacement.  The one MC68020 instruction that
; involves two effective address calculations, the MOVE instruction,
```

```
; will have its second effective address calculation performed by us
; with the pc not pointing necessarily to the MOVE instruction but
; rather (possibly) pointing to the next word after the calculation
; of the first effective address.  However, this discrepancy does not
; cause a problem with pc relative addressing because pc relative
; addressing is prohibited in the second effective address calculation.
;
; Address register indirect, mode 010.
; Number of extension words: 0.
; addr-indirect is a function of two arguments, rn and s.  rn should
; be a natural number and s should be a machine state.  It returns the
; contents of the rn element of the address register file.
(defn addr-indirect (rn s)
  (cons s (cons 'm (read-an (l) rn s))))

; Address register indirect with postincrement, mode 011.
; Number of extension words: 0.
(defn addr-postinc (oplen rn s)
  (let ((addr (read-an (l) rn s)))
    (cons (write-an (l) (post-effect oplen rn addr) rn s)
          (cons 'm addr))))

; Address register indirect with predecrement, mode 100.
; Number of extension words: 0.
; The function addr-predec returns a cons of the given state s and
; the contents of the rn element of the register file after
; the register has been predecremented.
(defn addr-predec (oplen rn s)
  (let ((addr (read-an (l) rn s)))
    (cons (write-an (l) (pre-effect oplen rn addr) rn s)
          (cons 'm (pre-effect oplen rn addr)))))

; Address register indirect with index, mode 101.
; Number of extension words: 1.
; We now begin handling an effective address calculation which involves
; an extension word.  In this mode, we add in the sign-extended 16-bit
; quantity in the word after the pc.  We return a cons with (a) the
; state with pc incremented and (b) the sum of the address register rn
; and the sign-extended contents of the next word.
(defn addr-disp (pc rn s)
  (if (pc-word-readp pc (mc-mem s))
      (cons (update-pc (add (l) pc (wsz)) s)
            (cons 'm (add (l)
                          (read-an (l) rn s)
                          (ext (w) (pc-word-read pc (mc-mem s)) (l)))))
    (cons (halt (pc-signal) s) nil)))

; Address register indirect with index (8-bit displacement), mode 110.
; Number of extension words: 1.
(defn index-rn (indexwd)
  (bits indexwd 12 14))

(defn index-register (indexwd s)
  (if (b0p (bitn indexwd 15))
```

```
      (if (b0p (bitn indexwd 11))
          (ext (w) (read-dn (w) (index-rn indexwd) s) (l))
        (read-dn (l) (index-rn indexwd) s))
    (if (b0p (bitn indexwd 11))
        (ext (w) (read-an (w) (index-rn indexwd) s) (l))
      (read-an (l) (index-rn indexwd) s))))

(defn ir-scaled (indexwd s)
  (asl (l)
       (index-register indexwd s)
       (bits indexwd 9 10)))

(defn addr-index-disp (pc rn indexwd s)
  (cons (update-pc pc s)
        (cons 'm (add (l)
                      (add (l)
                           (read-an (l) rn s)
                           (ext (b) (head indexwd (b)) (l)))
                      (ir-scaled indexwd s)))))

; Address register indirect with index (base displacement), mode 110.
; Number of extension words: 1, 2, or 3.
(defn addr-index-bd (pc addr indexwd s)
  (cons (update-pc pc s)
        (cons 'm (add (l) addr (ir-scaled indexwd s)))))

; Memory indirect without index, mode 110.
; Number of extension words: 1, 2, 3, 4, or 5.
(defn mem-indirect (pc addr olen s)
  (if (long-readp addr (mc-mem s))
      (if (pc-read-memp pc (mc-mem s) (op-sz olen))
          (cons (update-pc (add (l) pc (op-sz olen)) s)
                (cons 'm (add (l)
                              (long-read addr (mc-mem s))
                              (ext olen
                                   (pc-read-mem pc (mc-mem s) (op-sz olen))
                                   (l)))))
        (cons (halt (pc-signal) s) nil))
    (cons (halt (read-signal) s) nil)))

; Memory indirect postindexed mode.
(defn mem-postindex (pc addr indexwd olen s)
  (if (long-readp addr (mc-mem s))
      (if (pc-read-memp pc (mc-mem s) (op-sz olen))
          (cons (update-pc (add (l) pc (op-sz olen)) s)
                (cons 'm
                      (add (l)
                           (add (l)
                                (long-read addr (mc-mem s))
                                (ir-scaled indexwd s))
                           (ext olen
                                (pc-read-mem pc (mc-mem s) (op-sz olen))
                                (l)))))
        (cons (halt (pc-signal) s) nil))
```

```
    (cons (halt (read-signal) s) nil)))

; Memory indirect preindexed mode.
(defn mem-preindex (pc addr indexwd olen s)
  (mem-indirect pc (add (l) addr (ir-scaled indexwd s)) olen s))

(defn i-is (indexwd)
  (bits indexwd 0 2))

; The base displacement has been added to addr, if necessary.  addr-index3
; is to consider the index register and index/indirect selection.
(defn addr-index3 (pc addr indexwd s)
  (if (b0p (bitn indexwd 6))
      (if (lessp (i-is indexwd) 4)
          (if (lessp (i-is indexwd) 2)
              (if (equal (i-is indexwd) 0)
                  (addr-index-bd pc addr indexwd s)
                (mem-preindex pc addr indexwd 0 s))
            (if (equal (i-is indexwd) 2)
                (mem-preindex pc addr indexwd (w) s)
              (mem-preindex pc addr indexwd (l) s)))
        (if (lessp (i-is indexwd) 6)
            (if (equal (i-is indexwd) 4)
                (cons (halt (reserved-signal) s) nil)
              (mem-postindex pc addr indexwd 0 s))
          (if (equal (i-is indexwd) 6)
              (mem-postindex pc addr indexwd (w) s)
            (mem-postindex pc addr indexwd (l) s))))
    (if (lessp (i-is indexwd) 4)
        (if (lessp (i-is indexwd) 2)
            (if (equal (i-is indexwd) 0)
                (cons (update-pc pc s) (cons 'm addr))
              (mem-indirect pc addr 0 s))
          (if (equal (i-is indexwd) 2)
              (mem-indirect pc addr (w) s)
            (mem-indirect pc addr (l) s)))
      (cons (halt (reserved-signal) s) nil))))

(defn bd-sz (indexwd)
  (bits indexwd 4 5))

; The address register (base register) has been added to addr, if necessary.
; addr-index2 is to consider the base displacement.
(defn addr-index2 (pc addr indexwd s)
  (if (lessp (bd-sz indexwd) 2)
      (if (equal (bd-sz indexwd) 0)
          (cons (halt (reserved-signal) s) nil)
        (addr-index3 pc addr indexwd s))
    (if (equal (bd-sz indexwd) 2)
        (if (pc-word-readp pc (mc-mem s))
            (addr-index3 (add (l) pc (wsz))
                         (add (l)
                              addr
                              (ext (w) (pc-word-read pc (mc-mem s)) (l)))
```

```
                        indexwd
                        s)
          (cons (halt (pc-signal) s) nil))
      (if (pc-long-readp pc (mc-mem s))
          (addr-index3 (add (l) pc (lsz))
                       (add (l) addr (pc-long-read pc (mc-mem s)))
                       indexwd
                       s)
        (cons (halt (pc-signal) s) nil)))))

(defn bs-register (rn indexwd s)
  (if (b0p (bitn indexwd 7))
      (read-an (l) rn s)
    0))

; addr-index1 is to consider the address register (base register).
(defn addr-index1 (pc rn indexwd s)
  (if (b0p (bitn indexwd 8))
      (addr-index-disp pc rn indexwd s)
    (if (b0p (bitn indexwd 3))
        (addr-index2 pc (bs-register rn indexwd s) indexwd s)
      (cons (halt (reserved-signal) s) nil))))

(defn addr-index (pc rn s)
  (if (pc-word-readp pc (mc-mem s))
      (addr-index1 (add (l) pc (wsz)) rn (pc-word-read pc (mc-mem s)) s)
    (cons (halt (pc-signal) s) nil)))

; Absolute short address.  Mode 111, rn 000.
(defn absolute-short (pc s)
  (if (pc-word-readp pc (mc-mem s))
      (cons (update-pc (add (l) pc (wsz)) s)
            (cons 'm (ext (w) (pc-word-read pc (mc-mem s)) (l))))
    (cons (halt (pc-signal) s) nil)))

; Absolute long address.  Mode 111, rn 001.
(defn absolute-long (pc s)
  (if (pc-long-readp pc (mc-mem s))
      (cons (update-pc (add (l) pc (lsz)) s)
            (cons 'm (pc-long-read pc (mc-mem s))))
    (cons (halt (pc-signal) s) nil)))

; Surprisingly, the design of the MC68020 deliberately avoids having
; two program counter addressing modes.  This specification here
; relies on this very fact.

; Program counter indirect with displacement.  Mode 111, rn 010.
; Number of extension words: 1.
(defn pc-disp (pc s)
  (if (pc-word-readp pc (mc-mem s))
      (cons (update-pc (add (l) pc (wsz)) s)
            (cons 'm
                  (add (l)
                       pc
```

```
                     (ext (w) (pc-word-read pc (mc-mem s)) (l)))))
    (cons (halt (pc-signal) s) nil)))

; Program counter indirect with index (8-bit displacement). mode 111, rn 011.
(defn pc-index-disp (pc indexwd s)
  (cons (update-pc (add (l) pc (wsz)) s)
        (cons 'm (add (l)
                      (add (l)
                           pc
                           (ext (b) (head indexwd (b)) (l)))
                      (ir-scaled indexwd s)))))

; Program counter indirect with index (base displacement) mode.
; Program counter memory indirect postindexed mode.
; Program counter memory indirect preindexed mode.
(defn bs-pc (pc indexwd)
  (if (b0p (bitn indexwd 7)) pc  0))

(defn pc-index1 (pc indexwd s)
  (if (b0p (bitn indexwd 8))
      (pc-index-disp pc indexwd s)
    (if (b0p (bitn indexwd 3))
        (addr-index2 (add (l) pc (wsz)) (bs-pc pc indexwd) indexwd s)
      (cons (halt (reserved-signal) s) nil))))

(defn pc-index (pc s)
  (if (pc-word-readp pc (mc-mem s))
      (pc-index1 pc (pc-word-read pc (mc-mem s)) s)
    (cons (halt (pc-signal) s) nil)))

; Immediate data. Mode 111, rn 100.
; Number of extension words: 1 or 2.
(defn immediate (oplen pc s)
  (if (equal oplen (b))
      (if (pc-word-readp pc (mc-mem s))
          (cons (update-pc (add (l) pc (wsz)) s)
                (cons 'i (pc-byte-read (add (l) pc (bsz)) (mc-mem s))))
        (cons (halt (pc-signal) s) nil))
    (if (pc-read-memp pc (mc-mem s) (op-sz oplen))
        (cons (update-pc (add (l) pc (op-sz oplen)) s)
              (cons 'i (pc-read-mem pc (mc-mem s) (op-sz oplen))))
      (cons (halt (pc-signal) s) nil))))

; Effective address calculation.  effec-addr is a function of
; four arguments, oplen, mode, rn, and s.  oplen should be (b),
; (w), or (l); it is the size of the datum we are computing the
; effective address of.  mode is a natural number extracted from
; the first word of the instruction;  mode indicates pre-decrement,
; post-increment, etc.  rn is a natural number extracted from the
; first word of the instruction; rn designates a register. s the
; current machine state.  effec-addr returns a pair, or cons as it
; is called in Lisp and Nqthm.  The first element (or car) of this
; pair is an internal state after this effective address calculation.
; The second element (or cdr) is another cons consisting of the
```

```
; direction ('d, 'a, 'm, or 'i),  and the effective address (a
; nonnegative integer).  Because MC68020 instructions can be as
; many as 11 words long, the calculation of the next pc is intimately
; tied to the effective address calculation.
(defn effec-addr (oplen mode rn s)
  (if (lessp mode 4)
      (if (lessp mode 2)
          (if (equal mode 0)
              (dn-direct rn s)                    ; 000
            (an-direct rn s))                     ; 001
        (if (equal mode 2)
            (addr-indirect rn s)                  ; 010
          (addr-postinc oplen rn s)))             ; 011
    (if (lessp mode 6)
        (if (equal mode 4)
            (addr-predec oplen rn s)              ; 100
          (addr-disp (mc-pc s) rn s))             ; 101
      (if (equal mode 6)
          (addr-index (mc-pc s) rn s)             ; 110
        (if (lessp rn 4)                          ; 111
            (if (lessp rn 2)
                (if (equal rn 0)
                    (absolute-short (mc-pc s) s)  ; 111 000
                  (absolute-long (mc-pc s) s))    ; 111 001
              (if (equal rn 2)
                  (pc-disp (mc-pc s) s)           ; 111 010
                (pc-index (mc-pc s) s)))          ; 111 011
          (immediate oplen (mc-pc s) s)))))       ; 111 100
 )

; Given an effective address field, test if it is one of the existing
; addressing modes.
(defn addr-modep (mode rn)
  (if (equal mode 7) (leq rn 4) t))

; Given an effective address field, test if it is a data addressing mode.
(defn data-addr-modep (mode rn)
  (if (equal mode 7) (leq rn 4) (not (equal mode 1))))

; Given an effective address field, test if it is a memory addressing mode.
(defn memory-addr-modep (mode rn)
  (if (equal mode 7) (leq rn 4) (geq mode 2)))

; Given an effective address field, test if it is a control addressing mode.
(defn control-addr-modep (mode rn)
  (if (equal mode 7)
      (leq rn 3)
    (or (equal mode 2)
        (geq mode 5))))

; Given an effective address field, test if it is an alterable addressing mode.
(defn alterable-addr-modep (mode rn)
  (or (not (equal mode 7))
      (equal rn 0)
```

```
      (equal rn 1)))

; dn-direct-modep returns t if the addressing mode is a data register direct.
; Returns f otherwise.
(defn dn-direct-modep (mode)
  (equal mode 0))

; an-direct-modep returns t if the addressing mode is an address register direct,
; and returns f otherwise.
(defn an-direct-modep (mode)
  (equal mode 1))

; Postincrement.
(defn postinc-modep (mode)
  (equal mode 3))

; Predecrement.
(defn predec-modep (mode)
  (equal mode 4))

(defn idata-modep (mode rn)
  (and (equal mode 7)
       (equal rn 4)))

; In address register direct (001), a byte size operation is not allowed.
(defn byte-an-direct-modep (oplen mode)
  (and (equal oplen (b))
       (an-direct-modep mode)))

; An internal state in the execution of one instruction.
(defn mc-instate (oplen ins s)
  (let ((s&addr (effec-addr oplen (s_mode ins) (s_rn ins) s)))
    (if (equal (cadr s&addr) 'm)
        (if (read-memp (cddr s&addr) (mc-mem s) (op-sz oplen))
            s&addr
          (cons (halt (read-signal) s) nil))
      s&addr)))

; Mapping functions.  mapping finishes the execution of instructions.
; mapping maps a machine state into the next state.
(defn d-mapping (oplen v&cvznx addr s)
  (mc-state (mc-status s)
            (write-rn oplen (car v&cvznx) addr (mc-rfile s))
            (mc-pc s)
            (set-cvznx (cdr v&cvznx) (mc-ccr s))
            (mc-mem s)))

(defn a-mapping (oplen v&cvznx addr s)
  (mc-state (mc-status s)
            (write-rn oplen (car v&cvznx) (plus 8 addr) (mc-rfile s))
            (mc-pc s)
            (set-cvznx (cdr v&cvznx) (mc-ccr s))
            (mc-mem s)))
```

```
(defn m-mapping (oplen v&cvznx addr s)
  (if (write-memp addr (mc-mem s) (op-sz oplen))
      (mc-state (mc-status s)
                (mc-rfile s)
                (mc-pc s)
                (set-cvznx (cdr v&cvznx) (mc-ccr s))
                (write-mem (car v&cvznx) addr (mc-mem s) (op-sz oplen)))
    (halt (write-signal) s)))

(defn mapping (oplen v&cvznx s&addr)
  (if (equal (cadr s&addr) 'd)
      (d-mapping oplen v&cvznx (cddr s&addr) (car s&addr))
    (if (equal (cadr s&addr) 'a)
        (a-mapping oplen v&cvznx (cddr s&addr) (car s&addr))
      (m-mapping oplen v&cvznx (cddr s&addr) (car s&addr)))))


;        The Individual Instructions

; ADD instruction.
; The computation of the condition code register(CCR).
(defn add-c (n sopd dopd)
  (let ((result (add n sopd dopd)))
    (b-or (b-or (b-and (mbit sopd n) (mbit dopd n))
                (b-and (b-not (mbit result n)) (mbit dopd n)))
          (b-and (mbit sopd n) (b-not (mbit result n))))))

(defn add-v (n sopd dopd)
  (let ((result (add n sopd dopd)))
    (b-or (b-and (b-and (mbit sopd n) (mbit dopd n))
                 (b-not (mbit result n)))
          (b-and (b-and (b-not (mbit sopd n)) (b-not (mbit dopd n)))
                 (mbit result n)))))

(defn add-z (oplen sopd dopd)
  (if (equal (add oplen dopd sopd) 0)
      (b1) (b0)))

(defn add-n (oplen sopd dopd)
  (mbit (add oplen dopd sopd) oplen))

(defn add-cvznx (oplen sopd dopd)
  (cvznx (add-c oplen sopd dopd)
         (add-v oplen sopd dopd)
         (add-z oplen sopd dopd)
         (add-n oplen sopd dopd)
         (add-c oplen sopd dopd)))

; The effects of the execution of an ADD instruction are given as follows.
(defn add-effect (oplen sopd dopd)
  (cons (add oplen dopd sopd)
        (add-cvznx oplen sopd dopd)))

; Test if the addressing mode is legal.
```

```
(defn add-addr-modep1 (oplen ins)
  (and (addr-modep (s_mode ins) (s_rn ins))
       (not (byte-an-direct-modep oplen (s_mode ins)))))

(defn add-addr-modep2 (ins)
  (and (alterable-addr-modep (s_mode ins) (s_rn ins))
       (memory-addr-modep (s_mode ins) (s_rn ins))))

; An execution of an ADD instruction.
(defn add-ins1 (oplen ins s)
  (if (add-addr-modep1 oplen ins)
      (let ((s&addr (mc-instate oplen ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (d-mapping oplen
                     (add-effect oplen
                                 (operand oplen (cdr s&addr) s)
                                 (read-dn oplen (d_rn ins) s))
                     (d_rn ins)
                     (car s&addr))))
    (halt (mode-signal) s)))

(defn add-mapping (opd oplen ins s)
  (let ((s&addr (mc-instate oplen ins s)))
    (if (mc-haltp (car s&addr))
        (car s&addr)
      (mapping oplen
               (add-effect oplen opd (operand oplen (cdr s&addr) s))
               s&addr))))

(defn add-ins2 (oplen ins s)
  (if (add-addr-modep2 ins)
      (add-mapping (read-dn oplen (d_rn ins) s)
                   oplen
                   ins
                   s)
    (halt (mode-signal) s)))

; ADDA instruction.
(defn adda-addr-modep (ins)
  (addr-modep (s_mode ins) (s_rn ins)))

; Notice that the ADDA instruction does not affect CCR.
(defn adda-ins (oplen ins s)
  (if (adda-addr-modep ins)
      (let ((s&addr (mc-instate oplen ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (write-an (l)
                    (add (l)
                         (read-an (l) (d_rn ins) (car s&addr))
                         (ext oplen (operand oplen (cdr s&addr) s) (l)))
                    (d_rn ins)
                    (car s&addr))))
```

```
    (halt (mode-signal) s)))

; ADDX instruction.
(defn addx-c (n x sopd dopd)
  (let ((result (adder n x sopd dopd)))
    (b-or (b-or (b-and (mbit sopd n) (mbit dopd n))
                (b-and (b-not (mbit result n)) (mbit dopd n)))
          (b-and (mbit sopd n) (b-not (mbit result n))))))

(defn addx-v (n x sopd dopd)
  (let ((result (adder n x sopd dopd)))
    (b-or (b-and (b-and (mbit sopd n) (mbit dopd n))
                 (b-not (mbit result n)))
          (b-and (b-and (b-not (mbit sopd n))
                        (b-not (mbit dopd n)))
                 (mbit result n)))))

(defn addx-z (oplen z x sopd dopd)
  (b-and z
         (if (equal (adder oplen x dopd sopd) 0)
             (b1) (b0))))

(defn addx-n (oplen x sopd dopd)
  (mbit (adder oplen x dopd sopd) oplen))

(defn addx-cvznx (oplen z x sopd dopd)
  (cvznx (addx-c oplen x sopd dopd)
         (addx-v oplen x sopd dopd)
         (addx-z oplen z x sopd dopd)
         (addx-n oplen x sopd dopd)
         (addx-c oplen x sopd dopd)))

(defn addx-effect (oplen sopd dopd ccr)
  (cons (adder oplen (ccr-x ccr) dopd sopd)
        (addx-cvznx oplen (ccr-z ccr) (ccr-x ccr) sopd dopd)))

(defn addx-ins1 (oplen ins s)
  (d-mapping oplen
             (addx-effect oplen
                          (read-dn oplen (s_rn ins) s)
                          (read-dn oplen (d_rn ins) s)
                          (mc-ccr s))
             (d_rn ins)
             s))

(defn addx-ins2 (oplen ins s)
  (let ((s&addr0 (addr-predec oplen (s_rn ins) s)))
    (if (read-memp (cddr s&addr0) (mc-mem s) (op-sz oplen))
        (let ((s&addr (addr-predec oplen (d_rn ins) (car s&addr0))))
          (if (read-memp (cddr s&addr) (mc-mem s) (op-sz oplen))
              (mapping oplen
                       (addx-effect oplen
                                    (operand oplen (cdr s&addr0) (car s&addr0))
                                    (operand oplen (cdr s&addr) (car s&addr))
```

```
                                  (mc-ccr s))
                        s&addr)
              (halt (read-signal) s)))
      (halt (read-signal) s))))

; Opcode 1101.
; The ADD instruction group includes three instructions ADD, ADDA, and ADDX.
(defn add-group (opmode ins s)
  (if (lessp opmode 4)
      (if (equal opmode 3)
          (adda-ins (w) ins s)
        (add-ins1 (op-len ins) ins s))
    (if (equal opmode 7)
        (adda-ins (l) ins s)
      (if (equal (s_mode ins) 0)
          (addx-ins1 (op-len ins) ins s)
        (if (equal (s_mode ins) 1)
            (addx-ins2 (op-len ins) ins s)
          (add-ins2 (op-len ins) ins s))))))

; SUB instruction.
; The computation of the condition code register (ccr).
(defn sub-c (n sopd dopd)
  (let ((result (sub n sopd dopd)))
    (b-or (b-or (b-and (mbit sopd n) (b-not (mbit dopd n)))
                (b-and (mbit result n) (b-not (mbit dopd n))))
          (b-and (mbit sopd n) (mbit result n)))))

(defn sub-v (n sopd dopd)
  (let ((result (sub n sopd dopd)))
    (b-or (b-and (b-and (b-not (mbit sopd n)) (mbit dopd n))
                 (b-not (mbit result n)))
          (b-and (b-and (mbit sopd n) (b-not (mbit dopd n)))
                 (mbit result n)))))

(defn sub-z (oplen sopd dopd)
  (if (equal (sub oplen sopd dopd) 0) (b1) (b0)))

(defn sub-n (oplen sopd dopd)
  (mbit (sub oplen sopd dopd) oplen))

(defn sub-cvznx (oplen sopd dopd)
  (cvznx (sub-c oplen sopd dopd)
         (sub-v oplen sopd dopd)
         (sub-z oplen sopd dopd)
         (sub-n oplen sopd dopd)
         (sub-c oplen sopd dopd)))

; The effect of an execution of a SUB instruction.
(defn sub-effect (oplen sopd dopd)
  (cons (sub oplen sopd dopd)
        (sub-cvznx oplen sopd dopd)))

; Test if the addressing mode is illegal.
```

```
(defn sub-addr-modep1 (oplen ins)
  (and (addr-modep (s_mode ins) (s_rn ins))
       (not (byte-an-direct-modep oplen (s_mode ins)))))

(defn sub-addr-modep2 (ins)
  (and (alterable-addr-modep (s_mode ins) (s_rn ins))
       (memory-addr-modep (s_mode ins) (s_rn ins))))

; The execution of the SUB instruction.
(defn sub-ins1 (oplen ins s)
  (if (sub-addr-modep1 oplen ins)
      (let ((s&addr (mc-instate oplen ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (d-mapping oplen
                     (sub-effect oplen
                                 (operand oplen (cdr s&addr) s)
                                 (read-dn oplen (d_rn ins) s))
                     (d_rn ins)
                     (car s&addr))))
    (halt (mode-signal) s)))

(defn sub-mapping (opd oplen ins s)
  (let ((s&addr (mc-instate oplen ins s)))
    (if (mc-haltp (car s&addr))
        (car s&addr)
      (mapping oplen
               (sub-effect oplen opd (operand oplen (cdr s&addr) s))
               s&addr))))

(defn sub-ins2 (oplen ins s)
  (if (sub-addr-modep2 ins)
      (sub-mapping (read-dn oplen (d_rn ins) s)
                   oplen
                   ins
                   s)
    (halt (mode-signal) s)))

; SUBA instruction.
(defn suba-addr-modep (ins)
  (addr-modep (s_mode ins) (s_rn ins)))

(defn suba-ins (oplen ins s)
  (if (suba-addr-modep ins)
      (let ((s&addr (mc-instate oplen ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (write-an (l)
                    (sub (l)
                         (ext oplen (operand oplen (cdr s&addr) s) (l))
                         (read-an (l) (d_rn ins) (car s&addr)))
                    (d_rn ins)
                    (car s&addr))))
    (halt (mode-signal) s)))
```

```
; SUBX instruction.
(defn subx-c (n x sopd dopd)
  (let ((result (subtractor n x sopd dopd)))
    (b-or (b-or (b-and (mbit sopd n) (b-not (mbit dopd n)))
                (b-and (mbit result n) (b-not (mbit dopd n))))
          (b-and (mbit sopd n) (mbit result n)))))

(defn subx-v (n x sopd dopd)
  (let ((result (subtractor n x sopd dopd)))
    (b-or (b-and (b-and (b-not (mbit sopd n)) (mbit dopd n))
                 (b-not (mbit result n)))
          (b-and (b-and (mbit sopd n) (b-not (mbit dopd n)))
                 (mbit result n)))))

(defn subx-z (oplen z x sopd dopd)
  (b-and z
         (if (equal (subtractor oplen x sopd dopd) 0)
             (b1) (b0))))

(defn subx-n (oplen x sopd dopd)
  (mbit (subtractor oplen x sopd dopd) oplen))

(defn subx-cvznx (oplen z x sopd dopd)
  (cvznx (subx-c oplen x sopd dopd)
         (subx-v oplen x sopd dopd)
         (subx-z oplen z x sopd dopd)
         (subx-n oplen x sopd dopd)
         (subx-c oplen x sopd dopd)))

(defn subx-effect (oplen sopd dopd ccr)
  (cons (subtractor oplen (ccr-x ccr) sopd dopd)
        (subx-cvznx oplen (ccr-z ccr) (ccr-x ccr) sopd dopd)))

(defn subx-ins1 (oplen ins s)
  (d-mapping oplen
             (subx-effect oplen
                          (read-dn oplen (s_rn ins) s)
                          (read-dn oplen (d_rn ins) s)
                          (mc-ccr s))
             (d_rn ins)
             s))

(defn subx-ins2 (oplen ins s)
  (let ((s&addr0 (addr-predec oplen (s_rn ins) s)))
    (if (read-memp (cddr s&addr0) (mc-mem s) (op-sz oplen))
        (let ((s&addr (addr-predec oplen (d_rn ins) (car s&addr0))))
          (if (read-memp (cddr s&addr) (mc-mem s) (op-sz oplen))
              (mapping oplen
                       (subx-effect oplen
                                    (operand oplen (cdr s&addr0) (car s&addr0))
                                    (operand oplen (cdr s&addr) (car s&addr))
                                    (mc-ccr s))
                       s&addr)
```

```
            (halt (read-signal) s)))
      (halt (read-signal) s))))

; Opcode 1001.
; The SUB instruction group includes three instructions SUB, SUBA, and SUBX.
(defn sub-group (opmode ins s)
  (if (lessp opmode 4)
      (if (equal opmode 3)
          (suba-ins (w) ins s)
        (sub-ins1 (op-len ins) ins s))
    (if (equal opmode 7)
        (suba-ins (l) ins s)
      (if (equal (s_mode ins) 0)
          (subx-ins1 (op-len ins) ins s)
        (if (equal (s_mode ins) 1)
            (subx-ins2 (op-len ins) ins s)
          (sub-ins2 (op-len ins) ins s))))))

; AND instruction.
; The computation of the condition code register(CCR).
(defn and-z (oplen sopd dopd)
  (if (equal (logand sopd dopd) 0) (b1) (b0)))

(defn and-n (oplen sopd dopd)
  (mbit (logand sopd dopd) oplen))

(defn and-cvznx (oplen sopd dopd ccr)
  (cvznx (b0)
         (b0)
         (and-z oplen sopd dopd)
         (and-n oplen sopd dopd)
         (ccr-x ccr)))

; The effect of the execution of the AND instruction.
(defn and-effect (oplen sopd dopd ccr)
  (cons (logand sopd dopd)
        (and-cvznx oplen sopd dopd ccr)))

; Test if the addressing mode is legal.
(defn and-addr-modep1 (ins)
  (data-addr-modep (s_mode ins) (s_rn ins)))

(defn and-addr-modep2 (ins)
  (and (alterable-addr-modep (s_mode ins) (s_rn ins))
       (memory-addr-modep (s_mode ins) (s_rn ins))))

; The execution of the AND instruction.
(defn and-ins1 (oplen ins s)
  (if (and-addr-modep1 ins)
      (let ((s&addr (mc-instate oplen ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (d-mapping oplen
                     (and-effect oplen
```

```
                                (operand oplen (cdr s&addr) s)
                                (read-dn oplen (d_rn ins) s)
                                (mc-ccr s))
                    (d_rn ins)
                    (car s&addr))))
    (halt (mode-signal) s)))

(defn and-mapping (sopd oplen ins s)
  (let ((s&addr (mc-instate oplen ins s)))
    (if (mc-haltp (car s&addr))
        (car s&addr)
      (mapping oplen
               (and-effect oplen
                           sopd
                           (operand oplen (cdr s&addr) s)
                           (mc-ccr s))
               s&addr))))

(defn and-ins2 (oplen ins s)
  (if (and-addr-modep2 ins)
      (and-mapping (read-dn oplen (d_rn ins) s)
                   oplen
                   ins
                   s)
    (halt (mode-signal) s)))

; MULU.W/MULS.W instruction.  S * D --> D.
; MULU expects x and y to be two natural numbers.
(defn mulu (n x y i)
  (head (times x y) n))

; MULS expects x and y to be two natural numbers.
(defn muls (n x y i)
  (head (int-to-nat (itimes (nat-to-int x i) (nat-to-int y i))
                    (times 2 i))
        n))

(defn mul&div-addr-modep (ins)
  (data-addr-modep (s_mode ins) (s_rn ins)))

; The condition codes for MULU.
(defn mulu-v (n sopd dopd i)
  (if (lessp (times sopd dopd) (exp 2 n))
      (b0) (b1)))

(defn mulu-z (n sopd dopd i)
  (if (equal (mulu n sopd dopd i) 0) (b1) (b0)))

(defn mulu-n (n sopd dopd i)
  (mbit (mulu n sopd dopd i) n))

(defn mulu-cvznx (n sopd dopd i ccr)
  (cvznx (b0)
         (mulu-v n sopd dopd i)
```

```
        (mulu-z n sopd dopd i)
        (mulu-n n sopd dopd i)
        (ccr-x ccr)))

(defn mulu_w-ins (sopd dn s)
  (let ((dopd (read-dn (w) dn s)))
    (update-ccr (mulu-cvznx (l) sopd dopd (w) (mc-ccr s))
                (write-dn (l) (mulu (l) sopd dopd (w)) dn s))))

; The condition codes for MULS.
(defn muls-v (n sopd dopd i)
  (if (int-rangep (itimes (nat-to-int sopd i)
                          (nat-to-int dopd i))
                  n)
      (b0) (b1)))

(defn muls-z (n sopd dopd i)
  (if (equal (muls n sopd dopd i) 0)
      (b1) (b0)))

(defn muls-n (n sopd dopd i)
  (mbit (muls n sopd dopd i) n))

(defn muls-cvznx (n sopd dopd i ccr)
  (cvznx (b0)
         (muls-v n sopd dopd i)
         (muls-z n sopd dopd i)
         (muls-n n sopd dopd i)
         (ccr-x ccr)))

(defn muls_w-ins (sopd dn s)
  (let ((dopd (read-dn (w) dn s)))
    (update-ccr (muls-cvznx (l) sopd dopd (w) (mc-ccr s))
                (write-dn (l) (muls (l) sopd dopd (w)) dn s))))

; EXG instruction.
; Exchange the contents of two data registers.
(defn exg-drdr-ins (ins s)
  (let ((dx (read-dn (l) (d_rn ins) s))
        (dy (read-dn (l) (s_rn ins) s)))
    (write-dn (l)
              dy
              (d_rn ins)
              (write-dn (l) dx (s_rn ins) s))))

; Exchange the contents of two address registers.
(defn exg-arar-ins (ins s)
  (let ((ax (read-an (l) (d_rn ins) s))
        (ay (read-an (l) (s_rn ins) s)))
    (write-an (l)
              ay
              (d_rn ins)
              (write-an (l) ax (s_rn ins) s))))
```

```
; Exchange the contents of data and address registers.
(defn exg-drar-ins (ins s)
  (let ((dx (read-dn (l) (d_rn ins) s))
        (ay (read-an (l) (s_rn ins) s)))
    (write-dn (l)
              ay
              (d_rn ins)
              (write-an (l) dx (s_rn ins) s))))

(defn mul_w-ins (ins s)
  (if (mul&div-addr-modep ins)
      (let ((s&addr (mc-instate (w) ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (let ((sopd (operand (w) (cdr s&addr) s)))
            (if (b0p (bitn ins 8))
                (mulu_w-ins sopd (d_rn ins) (car s&addr))
              (muls_w-ins sopd (d_rn ins) (car s&addr))))))
    (halt (mode-signal) s)))

; Opcode 1100.
; The AND instruction group includes three instructions AND, MULS.W/MULU.W,
; and EXG.  Detect ABCD.
(defn and-group (oplen ins s)
  (if (equal oplen (q))
      (mul_w-ins ins s)
    (if (b0p (bitn ins 8))
        (and-ins1 oplen ins s)
      (if (lessp (s_mode ins) 2)
          (if (equal oplen (b))
              (halt 'abcd-unspecified s)
            (if (equal oplen (w))
                (if (equal (s_mode ins) 0)
                    (exg-drdr-ins ins s)
                  (exg-arar-ins ins s))
              (if (equal (s_mode ins) 0)
                  (halt (reserved-signal) s)
                (exg-drar-ins ins s))))
        (and-ins2 oplen ins s)))))

; OR instruction.
; The computation of the condition code register.
(defn or-z (oplen sopd dopd)
  (if (equal (logor sopd dopd) 0) (b1) (b0)))

(defn or-n (oplen sopd dopd)
  (b-or (mbit sopd oplen)
        (mbit dopd oplen)))

(defn or-cvznx (oplen sopd dopd ccr)
  (cvznx (b0)
         (b0)
         (or-z oplen sopd dopd)
         (or-n oplen sopd dopd)
```

```
        (ccr-x ccr)))

; The effect of an execution of an  OR instruction.
(defn or-effect (oplen sopd dopd ccr)
  (cons (logor sopd dopd)
        (or-cvznx oplen sopd dopd ccr)))

; Test if the addressing mode is illegal.
(defn or-addr-modep1 (ins)
  (data-addr-modep (s_mode ins) (s_rn ins)))

(defn or-addr-modep2 (ins)
  (and (alterable-addr-modep (s_mode ins) (s_rn ins))
       (memory-addr-modep (s_mode ins) (s_rn ins))))

; The execution of the OR instruction.
(defn or-ins1 (oplen ins s)
  (if (or-addr-modep1 ins)
      (let ((s&addr (mc-instate oplen ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (d-mapping oplen
                     (or-effect oplen
                                (operand oplen (cdr s&addr) s)
                                (read-dn oplen (d_rn ins) s)
                                (mc-ccr s))
                     (d_rn ins)
                     (car s&addr))))
    (halt (mode-signal) s)))

(defn or-mapping (sopd oplen ins s)
  (let ((s&addr (mc-instate oplen ins s)))
    (if (mc-haltp (car s&addr))
        (car s&addr)
      (mapping oplen
               (or-effect oplen
                          sopd
                          (operand oplen (cdr s&addr) s)
                          (mc-ccr s))
               s&addr))))

(defn or-ins2 (oplen ins s)
  (if (or-addr-modep2 ins)
      (or-mapping (read-dn oplen (d_rn ins) s)
                  oplen
                  ins
                  s)
    (halt (mode-signal) s)))

; DIVU.W/DIVS.W instructions.  D(32)/S(16) --> D(16r:16q).
; iquot and irem expect that s and d are unsigned integers.  They are
; used in the DIV instruction.
(defn iquot (n s i d j)
  (head (int-to-nat (iquotient (nat-to-int d j)
```

```
                              (nat-to-int s i))
                    j)
        n))

(defn irem (n s i d j)
  (head (int-to-nat (iremainder (nat-to-int d j)
                                (nat-to-int s i))
                    i)
        n))

; DIVS.W instruction.
(defn divs-v (n sopd i dopd j)
  (if (int-rangep (iquotient (nat-to-int dopd j) (nat-to-int sopd i))
                  n)
      (b0) (b1)))

(defn divs-z (n sopd i dopd j)
  (if (equal (iquot n sopd i dopd j) 0)
      (b1) (b0)))

(defn divs-n (n sopd i dopd j)
  (mbit (iquot n sopd i dopd j) n))

; In our specification of DIV,  we only make sure that the N and Z bits
; are set correctly when there is NO overflow.
; Since we test for overflow before this instruction is fully completed,
; the setting of CCR is actually the same as AND's if NO overflow occurs.
; When an overflow is detected, we simply halt the machine with an
; error signal.

; If overflow or divide by zero happens during the DIV instructions, then
; the MC68020 manual states that values of N, Z, and V are undefined.
; Thus one should not count on the validity of these values in the
; error state returned by stepi.

(defn divs-cvznx (n sopd i dopd j ccr)
  (cvznx (b0)
         (b0)
         (divs-z n sopd i dopd j)
         (divs-n n sopd i dopd j)
         (ccr-x ccr)))

; 32/16 --> 16r:16q.
(defn divs_w-ins (sopd dn s)
  (if (equal (nat-to-int sopd (w)) 0)
      (halt 'trap-exception s)
    (let ((dopd (read-dn (l) dn s)))
      (if (b0p (divs-v (w) sopd (w) dopd (l)))
          (update-ccr (divs-cvznx (w) sopd (w) dopd (l) (mc-ccr s))
                      (write-dn (l)
                                (app (w)
                                     (iquot (w) sopd (w) dopd (l))
                                     (irem (w) sopd (w) dopd (l)))
                                dn
```

```
                              s))
      (halt 'divs-overflow (update-ccr (set-v (b1) (mc-ccr s)) s))))))

; DIVU.W instruction.
(defn quot (n x y)
  (head (quotient y x) n))

(defn rem (n x y)
  (head (remainder y x) n))

; The condition codes for DIVU.
(defn divu-v (n sopd dopd)
  (if (lessp (quotient dopd sopd) (exp 2 n))
      (b0) (b1)))

(defn divu-z (n sopd dopd)
  (if (equal (quot n sopd dopd) 0)
      (b1) (b0)))

(defn divu-n (n sopd dopd)
  (mbit (quot n sopd dopd) n))

; Same treatment as divs-cvznx.
(defn divu-cvznx (n sopd dopd ccr)
  (cvznx (b0)
         (b0)
         (divu-z n sopd dopd)
         (divu-n n sopd dopd)
         (ccr-x ccr)))

; 32/16 --> 16r:16q.
(defn divu_w-ins (sopd dn s)
  (if (equal (nat-to-uint sopd) 0)
      (halt 'trap-exception s)
    (let ((dopd (read-dn (l) dn s)))
      (if (b0p (divu-v (w) sopd dopd))
          (update-ccr (divu-cvznx (w) sopd dopd (mc-ccr s))
                      (write-dn (l)
                                (app (w)
                                     (quot (w) sopd dopd)
                                     (rem (w) sopd dopd))
                                dn
                                s))
      (halt 'divu-overflow (update-ccr (set-v (b1) (mc-ccr s)) s))))))

(defn div_w-ins (ins s)
  (if (mul&div-addr-modep ins)
      (let ((s&addr (mc-instate (w) ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (let ((sopd (operand (w) (cdr s&addr) s)))
            (if (b0p (bitn ins 8))
                (divu_w-ins sopd (d_rn ins) (car s&addr))
              (divs_w-ins sopd (d_rn ins) (car s&addr))))))
```

```
    (halt (mode-signal) s)))

; Opcode 1000.
; The OR instruction group includes two instructions OR and DIVU.W/DIVS.W.
(defn or-group (oplen ins s)
  (if (equal oplen (q))
      (div_w-ins ins s)
    (if (b0p (bitn ins 8))
        (or-ins1 oplen ins s)
      (if (lessp (s_mode ins) 2)
          (halt 'sbcd-pack-unpk-unspecified s)
        (or-ins2 oplen ins s)))))

; Rotate operations.
; Rotate left cnt times.  len is supposed to be the length of x.
(defn rol (len x cnt)
  (let ((n (remainder cnt len)))
    (app n (tail x (difference len n)) (head x (difference len n)))))

; Rotate right cnt times.  len is supposed to be the length of x.
(defn ror (len x cnt)
  (let ((n (remainder cnt len)))
    (app (difference len n) (tail x n) (head x n))))

; For memory shift/rotate, only memory alterable addressing modes are allowed.
(defn s&r-addr-modep (ins)
  (and (memory-addr-modep (s_mode ins) (s_rn ins))
       (alterable-addr-modep (s_mode ins) (s_rn ins))))

; i-data returns a nonnegative integer.  In register shift/rotate, it is
; the shift/rotate cnt.  In ADDQ and SUBQ, it is the immediate data.
(defn i-data (n)
  (if (zerop n) 8 n))

(defn sr-cnt (ins s)
  (if (b0p (bitn ins 5))
      (i-data (d_rn ins))
    (remainder (read-dn (b) (d_rn ins) s) 64)))

; ROL and ROR instructions.
; We divide the ROL/ROR instruction into a few subinstructions.

; Register ROL instruction.
; The setting of cvznx-flags for ROL.
(defn rol-c (len x cnt)
  (if (equal cnt 0)
      (b0)
    (let ((n (remainder cnt len)))
      (if (zerop n)
          (bcar x)
        (bitn x (difference len n))))))

(defn rol-z (len x cnt)
  (if (equal x 0) (b1) (b0)))
```

```
(defn rol-n (len x cnt)
  (bitn x (sub1 (difference len (remainder cnt len)))))

(defn rol-cvznx (len opd cnt ccr)
  (cvznx (rol-c len opd cnt)
         (b0)
         (rol-z len opd cnt)
         (rol-n len opd cnt)
         (ccr-x ccr)))

(defn rol-effect (len opd cnt ccr)
  (cons (rol len opd cnt)
        (rol-cvznx len opd cnt ccr)))

(defn register-rol-ins (oplen ins s)
  (d-mapping oplen
             (rol-effect oplen
                         (read-dn oplen (s_rn ins) s)
                         (sr-cnt ins s)
                         (mc-ccr s))
             (s_rn ins)
             s))

; Register ROR instruction.
(defn ror-c (len x cnt)
  (if (equal cnt 0)
      (b0)
    (let ((n (remainder cnt len)))
      (if (equal n 0)
          (bitn x (sub1 len))
        (bitn x (sub1 n))))))

(defn ror-z (len opd cnt)
  (if (equal opd 0) (b1) (b0)))

(defn ror-n (len x cnt)
  (let ((n (remainder cnt len)))
    (if (zerop n)
        (bitn x (sub1 len))
      (bitn x (sub1 n)))))

(defn ror-cvznx (len opd cnt ccr)
  (cvznx (ror-c len opd cnt)
         (b0)
         (ror-z len opd cnt)
         (ror-n len opd cnt)
         (ccr-x ccr)))

(defn ror-effect (oplen opd cnt ccr)
  (cons (ror oplen opd cnt)
        (ror-cvznx oplen opd cnt ccr)))

(defn register-ror-ins (oplen ins s)
```

```
  (d-mapping oplen
             (ror-effect oplen
                         (read-dn oplen (s_rn ins) s)
                         (sr-cnt ins s)
                         (mc-ccr s))
             (s_rn ins)
             s))

; Memory ROL instruction.
; The operand size should be word, and the shift operation is one bit only.
(defn mem-rol-effect (opd ccr)
  (rol-effect (w) opd 1 ccr))

(defn mem-rol-ins (ins s)
  (if (s&r-addr-modep ins)
      (let ((s&addr (mc-instate (w) ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (mapping (w)
                   (mem-rol-effect (operand (w) (cdr s&addr) s)
                                   (mc-ccr s))
                   s&addr)))
    (halt (mode-signal) s)))

; Memory ROR instruction.
; The operand size should be word, and the shift operation is one bit only.
(defn mem-ror-effect (opd ccr)
  (ror-effect (w) opd 1 ccr))

(defn mem-ror-ins (ins s)
  (if (s&r-addr-modep ins)
      (let ((s&addr (mc-instate (w) ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (mapping (w)
                   (mem-ror-effect (operand (w) (cdr s&addr) s)
                                   (mc-ccr s))
                   s&addr)))
    (halt (mode-signal) s)))

; LSL and LSR instructions.
; We divided the LSL/LSR instruction into several subinstructions.

; Register LSL instruction.
(defn lsl-c (len opd cnt)
  (if (equal cnt 0)
      (b0)
    (if (lessp len cnt)
        (b0)
      (bitn opd (difference len cnt)))))

(defn lsl-z (len opd cnt)
  (if (equal (lsl len opd cnt) 0)
      (b1) (b0)))
```

```
(defn lsl-n (len opd cnt)
  (mbit (lsl len opd cnt) len))

(defn lsl-x (len opd cnt ccr)
  (if (equal cnt 0)
      (ccr-x ccr)
    (lsl-c len opd cnt)))

(defn lsl-cvznx (len opd cnt ccr)
  (cvznx (lsl-c len opd cnt)
         (b0)
         (lsl-z len opd cnt)
         (lsl-n len opd cnt)
         (lsl-x len opd cnt ccr)))

(defn lsl-effect (len opd cnt ccr)
  (cons (lsl len opd cnt)
        (lsl-cvznx len opd cnt ccr)))

(defn register-lsl-ins (oplen ins s)
  (d-mapping oplen
             (lsl-effect oplen
                         (read-dn oplen (s_rn ins) s)
                         (sr-cnt ins s)
                         (mc-ccr s))
             (s_rn ins)
             s))

; Register LSR instruction.
(defn lsr-c (len opd cnt)
  (if (equal cnt 0)
      (b0)
    (if (lessp len cnt)
        (b0)
      (bitn opd (sub1 cnt)))))

(defn lsr-z (len opd cnt)
  (if (equal (lsr opd cnt) 0)
      (b1) (b0)))

(defn lsr-n (len opd cnt)
  (mbit (lsr opd cnt) len))

(defn lsr-x (len opd cnt ccr)
  (if (equal cnt 0)
      (ccr-x ccr)
    (lsr-c len opd cnt)))

(defn lsr-cvznx (len opd cnt ccr)
  (cvznx (lsr-c len opd cnt)
         (b0)
         (lsr-z len opd cnt)
         (lsr-n len opd cnt)
```

```
        (lsr-x len opd cnt ccr)))

(defn lsr-effect (len opd cnt ccr)
  (cons (lsr opd cnt)
        (lsr-cvznx len opd cnt ccr)))

(defn register-lsr-ins (oplen ins s)
  (d-mapping oplen
             (lsr-effect oplen
                         (read-dn oplen (s_rn ins) s)
                         (sr-cnt ins s)
                         (mc-ccr s))
             (s_rn ins)
             s))

; Memory LSL instruction.
(defn mem-lsl-effect (opd ccr)
  (lsl-effect (w) opd 1 ccr))

(defn mem-lsl-ins (ins s)
  (if (s&r-addr-modep ins)
      (let ((s&addr (mc-instate (w) ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (mapping (w)
                   (mem-lsl-effect (operand (w) (cdr s&addr) s)
                                   (mc-ccr s))
                   s&addr)))
    (halt (mode-signal) s)))

; Memory LSR instruction.
(defn mem-lsr-effect (opd ccr)
  (lsr-effect (w) opd 1 ccr))

(defn mem-lsr-ins (ins s)
  (if (s&r-addr-modep ins)
      (let ((s&addr (mc-instate (w) ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (mapping (w)
                   (mem-lsr-effect (operand (w) (cdr s&addr) s)
                                   (mc-ccr s))
                   s&addr)))
    (halt (mode-signal) s)))

; ASL and ASR instructions.

; Register ASL instruction.
(defn asl-c (len opd cnt)
  (lsl-c len opd cnt))

(defn asl-v (len opd cnt)
  (if (int-rangep (nat-to-int opd len)
                  (difference len cnt))
```

```
      (b0) (b1)))

(defn asl-z (len opd cnt)
  (if (equal (asl len opd cnt) 0)
      (b1) (b0)))

(defn asl-n (len opd cnt)
  (mbit (asl len opd cnt) len))

(defn asl-x (len opd cnt ccr)
  (if (equal cnt 0)
      (ccr-x ccr)
    (asl-c len opd cnt)))

(defn asl-cvznx (len opd cnt ccr)
  (cvznx (asl-c len opd cnt)
         (asl-v len opd cnt)
         (asl-z len opd cnt)
         (asl-n len opd cnt)
         (asl-x len opd cnt ccr)))

(defn asl-effect (len opd cnt ccr)
  (cons (asl len opd cnt)
        (asl-cvznx len opd cnt ccr)))

(defn register-asl-ins (oplen ins s)
  (d-mapping oplen
             (asl-effect oplen
                         (read-dn oplen (s_rn ins) s)
                         (sr-cnt ins s)
                         (mc-ccr s))
             (s_rn ins)
             s))

; Register ASR instruction.
(defn asr-c (len opd cnt)
  (if (equal cnt 0)
      (b0)
    (if (lessp cnt len)
        (bitn opd (sub1 cnt))
      (bitn opd (sub1 len)))))

(defn asr-z (len opd cnt)
  (if (equal (asr len opd cnt) 0)
      (b1) (b0)))

(defn asr-n (len opd cnt)
  (mbit (asr len opd cnt) len))

(defn asr-x (len opd cnt ccr)
  (if (equal cnt 0)
      (ccr-x ccr)
    (asr-c len opd cnt)))
```

```
(defn asr-cvznx (len opd cnt ccr)
  (cvznx (asr-c len opd cnt)
         (b0)
         (asr-z len opd cnt)
         (asr-n len opd cnt)
         (asr-x len opd cnt ccr)))

(defn asr-effect (len opd cnt ccr)
  (cons (asr len opd cnt)
        (asr-cvznx len opd cnt ccr)))

(defn register-asr-ins (oplen ins s)
  (d-mapping oplen
             (asr-effect oplen
                         (read-dn oplen (s_rn ins) s)
                         (sr-cnt ins s)
                         (mc-ccr s))
             (s_rn ins)
             s))

; Memory ASL instruction.
(defn mem-asl-effect (opd ccr)
  (asl-effect (w) opd 1 ccr))

(defn mem-asl-ins (ins s)
  (if (s&r-addr-modep ins)
      (let ((s&addr (mc-instate (w) ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (mapping (w)
                   (mem-asl-effect (operand (w) (cdr s&addr) s)
                                   (mc-ccr s))
                   s&addr)))
    (halt (mode-signal) s)))

; Memory ASR instruction.
(defn mem-asr-effect (opd ccr)
  (asr-effect (w) opd 1 ccr))

(defn mem-asr-ins (ins s)
  (if (s&r-addr-modep ins)
      (let ((s&addr (mc-instate (w) ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (mapping (w)
                   (mem-asr-effect (operand (w) (cdr s&addr) s)
                                   (mc-ccr s))
                   s&addr)))
    (halt (mode-signal) s)))

; ROXL and ROXR instructions.

; roxl defines the rotate left with extend operation.
(defn roxl (len opd cnt x)
```

```
  (let ((temp (plus x (times 2 opd))))
    (bcdr (rol (add1 len) temp cnt))))

; roxr defines the rotate right with extend operation.
(defn roxr (len opd cnt x)
  (let ((temp (plus opd (times x (exp 2 len)))))
    (head (ror (add1 len) temp cnt) len)))

; Register ROXL instruction.
(defn roxl-c (len opd cnt x)
  (let ((tmp (remainder cnt (add1 len))))
    (if (equal tmp 0)
        (fix-bit x)
      (bitn opd (difference len tmp)))))

(defn roxl-z (len opd cnt x)
  (if (equal (roxl len opd cnt x) 0) (b1) (b0)))

(defn roxl-n (len opd cnt x)
  (bitn (roxl len opd cnt x) (sub1 len)))

(defn roxl-cvznx (len opd cnt x)
  (cvznx (roxl-c len opd cnt x)
         (b0)
         (roxl-z len opd cnt x)
         (roxl-n len opd cnt x)
         (roxl-c len opd cnt x)))

(defn roxl-effect (len opd cnt ccr)
  (cons (roxl len opd cnt (ccr-x ccr))
        (roxl-cvznx len opd cnt (ccr-x ccr))))

(defn register-roxl-ins (oplen ins s)
  (d-mapping oplen
             (roxl-effect oplen
                          (read-dn oplen (s_rn ins) s)
                          (sr-cnt ins s)
                          (mc-ccr s))
             (s_rn ins)
             s))

; Register ROXR instruction.
(defn roxr-c (len opd cnt x)
  (let ((tmp (remainder cnt (add1 len))))
    (if (equal tmp 0)
        (fix-bit x)
      (bitn opd (sub1 tmp)))))

(defn roxr-z (len opd cnt x)
  (if (equal (roxr len opd cnt x) 0) (b1) (b0)))

(defn roxr-n (len opd cnt x)
  (bitn (roxr len opd cnt x) (sub1 len)))
```

```
(defn roxr-cvznx (len opd cnt x)
  (cvznx (roxr-c len opd cnt x)
         (b0)
         (roxr-z len opd cnt x)
         (roxr-n len opd cnt x)
         (roxr-c len opd cnt x)))

(defn roxr-effect (len opd cnt ccr)
  (cons (roxr len opd cnt (ccr-x ccr))
        (roxr-cvznx len opd cnt (ccr-x ccr))))

(defn register-roxr-ins (oplen ins s)
  (d-mapping oplen
             (roxr-effect oplen
                          (read-dn oplen (s_rn ins) s)
                          (sr-cnt ins s)
                          (mc-ccr s))
             (s_rn ins)
             s))

; Memory ROXL instruction.
(defn mem-roxl-effect (opd ccr)
  (roxl-effect (w) opd 1 ccr))

(defn mem-roxl-ins (ins s)
  (if (s&r-addr-modep ins)
      (let ((s&addr (mc-instate (w) ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (mapping (w)
                   (mem-roxl-effect (operand (w) (cdr s&addr) s)
                                    (mc-ccr s))
                   s&addr)))
    (halt (mode-signal) s)))

; Memory ROXR instruction.
(defn mem-roxr-effect (opd ccr)
  (roxr-effect (w) opd 1 ccr))

(defn mem-roxr-ins (ins s)
  (if (s&r-addr-modep ins)
      (let ((s&addr (mc-instate (w) ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (mapping (w)
                   (mem-roxr-effect (operand (w) (cdr s&addr) s)
                                    (mc-ccr s))
                   s&addr)))
    (halt (mode-signal) s)))

; Memory shift/rotate.
(defn memory-shift-rotate (ins s)
  (if (b0p (bitn ins 10))
      (if (b0p (bitn ins 9))
```

```
          (if (b0p (bitn ins 8))
              (mem-asr-ins ins s)
            (mem-asl-ins ins s))
        (if (b0p (bitn ins 8))
            (mem-lsr-ins ins s)
          (mem-lsl-ins ins s)))
    (if (b0p (bitn ins 9))
        (if (b0p (bitn ins 8))
            (mem-roxr-ins ins s)
          (mem-roxl-ins ins s))
      (if (b0p (bitn ins 8))
          (mem-ror-ins ins s)
        (mem-rol-ins ins s)))))

; Register shift/rotate.
(defn register-shift-rotate (oplen ins s)
  (if (b0p (bitn ins 4))
      (if (b0p (bitn ins 3))
          (if (b0p (bitn ins 8))
              (register-asr-ins oplen ins s)
            (register-asl-ins oplen ins s))
        (if (b0p (bitn ins 8))
            (register-lsr-ins oplen ins s)
          (register-lsl-ins oplen ins s)))
    (if (b0p (bitn ins 3))
        (if (b0p (bitn ins 8))
            (register-roxr-ins oplen ins s)
          (register-roxl-ins oplen ins s))
      (if (b0p (bitn ins 8))
          (register-ror-ins oplen ins s)
        (register-rol-ins oplen ins s)))))

; The bit field instruction group consists of BFxxx instructions. All of
; these instructions are new in the MC68020.  Note that bit 15 in the extension
; word has to be 0!
(defn bf-subgroup (ins s)
  (halt 'i-will-do-it-later s))


; Opcode 1110.
; The shift/rotate instruction group includes the ASL/ASR, LSL/LSR, ROL/ROR,
; ROXL/RORL, BFTST, BFEXTU, BFCHG, BFEXTS, BFCLR, BFFFO, BFSET, and BFINS
; instructions.  But it actually divides into many varieties of these
; instructions.
(defn s&r-group (ins s)
  (if (equal (op-len ins) (q))
      (if (b0p (bitn ins 11))
          (memory-shift-rotate ins s)
        (bf-subgroup ins s))
    (register-shift-rotate (op-len ins) ins s)))

; MOVE instruction.
(defn move-addr-modep (oplen ins)
  (and (addr-modep (s_mode ins) (s_rn ins))
```

```
       (data-addr-modep (d_mode ins) (d_rn ins))
       (alterable-addr-modep (d_mode ins) (d_rn ins))
       (not (byte-an-direct-modep oplen (s_mode ins)))))

(defn move-z (oplen sopd)
  (if (equal (head sopd oplen) 0) (b1) (b0)))

(defn move-n (oplen sopd)
  (mbit sopd oplen))

; The definition of cvznx-flags of MOVE instruction.  It is also used in
; TST and TAS instructions.
(defn move-cvznx (oplen sopd ccr)
  (cvznx (b0)
         (b0)
         (move-z oplen sopd)
         (move-n oplen sopd)
         (ccr-x ccr)))

(defn move-effect (oplen sopd ccr)
  (cons sopd
        (move-cvznx oplen sopd ccr)))

(defn move-mapping (sopd oplen ins s)
  (let ((s&addr (effec-addr oplen (d_mode ins) (d_rn ins) s)))
    (if (mc-haltp (car s&addr))
        (car s&addr)
      (mapping oplen
               (move-effect oplen sopd (mc-ccr s))
               s&addr))))

(defn move-ins (oplen ins s)
  (if (move-addr-modep oplen ins)
      (let ((s&addr (mc-instate oplen ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (move-mapping (operand oplen (cdr s&addr) s)
                        oplen
                        ins
                        (car s&addr))))
    (halt (mode-signal) s)))

; MOVEA instruction.
; MOVEA differs from MOVE in several ways:  ccr is not affected and word
; operation is sign-extended.  Therefore, we define it separately.
(defn movea-addr-modep (ins)
  (addr-modep (s_mode ins) (s_rn ins)))

(defn movea-ins (oplen ins s)
  (if (movea-addr-modep ins)
      (let ((s&addr (mc-instate oplen ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (write-an (l)
```

```
                    (ext oplen (operand oplen (cdr s&addr) s) (l))
                    (d_rn ins)
                    (car s&addr))))
    (halt (mode-signal) s)))

; Opcode 0010 and 0011.
; The following definition is defined to distinguish MOVE and MOVEA
; instructions.  This definition is only for word and long operations.
(defn move-group (oplen ins s)
  (if (equal (d_mode ins) 1)
      (movea-ins oplen ins s)
    (move-ins oplen ins s)))

; LEA instruction.
(defn lea-addr-modep (ins)
  (control-addr-modep (s_mode ins) (s_rn ins)))

; lea-ins calls effec-addr, instead of mc-instate, since the effective
; address is JUST what we need.  Notice that LEA and PEA only deal with
; memory address.  The address direct modes are not allowed.
; Operation size: long.
(defn lea-ins (smode ins s)
  (if (lea-addr-modep ins)
      (let ((s&addr (effec-addr (l) smode (s_rn ins) s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (write-an (l)
                    (cddr s&addr)
                    (d_rn ins)
                    (car s&addr))))
    (halt (mode-signal) s)))

; EXTB instruction.
; Sign-extend a byte to a longword.  It is new in the MC68020.
(defn ext-z (oplen opd size)
  (if (equal (ext oplen opd size) 0)
      (b1) (b0)))

(defn ext-n (oplen opd size)
  (mbit (ext oplen opd size) size))

(defn ext-cvznx (oplen opd size ccr)
  (cvznx (b0)
         (b0)
         (ext-z oplen opd size)
         (ext-n oplen opd size)
         (ccr-x ccr)))

(defn ext-effect (oplen opd size ccr)
  (cons (ext oplen opd size)
        (ext-cvznx oplen opd size ccr)))

(defn extb-ins (ins s)
  (d-mapping (l)
```

```
            (ext-effect (b)
                        (read-dn (b) (s_rn ins) s)
                        (1)
                        (mc-ccr s))
            (s_rn ins)
            s))

; The LEA instruction subgroup includes LEA and EXTB instructions.
(defn lea-subgroup (ins s)
  (if (equal (s_mode ins) 0)
      (if (equal (bits ins 9 11) 4)
          (extb-ins ins s)
        (halt (reserved-signal) s))
    (lea-ins (s_mode ins) ins s)))

; CLR instruction.
(defn clr-addr-modep (ins)
  (and (data-addr-modep (s_mode ins) (s_rn ins))
       (alterable-addr-modep (s_mode ins) (s_rn ins))))

(defn clr-cvznx (ccr)
  (cvznx (b0) (b0) (b1) (b0) (ccr-x ccr)))

(defn clr-effect (ccr)
  (cons 0 (clr-cvznx ccr)))

(defn clr-ins (oplen ins s)
  (if (clr-addr-modep ins)
      (let ((s&addr (effec-addr oplen (s_mode ins) (s_rn ins) s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (mapping oplen
                   (clr-effect (mc-ccr s))
                   s&addr)))
    (halt (mode-signal) s)))

; MOVE from CCR instruction.
(defn move-from-ccr-addr-modep (ins)
  (and (data-addr-modep (s_mode ins) (s_rn ins))
       (alterable-addr-modep (s_mode ins) (s_rn ins))))

; This instruction has no effect on CCR. Therefore, the original CCR is
; copied for the updating.  This is intended to have a uniform treatment
; for cvznx-flags.  It makes it possible to use one theorem to characterize
; the action.
(defn move-from-ccr-effect (ccr)
  (cons ccr ccr))

(defn move-from-ccr-ins (ins s)
  (if (move-from-ccr-addr-modep ins)
      (let ((s&addr (mc-instate (w) ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (mapping (w)
```

```
                  (move-from-ccr-effect (mc-ccr s))
                  s&addr)))
    (halt (mode-signal) s)))

; The CLR instruction subgroup includes CLR and MOVE from CCR instructions.
(defn clr-subgroup (ins s)
  (if (equal (op-len ins) (q))
      (move-from-ccr-ins ins s)
    (clr-ins (op-len ins) ins s)))

; NEGX instruction.
(defn negx-addr-modep (ins)
  (and (data-addr-modep (s_mode ins) (s_rn ins))
       (alterable-addr-modep (s_mode ins) (s_rn ins))))

(defn negx-ins (oplen ins s)
  (if (negx-addr-modep ins)
      (let ((s&addr (mc-instate oplen ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (mapping oplen
                   (subx-effect oplen
                                (operand oplen (cdr s&addr) s)
                                0
                                (mc-ccr s))
                   s&addr)))
    (halt (mode-signal) s)))

; The NEGX instruction subgroup includes the NEGX instruction.
; Detect MOVE from SR.
(defn negx-subgroup (ins s)
  (if (equal (op-len ins) (q))
      (halt 'move-from-sr-privileged s)
    (negx-ins (op-len ins) ins s)))

; NEG instruction.
(defn neg-addr-modep (ins)
  (and (data-addr-modep (s_mode ins) (s_rn ins))
       (alterable-addr-modep (s_mode ins) (s_rn ins))))

(defn neg-ins (oplen ins s)
  (if (neg-addr-modep ins)
      (let ((s&addr (mc-instate oplen ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (mapping oplen
                   (sub-effect oplen
                               (operand oplen (cdr s&addr) s)
                               0)
                   s&addr)))
    (halt (mode-signal) s)))

; MOVE to CCR instruction.
(defn move-to-ccr-addr-modep (ins)
```

```
  (data-addr-modep (s_mode ins) (s_rn ins)))

(defn move-to-ccr-ins (ins s)
  (if (move-to-ccr-addr-modep ins)
      (let ((s&addr (mc-instate (w) ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (update-ccr (head (operand (w) (cdr s&addr) s) (b))
                      (car s&addr))))
    (halt (mode-signal) s)))

; The NEG instruction subgroup includes NEG and MOVE to CCR instructions.
(defn neg-subgroup (ins s)
  (if (equal (op-len ins) (q))
      (move-to-ccr-ins ins s)
    (neg-ins (op-len ins) ins s)))

; PEA instruction.
(defn pea-addr-modep (ins)
  (control-addr-modep (s_mode ins) (s_rn ins)))

(defn pea-ins (smode ins s)
  (if (pea-addr-modep ins)
      (let ((s&addr (effec-addr (l) smode (s_rn ins) s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (push-sp (lsz) (cddr s&addr) (car s&addr))))
    (halt (mode-signal) s)))

; SWAP instruction.
(defn swap-z (opd)
  (if (equal (fix opd) 0) (b1) (b0)))

(defn swap-n (opd)
  (bitn opd 15))

(defn swap-cvznx (opd ccr)
  (cvznx (b0)
         (b0)
         (swap-z opd)
         (swap-n opd)
         (ccr-x ccr)))

(defn swap-effect (opd ccr)
  (cons (app (w) (tail opd (w)) (head opd (w)))
        (swap-cvznx opd ccr)))

(defn swap-ins (ins s)
  (d-mapping (l)
             (swap-effect (read-dn (l) (s_rn ins) s)
                          (mc-ccr s))
             (s_rn ins)
             s))
```

```
; The PEA instruction subgroup includes PEA and SWAP.
; Detect BKPT.
(defn pea-subgroup (ins s)
  (if (lessp (s_mode ins) 2)
      (if (equal (s_mode ins) 0)
          (swap-ins ins s)
        (halt 'bkpt-unspecified s))
    (pea-ins (s_mode ins) ins s)))

; EXT instruction.
(defn ext-ins (ins s)
  (if (b0p (bitn ins 6))
      (d-mapping (w)
                 (ext-effect (b)
                             (read-dn (b) (s_rn ins) s)
                             (w)
                             (mc-ccr s))
                 (s_rn ins)
                 s)
    (d-mapping (l)
               (ext-effect (w)
                           (read-dn (w) (s_rn ins) s)
                           (l)
                           (mc-ccr s))
               (s_rn ins)
               s)))

; MOVEM Rn to EA instruction.
; A pair of functions for multiple read/write on memory.
(defn readm-mem (opsz addr mem n)
  (if (zerop n)
      nil
    (cons (read-mem addr mem opsz)
          (readm-mem opsz (add (l) addr opsz) mem (sub1 n)))))

(defn writem-mem (opsz vlst addr mem)
  (if (listp vlst)
      (writem-mem opsz
                  (cdr vlst)
                  (add (l) addr opsz)
                  (write-mem (car vlst) addr mem opsz))
    mem))

; A pair of functions for multiple read/write on the register file.
(defn readm-rn (oplen rnlst rfile)
  (if (listp rnlst)
      (cons (read-rn oplen (car rnlst) rfile)
            (readm-rn oplen (cdr rnlst) rfile))
    nil))

(defn writem-rn (oplen vlst rnlst rfile)
  (if (listp rnlst)
      (writem-rn oplen
                 (cdr vlst)
```

```
                (cdr rnlst)
                (write-rn (l) (ext oplen (car vlst) (l)) (car rnlst) rfile))
    rfile))

; A list of the number of registers to be moved.
(defn movem-rnlst (mask i)
  (if (zerop mask)
      nil
    (if (b0p (bcar mask))
        (movem-rnlst (bcdr mask) (add1 i))
      (cons i (movem-rnlst (bcdr mask) (add1 i))))))

(defn movem-len (mask)
  (if (zerop mask)
      0
    (if (b0p (bcar mask))
        (movem-len (bcdr mask))
      (add1 (movem-len (bcdr mask))))))

(defn writemp (mask oplen addr mem)
  (write-memp addr mem (times (op-sz oplen) (movem-len mask))))

; In the case of predecrement, there are a few things we have to treat
; separately.
; The order of the mask is the reverse of the other cases.
(defn movem-pre-rnlst (mask i lst)
  (if (zerop mask)
      lst
    (if (b0p (bcar mask))
        (movem-pre-rnlst (bcdr mask) (sub1 i) lst)
      (movem-pre-rnlst (bcdr mask) (sub1 i) (cons i lst)))))

; The reason we modify the address register rn here is that if it is also
; moved to memory, it is changed before it is moved.  This function returns
; a `cons':  the first element is an intermediate state with the address
; register rn changed, the second element is the starting memory address
; to move those registers.
(defn movem-predec (mask oplen rn s)
  (let ((addr (read-an (l) rn s)))
    (cons (write-an (l) (pre-effect oplen rn addr) rn s)
          (cons 'm (sub (l) (times (op-sz oplen) (movem-len mask)) addr)))))

; The addressing modes are control alterable plus predecrement.  We
; deal with -(An) separately.
(defn movem-rn-ea-addr-modep (ins)
  (and (alterable-addr-modep (s_mode ins) (s_rn ins))
       (control-addr-modep (s_mode ins) (s_rn ins))))

; Note in the predecrement mode, if mask = 0, there is no action on An.
(defn movem-rn-ea-ins (mask oplen ins s)
  (if (predec-modep (s_mode ins))
      (let ((s&addr (movem-predec mask oplen (s_rn ins) s)))
        (if (writemp mask oplen (cddr s&addr) (mc-mem s))
            (write-an (l)
```

```
                    (cddr s&addr)
                    (s_rn ins)
                    (update-mem
                     (writem-mem (op-sz oplen)
                                 (readm-rn oplen
                                           (movem-pre-rnlst mask 15 nil)
                                           (mc-rfile (car s&addr)))
                                 (cddr s&addr)
                                 (mc-mem s))
                     (car s&addr)))
          (halt (write-signal) s)))
    (if (movem-rn-ea-addr-modep ins)
        (let ((s&addr (effec-addr oplen (s_mode ins) (s_rn ins) s)))
          (if (mc-haltp (car s&addr))
              (car s&addr)
            (if (writemp mask oplen (cddr s&addr) (mc-mem s))
                (update-mem (writem-mem (op-sz oplen)
                                        (readm-rn oplen
                                                  (movem-rnlst mask 0)
                                                  (mc-rfile s))
                                        (cddr s&addr)
                                        (mc-mem s))
                            (car s&addr))
              (halt (write-signal) s))))
      (halt (mode-signal) s))))

; The EXT instruction subgroup includes EXT and MOVEM Rn to EA.
(defn ext-subgroup (ins s)
  (if (equal (s_mode ins) 0)
      (ext-ins ins s)
    (if (pc-word-readp (mc-pc s) (mc-mem s))
        (movem-rn-ea-ins (pc-word-read (mc-pc s) (mc-mem s))
                         (if (b0p (bitn ins 6)) (w) (l))
                         ins
                         (update-pc (add (l) (mc-pc s) (wsz)) s))
      (halt (pc-signal) s))))

; TST instruction.
; MC68020 and MC68000 differ about addressing modes.
(defn tst-addr-modep (oplen ins)
  (if (equal oplen (b))
      (data-addr-modep (s_mode ins) (s_rn ins))
    (addr-modep (s_mode ins) (s_rn ins))))

(defn tst-ins (oplen ins s)
  (if (tst-addr-modep oplen ins)
      (let ((s&addr (mc-instate oplen ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (update-ccr (move-cvznx oplen
                                  (operand oplen (cdr s&addr) s)
                                  (mc-ccr s))
                      (car s&addr))))
    (halt (mode-signal) s)))
```

```
; TAS instruction.
; It is usually used as a multiprocessor operation.
(defn tas-addr-modep (ins)
  (and (data-addr-modep (s_mode ins) (s_rn ins))
       (alterable-addr-modep (s_mode ins) (s_rn ins))))

(defn tas-effect (opd ccr)
  (cons (setn opd 7 (b1))
        (move-cvznx (b) opd ccr)))

; The opsize of the TAS instruction is byte.
(defn tas-ins (ins s)
  (if (tas-addr-modep ins)
      (let ((s&addr (mc-instate (b) ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (mapping (b)
                   (tas-effect (operand (b) (cdr s&addr) s)
                               (mc-ccr s))
                   s&addr)))
    (halt (mode-signal) s)))

; The TST instruction subgroup includes TAS and TST.
; Detect ILLEGAL instruction.
(defn tst-subgroup (ins s)
  (if (equal (op-len ins) (q))
      (if (equal (head ins 6) 60)
          (halt 'illegal-unspecified s)
        (tas-ins ins s))
    (tst-ins (op-len ins) ins s)))

; DIVS_L instructions.  D / S --> D.
; 32/32 --> 32q,  32/32 --> 32r:32q.  The order of write-dn:  remainder first,
; and then quotient.  The overflow happens only when the dopd is -2^31
; and sopd is -1.
(defn divsl_l (sopd dopd dq dr s)
  (if (b0p (divs-v (l) sopd (l) dopd (l)))
      (let ((q (iquot (l) sopd (l) dopd (l)))
            (r (irem (l) sopd (l) dopd (l))))
        (update-ccr (divs-cvznx (l) sopd (l) dopd (l) (mc-ccr s))
                    (write-dn (l) q dq (write-dn (l) r dr s))))
    (halt 'divs-overflow (update-ccr (set-v (b1) (mc-ccr s)) s))))

; 64/32 --> 32r:32q.
(defn divs_l (sopd dopd_low dq dr s)
  (let ((dopd (app (l) dopd_low (read-dn (l) dr s))))
    (if (b0p (divs-v (l) sopd (l) dopd (q)))
        (let ((q (iquot (l) sopd (l) dopd (q)))
              (r (irem (l) sopd (l) dopd (q))))
          (update-ccr (divs-cvznx (l) sopd (l) dopd (q) (mc-ccr s))
                      (write-dn (l) q dq (write-dn (l) r dr s))))
      (halt 'divs-overflow (update-ccr (set-v (b1) (mc-ccr s)) s)))))
```

```
; DIVU_L instructions.  D / S --> D.
; 32/32 --> 32q,  32/32 --> 32r:32q.  In this case, overflow never happens!
; It is justified by the event quotient-nat-rangep.
(defn divul_l (sopd dopd dq dr s)
  (let ((q (quot (l) sopd dopd))
        (r (rem (l) sopd dopd)))
    (update-ccr (divu-cvznx (l) sopd dopd (mc-ccr s))
                (write-dn (l) q dq (write-dn (l) r dr s)))))

; 64/32 --> 32r:32q.
(defn divu_l (sopd dopd_low dq dr s)
  (let ((dopd (app (l) dopd_low (read-dn (l) dr s))))
    (if (b0p (divu-v (l) sopd dopd))
        (let ((q (quot (l) sopd dopd))
              (r (rem (l) sopd dopd)))
          (update-ccr (divu-cvznx (l) sopd dopd (mc-ccr s))
                      (write-dn (l) q dq (write-dn (l) r dr s))))
      (halt 'divu-overflow (update-ccr (set-v (b1) (mc-ccr s)) s)))))

(defn dq (word)
  (bits word 12 14))

(defn dr (word)
  (bits word 0 2))

(defn div_l-ins (sopd word s)
  (let ((dopd_low (read-dn (l) (dq word) s)))
    (if (b0p (bitn word 11))
        (if (equal (nat-to-uint sopd) 0)
            (halt 'trap-exception s)
          (if (b0p (bitn word 10))
              (divul_l sopd dopd_low (dq word) (dr word) s)
            (divu_l sopd dopd_low (dq word) (dr word) s)))
      (if (equal (nat-to-int sopd (l)) 0)
          (halt 'trap-exception s)
        (if (b0p (bitn word 10))
            (divsl_l sopd dopd_low (dq word) (dr word) s)
          (divs_l sopd dopd_low (dq word) (dr word) s))))))

; MULS/MULU-long instructions.  S * D --> D.
(defn mulu_l_dl (sopd dopd dl s)
  (update-ccr (mulu-cvznx (l) sopd dopd (l) (mc-ccr s))
              (write-dn (l) (mulu (l) sopd dopd (l)) dl s)))

(defn mulu_l_dldh (sopd dopd dl dh s)
  (if (equal dl dh)
      (halt 'mc-undefined s)
    (update-ccr (mulu-cvznx (q) sopd dopd (l) (mc-ccr s))
                (write-dn (l)
                          (tail (mulu (q) sopd dopd (l)) (l))
                          dh
                          (write-dn (l)
                                    (head (mulu (q) sopd dopd (l)) (l))
                                    dl
```

```
                                    s)))))

(defn muls_l_dl (sopd dopd dl s)
  (update-ccr (muls-cvznx (l) sopd dopd (l) (mc-ccr s))
              (write-dn (l) (muls (l) sopd dopd (l)) dl s)))

(defn muls_l_dldh (sopd dopd dl dh s)
  (if (equal dl dh)
      (halt 'mc-undefined s)
    (update-ccr (muls-cvznx (q) sopd dopd (l) (mc-ccr s))
                (write-dn (l)
                          (tail (muls (q) sopd dopd (l)) (l))
                          dh
                          (write-dn (l)
                                    (head (muls (q) sopd dopd (l)) (l))
                                    dl
                                    s)))))

(defn dl (word)
  (bits word 12 14))

(defn dh (word)
  (bits word 0 2))

(defn mul_l-ins (sopd word s)
  (let ((dopd (read-dn (l) (dl word) s)))
    (if (b0p (bitn word 11))
        (if (b0p (bitn word 10))
            (mulu_l_dl sopd dopd (dl word) s)
          (mulu_l_dldh sopd dopd (dl word) (dh word) s))
      (if (b0p (bitn word 10))
          (muls_l_dl sopd dopd (dl word) s)
        (muls_l_dldh sopd dopd (dl word) (dh word) s)))))

(defn mul-div_l-ins (word ins s)
  (if (and (b0p (bitn word 15))
           (equal (bits word 3 9) 0))
      (if (mul&div-addr-modep ins)
          (let ((s&addr (mc-instate (l) ins s)))
            (if (mc-haltp (car s&addr))
                (car s&addr)
              (let ((sopd (operand (l) (cdr s&addr) (car s&addr))))
                (if (b0p (bitn ins 6))
                    (mul_l-ins sopd word (car s&addr))
                  (div_l-ins sopd word (car s&addr))))))
        (halt (mode-signal) s))
    (halt (reserved-signal) s)))

; MOVEM EA to RN instruction.
; The addressing modes are control plus postincrement.  We deal with
; (An)+ separately.
(defn movem-ea-rn-addr-modep (ins)
  (control-addr-modep (s_mode ins) (s_rn ins)))
```

```
(defn readmp (mask oplen addr mem)
  (read-memp addr mem (times (op-sz oplen) (movem-len mask))))

; In the mode of postincrement, if the address register is also loaded
; from the memory, the value of it upon completion of this instruction
; has no difference from the other modes.
(defn movem-ea-rn-ins (mask oplen ins s)
  (if (postinc-modep (s_mode ins))
      (let ((addr (read-an (l) (s_rn ins) s)))
        (if (readmp mask oplen addr (mc-mem s))
            (write-an (l)
                      (add (l) addr (times (op-sz oplen) (movem-len mask)))
                      (s_rn ins)
                      (update-rfile (writem-rn oplen
                                               (readm-mem (op-sz oplen)
                                                          addr
                                                          (mc-mem s)
                                                          (movem-len mask))
                                               (movem-rnlst mask 0)
                                               (mc-rfile s))
                                    s))
          (halt (read-signal) s)))
    (if (movem-ea-rn-addr-modep ins)
        (let ((s&addr (effec-addr oplen (s_mode ins) (s_rn ins) s)))
          (if (mc-haltp (car s&addr))
              (car s&addr)
            (if (readmp mask oplen (cddr s&addr) (mc-mem s))
                (update-rfile (writem-rn oplen
                                         (readm-mem (op-sz oplen)
                                                    (cddr s&addr)
                                                    (mc-mem s)
                                                    (movem-len mask))
                                         (movem-rnlst mask 0)
                                         (mc-rfile s))
                              (car s&addr))
              (halt (read-signal) s))))
      (halt (mode-signal) s))))

; The MOVEM-EA-RN-SUBGROUP includes MOVEM, DIVS/U and MULS/U instructions.
(defn movem-ea-rn-subgroup (ins s)
  (if (pc-word-readp (mc-pc s) (mc-mem s))
      (let ((word (pc-word-read (mc-pc s) (mc-mem s))))
        (if (b0p (bitn ins 7))
            (mul-div_l-ins word
                           ins
                           (update-pc (add (l) (mc-pc s) (wsz)) s))
          (movem-ea-rn-ins word
                           (if (b0p (bitn ins 6)) (w) (l))
                           ins
                           (update-pc (add (l) (mc-pc s) (wsz)) s))))
    (halt (pc-signal) s)))

; LINK-long instruction.
; LINK and UNLK are somewhat complicated.  When sp is used as an,  the
```

```
; execution order seems different from a simple instantiation.
(defn link-mapping (an disp s)
  (let ((sp (sub (l) (lsz) (read-sp s))))
    (if (write-memp sp (mc-mem s) (lsz))
        (update-mem (write-mem (read-an (l) an s) sp (mc-mem s) (lsz))
                    (write-sp (add (l) sp disp)
                              (write-an (l) sp an s)))
      (halt (write-signal) s))))

(defn link_l-ins (an s)
  (if (pc-long-readp (mc-pc s) (mc-mem s))
      (link-mapping an
                    (pc-long-read (mc-pc s) (mc-mem s))
                    (update-pc (add (l) (mc-pc s) (lsz)) s))
    (halt (pc-signal) s)))

; LINK-word instruction.
(defn link_w-ins (an s)
  (if (pc-word-readp (mc-pc s) (mc-mem s))
      (link-mapping an
                    (ext (w) (pc-word-read (mc-pc s) (mc-mem s)) (l))
                    (update-pc (add (l) (mc-pc s) (wsz)) s))
    (halt (pc-signal) s)))

; UNLK instruction.
(defn unlk-ins (an s)
  (let ((sp (read-an (l) an s)))
    (if (long-readp sp (mc-mem s))
        (write-an (l)
                  (long-read sp (mc-mem s))
                  an
                  (write-sp (add (l) sp (lsz)) s))
      (halt (read-signal) s))))

; The unlk instruction subgroup includes UNLK and LINK-word instructions.
; detect trap instruction.
(defn unlk-subgroup (ins s)
  (if (b0p (bitn ins 4))
      (halt 'trap-unspecified s)
    (if (b0p (bitn ins 3))
        (link_w-ins (s_rn ins) s)
      (unlk-ins (s_rn ins) s))))

; NOP instruction.
; The machine state, except the program counter, is not affected.  But
; we have already incremented pc when we read the first word of the
; current instruction.  Therefore, we simply return s.
(defn nop-ins (s) s)

; RTD instruction.
(defn rtd-mapping (sp disp s)
  (if (long-readp sp (mc-mem s))
      (let ((new-sp (add (l) (add (l) sp (lsz)) (ext (w) disp (l)))))
        (update-pc (long-read sp (mc-mem s))
```

```
                  (write-sp new-sp s)))
    (halt (read-signal) s)))

(defn rtd-ins (s)
  (if (pc-word-readp (mc-pc s) (mc-mem s))
      (rtd-mapping (read-sp s)
                   (pc-word-read (mc-pc s) (mc-mem s))
                   s)
    (halt (pc-signal) s)))

; RTS instruction.
; Notice that disp is 0.
(defn rts-ins (s)
  (rtd-mapping (read-sp s) 0 s))

; RTR instruction.
; Notice that disp is 0.
(defn rtr-ins (s)
  (let ((sp (read-sp s)))
    (if (word-readp sp (mc-mem s))
        (rtd-mapping (add (1) sp (wsz))
                     0
                     (update-ccr (word-read sp (mc-mem s)) s))
      (halt (read-signal) s))))

; TRAPV instruction.
; If the overflow is set, we simply halt the machine.  Otherwise, nop.
; To handle this instruction in verifications, we intend to prove the
; overflow is not set, and hence the machine performs nop.
(defn bvs (v) (fix-bit v))

(defn trapv-ins (s)
  (if (b1p (bvs (ccr-v (mc-ccr s))))
      (halt 'trapv-exception s)
    s))

; The NOP instruction subgroup includes NOP, RTD, RTS, and RTR instructions.
; Detect RESET, STOP, RTE, and TRAPV.
(defn nop-subgroup (ins s)
  (if (b0p (bitn ins 2))
      (if (b0p (bitn ins 1))
          (if (b0p (bitn ins 0))
              (halt 'reset-privileged s)
            (nop-ins s))
        (if (b0p (bitn ins 0))
            (halt 'stop-privileged s)
          (halt 'rte-privileged s)))
    (if (b0p (bitn ins 1))
        (if (b0p (bitn ins 0))
            (rtd-ins s)
          (rts-ins s))
      (if (b0p (bitn ins 0))
          (trapv-ins s)
        (rtr-ins s)))))
```

```
; JMP instruction.
; The JMP instruction is unsized. To calculate the effective address by
; effec-addr,  one can arbitrarily supply the operand length.  Note
; that the addr-predec, addr-postinc and immediate are not allowed.
(defn jmp-addr-modep (ins)
  (control-addr-modep (s_mode ins) (s_rn ins)))

; JMP does not affect CCR!
(defn jmp-mapping (addr s)
  (if (mc-haltp s)
      s
    (update-pc addr s)))

(defn jmp-ins (ins s)
  (if (jmp-addr-modep ins)
      (let ((s&addr (effec-addr (l) (s_mode ins) (s_rn ins) s)))
        (jmp-mapping (cddr s&addr) (car s&addr)))
    (halt (mode-signal) s)))

; JSR instruction.
; JSR does not affect CCR!
(defn jsr-addr-modep (ins)
  (control-addr-modep (s_mode ins) (s_rn ins)))

(defn jsr-ins (ins s)
  (if (jsr-addr-modep ins)
      (let ((s&addr (effec-addr (l) (s_mode ins) (s_rn ins) s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (jmp-mapping (cddr s&addr)
                       (push-sp (lsz)
                                (mc-pc (car s&addr))
                                (car s&addr)))))
    (halt (mode-signal) s)))

; NOT instruction.
(defn not-addr-modep (ins)
  (and (data-addr-modep (s_mode ins) (s_rn ins))
       (alterable-addr-modep (s_mode ins) (s_rn ins))))

(defn not-z (oplen opd)
  (if (equal (lognot oplen opd) 0) (b1) (b0)))

(defn not-n (oplen opd)
  (mbit (lognot oplen opd) oplen))

(defn not-cvznx (oplen opd ccr)
  (cvznx (b0)
         (b0)
         (not-z oplen opd)
         (not-n oplen opd)
         (ccr-x ccr)))
```

```
(defn not-effect (oplen opd ccr)
  (cons (lognot oplen opd)
        (not-cvznx oplen opd ccr)))

(defn not-ins (oplen ins s)
  (if (not-addr-modep ins)
      (let ((s&addr (mc-instate oplen ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (mapping oplen
                   (not-effect oplen
                               (operand oplen (cdr s&addr) s)
                               (mc-ccr s))
                   s&addr)))
    (halt (mode-signal) s)))

(defn not-subgroup (ins s)
  (if (equal (op-len ins) (q))
      (halt 'move-to-sr-privileged s)
    (not-ins (op-len ins) ins s)))

; Opcode 0100.
; The miscellaneous instruction group includes LEA, CLR, MOVE from CCR,
; NEG, MOVE to CCR, NOT, SWAP, PEA, EXT-word, MOVEM to EA, TST, TAS,
; MOVEM to RN, LINK, UNLK, NOP, RTD, RTS, RTR, JSR, JMP.
(defn misc-group (ins s)
  (if (b0p (bitn ins 8))
      (if (b0p (bitn ins 11))
          (if (b0p (bitn ins 10))
              (if (b0p (bitn ins 9))
                  (negx-subgroup ins s)                    ; 0000...
                (clr-subgroup ins s))                      ; 0010...
            (if (b0p (bitn ins 9))
                (neg-subgroup ins s)                       ; 0100...
              (not-subgroup ins s)))                       ; 0110
        (if (b0p (bitn ins 10))
            (if (b0p (bitn ins 9))                         ; 1000...
                (if (b0p (bitn ins 7))
                    (if (b0p (bitn ins 6))
                        (if (and (b0p (bitn ins 5))        ; 100000...
                                 (b0p (bitn ins 4))
                                 (b1p (bitn ins 3)))
                            (link_l-ins (s_rn ins) s)
                          (halt 'nbcd-unspecified s))
                      (pea-subgroup ins s))                ; 100001...
                  (ext-subgroup ins s))                    ; 10001...
              (tst-subgroup ins s))                        ; 1010...
          (if (b0p (bitn ins 9))
              (movem-ea-rn-subgroup ins s)                 ; 1100...
            (if (b0p (bitn ins 7))
                (if (b0p (bitn ins 6))
                    (halt (reserved-signal) s)
                  (if (b0p (bitn ins 5))
                      (unlk-subgroup ins s)                ; 1110010...
```

```
                  (if (b0p (bitn ins 4))
                      (halt 'move-usp-unspecified s)
                    (if (b0p (bitn ins 3))
                        (nop-subgroup ins s)             ; 111001110...
                      (halt 'movec-unspecified s)))))    ; 111001111
              (if (b0p (bitn ins 6))
                  (jsr-ins ins s)
                (jmp-ins ins s))))))
    (if (and (b1p (bitn ins 6))                          ; 0...
             (b1p (bitn ins 7)))
        (lea-subgroup ins s)
      (halt 'chk-unspecified s))))

; Some useful definitions for Bcc and Scc instruction groups.  Notice that
; bvs has been defined in TRAPV.
(defn bcs (c) (fix-bit c))

(defn beq (z) (fix-bit z))

(defn bmi (n) (fix-bit n))

(defn ble (v z n)
  (b-or z (b-or (b-and n (b-not v))
                (b-and (b-not n) v))))

(defn bgt (v z n)
  (b-and (b-or (b-and n v)
               (b-and (b-not n) (b-not v)))
         (b-not z)))

(defn blt (v n)
  (b-or (b-and n (b-not v))
        (b-and (b-not n) v)))

(defn bge (v n)
  (b-or (b-and n v)
        (b-and (b-not n) (b-not v))))

(defn bls (c z)
  (b-or c z))

(defn bhi (c z)
  (b-and (b-not c) (b-not z)))

(defn branch-cc (cond ccr)
  (if (lessp cond 8)
      (if (lessp cond 4)
          (if (lessp cond 2)
              (if (equal cond 0)
                  (b1)                                   ; 0000
                (b0))                                    ; 0001
            (if (equal cond 2)
                (bhi (ccr-c ccr) (ccr-z ccr))            ; 0010
              (bls (ccr-c ccr) (ccr-z ccr))))            ; 0011
```

```
        (if (lessp cond 6)
            (if (equal cond 4)
                (b-not (bcs (ccr-c ccr)))                  ; 0100
              (bcs (ccr-c ccr)))                           ; 0101
          (if (equal cond 6)
              (b-not (beq (ccr-z ccr)))                    ; 0110
            (beq (ccr-z ccr)))))                           ; 0111
    (if (lessp cond 12)
        (if (lessp cond 10)
            (if (equal cond 8)
                (b-not (bvs (ccr-v ccr)))                  ; 1000
              (bvs (ccr-v ccr)))                           ; 1001
          (if (equal cond 10)
              (b-not (bmi (ccr-n ccr)))                    ; 1010
            (bmi (ccr-n ccr))))                            ; 1011
      (if (lessp cond 14)
          (if (equal cond 12)
              (bge (ccr-v ccr) (ccr-n ccr))                ; 1100
            (blt (ccr-v ccr) (ccr-n ccr)))                 ; 1101
        (if (equal cond 14)
            (bgt (ccr-v ccr) (ccr-z ccr) (ccr-n ccr))      ; 1110
          (ble (ccr-v ccr) (ccr-z ccr) (ccr-n ccr))))))   ; 1111
)

; BSR instruction.
(defn bsr-ins (pc disp s)
  (push-sp (lsz)
           pc
           (update-pc (add (l) (mc-pc s) disp) s)))

; Bcc and BRA instructions are specified as follows.  The BSR
; instruction needs
; some auxiliary functions to specify it.
; We define BRA and Bcc together.
; Since 0000 is always true.
(defn bcc-ra-sr (pc cond disp s)
  (if (equal cond 0)
      (update-pc (add (l) (mc-pc s) disp) s)                ; BRA
    (if (equal cond 1)
        (bsr-ins pc disp s)                                 ; BSR
      (if (b0p (branch-cc cond (mc-ccr s)))                 ; Bcc
          (update-pc pc s)
        (update-pc (add (l) (mc-pc s) disp)  s)))))

; Opcode 0110.
; The Bcc instruction group includes Bcc, BRA and BSR instructions.
(defn bcc-group (disp ins s)
  (if (equal disp 0)               ; disp = 0.
      (if (pc-word-readp (mc-pc s) (mc-mem s))
            (bcc-ra-sr (add (l) (mc-pc s) (wsz))
                       (cond-field ins)
                       (ext (w) (pc-word-read (mc-pc s) (mc-mem s)) (l))
                       s)
          (halt (pc-signal) s))
```

```
      (if (equal disp 255)        ; disp = 2^8 - 1.
          (if (pc-long-readp (mc-pc s) (mc-mem s))
              (bcc-ra-sr (add (l) (mc-pc s) (lsz))
                         (cond-field ins)
                         (pc-long-read (mc-pc s) (mc-mem s))
                         s)
            (halt (pc-signal) s))
        (bcc-ra-sr (mc-pc s)
                   (cond-field ins)
                   (ext (b) disp (l))
                   s))))

; Scc instruction.
(defn scc-addr-modep (ins)
  (and (data-addr-modep (s_mode ins) (s_rn ins))
       (alterable-addr-modep (s_mode ins) (s_rn ins))))

; CCR is not affected by Scc.
(defn scc-effect (cond ccr)
  (cons (if (b0p (branch-cc cond ccr)) 0 255)
        ccr))

(defn scc-ins (ins s)
  (if (scc-addr-modep ins)
      (let ((s&addr (mc-instate (b) ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (mapping (b)
                   (scc-effect (cond-field ins) (mc-ccr s))
                   s&addr)))
    (halt (mode-signal) s)))

; DBcc instruction.
(defn dbcc-loop (rn s)
  (let ((cnt (sub (w) 1 (read-dn (w) rn s))))
    (if (equal (nat-to-int cnt (w)) -1)
        (update-pc (add (l) (mc-pc s) (wsz))
                   (write-dn (w) cnt rn s))
      (update-pc (add (l)
                      (mc-pc s)
                      (ext (w) (pc-word-read (mc-pc s) (mc-mem s)) (l)))
                 (write-dn (w) cnt rn s)))))

(defn dbcc-ins (ins s)
  (if (pc-word-readp (mc-pc s) (mc-mem s))
      (if (b0p (branch-cc (cond-field ins) (mc-ccr s)))
          (dbcc-loop (s_rn ins) s)
        (update-pc (add (l) (mc-pc s) (wsz)) s))
    (halt (pc-signal) s)))

; ADDQ instruction.
(defn addq-addr-modep (oplen ins)
  (and (alterable-addr-modep (s_mode ins) (s_rn ins))
       (not (byte-an-direct-modep oplen (s_mode ins)))))
```

```
; It seems to us that there is no
; difference between word and long word operations for the
; ADDQ instruction in the address register direct mode.

(defn addq-ins (oplen ins s)
  (if (addq-addr-modep oplen ins)
      (if (an-direct-modep (s_mode ins))
          (write-an (l)
                    (add (l)
                         (read-an (l) (s_rn ins) s)
                         (i-data (d_rn ins)))
                    (s_rn ins)
                    s)
        (add-mapping (i-data (d_rn ins)) oplen ins s))
    (halt (mode-signal) s)))

; SUBQ instruction.
; Same remark as for ADDQ.
(defn subq-addr-modep (oplen ins)
  (and (alterable-addr-modep (s_mode ins) (s_rn ins))
       (not (byte-an-direct-modep oplen (s_mode ins)))))

(defn subq-ins (oplen ins s)
  (if (subq-addr-modep oplen ins)
      (if (an-direct-modep (s_mode ins))
          (write-an (l)
                    (sub (l)
                         (i-data (d_rn ins))
                         (read-an (l) (s_rn ins) s))
                    (s_rn ins)
                    s)
        (sub-mapping (i-data (d_rn ins)) oplen ins s))
    (halt (mode-signal) s)))

; Opcode 0101.
; The Scc instruction group includes Scc, DBcc, ADDQ, and SUBQ instructions.
(defn scc-group (ins s)
  (if (equal (op-len ins) (q))
      (if (equal (s_mode ins) 1)
          (dbcc-ins ins s)
        (if (and (equal (s_mode ins) 7)
                 (lessp 1 (s_rn ins)))
            (halt 'trapcc-unspecified s)
          (scc-ins ins s)))
    (if (b0p (bitn ins 8))
        (addq-ins (op-len ins) ins s)
      (subq-ins (op-len ins) ins s))))

; Opcode 0111.
; MOVEQ instruction.
(defn moveq-ins (ins s)
  (if (b0p (bitn ins 8))
      (d-mapping (l)
```

```
                (move-effect (l)
                             (ext (b) (head ins (b)) (l))
                             (mc-ccr s))
                (d_rn ins)
                s)
    (halt (reserved-signal) s)))

; CMP instruction.
(defn cmp-cvznx (oplen sopd dopd ccr)
  (cvznx (sub-c oplen sopd dopd)
         (sub-v oplen sopd dopd)
         (sub-z oplen sopd dopd)
         (sub-n oplen sopd dopd)
         (ccr-x ccr)))               ; it is different from sub-x.

(defn cmp-addr-modep (oplen ins)
  (and (addr-modep (s_mode ins) (s_rn ins))
       (not (byte-an-direct-modep oplen (s_mode ins)))))

; The execution of the CMP instruction.
(defn cmp-ins (oplen ins s)
  (if (cmp-addr-modep oplen ins)
      (let ((s&addr (mc-instate oplen ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (update-ccr (cmp-cvznx oplen
                                 (operand oplen (cdr s&addr) (car s&addr))
                                 (read-dn oplen (d_rn ins) s)
                                 (mc-ccr s))
                      (car s&addr))))
    (halt (mode-signal) s)))

; CMPA instruction.
(defn cmpa-addr-modep (ins)
  (addr-modep (s_mode ins) (s_rn ins)))

; The cvznx-flag setting is the same as the CMP instruction.
; The only difference is that word operation is sign-extended to longword
; operation.
(defn cmpa-ins (oplen ins s)
  (if (cmpa-addr-modep ins)
      (let ((s&addr (mc-instate oplen ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (update-ccr (cmp-cvznx (l)
                                 (ext oplen
                                      (operand oplen (cdr s&addr) s)
                                      (l))
                                 (read-an (l) (d_rn ins) (car s&addr))
                                 (mc-ccr s))
                      (car s&addr))))
    (halt (mode-signal) s)))

; EOR instruction.
```

```
(defn eor-z (oplen sopd dopd)
  (if (equal (logeor sopd dopd) 0) (b1) (b0)))

(defn eor-n (oplen sopd dopd)
  (b-eor (mbit sopd oplen)
         (mbit dopd oplen)))

(defn eor-cvznx (oplen sopd dopd ccr)
  (cvznx (b0)
         (b0)
         (eor-z oplen sopd dopd)
         (eor-n oplen sopd dopd)
         (ccr-x ccr)))

(defn eor-effect (oplen sopd dopd ccr)
  (cons (logeor sopd dopd)
        (eor-cvznx oplen sopd dopd ccr)))

(defn eor&eori-addr-modep (ins)
  (and (data-addr-modep (s_mode ins) (s_rn ins))
       (alterable-addr-modep (s_mode ins) (s_rn ins))))

(defn eor-mapping (sopd oplen ins s)
  (let ((s&addr (mc-instate oplen ins s)))
    (if (mc-haltp (car s&addr))
        (car s&addr)
      (mapping oplen
               (eor-effect oplen
                           sopd
                           (operand oplen (cdr s&addr) s)
                           (mc-ccr s))
               s&addr))))

(defn eor-ins (oplen ins s)
  (if (eor&eori-addr-modep ins)
      (eor-mapping (read-dn oplen (d_rn ins) s)
                   oplen
                   ins
                   s)
    (halt (mode-signal) s)))

; CMPM instruction.
(defn cmpm-mapping (addr oplen ins s)
  (let ((s&addr (addr-postinc oplen (d_rn ins) s)))
    (if (read-memp (cddr s&addr) (mc-mem s) (op-sz oplen))
        (update-ccr (cmp-cvznx oplen
                               (operand oplen addr s)
                               (operand oplen (cdr s&addr) s)
                               (mc-ccr s))
                    (car s&addr))
      (halt (read-signal) s))))

(defn cmpm-ins (oplen ins s)
  (let ((s&addr (addr-postinc oplen (s_rn ins) s)))
```

```
    (if (read-memp (cddr s&addr) (mc-mem s) (op-sz oplen))
        (cmpm-mapping (cdr s&addr) oplen ins (car s&addr))
      (halt (read-signal) s))))

; Opcode 1011.
; The CMP instruction group includes instructions CMP, CMPA, EOR, and CMPM.
(defn cmp-group (oplen ins s)
  (if (b0p (bitn ins 8))
      (if (equal oplen (q))
          (cmpa-ins (w) ins s)
        (cmp-ins oplen ins s))
    (if (equal oplen (q))
        (cmpa-ins (l) ins s)
      (if (equal (s_mode ins) 1)
          (cmpm-ins oplen ins s)
        (eor-ins oplen ins s)))))

; MOVEP instruction.
; MOVEP moves a data register into alternate bytes of memory.
(defn movep-writep (addr mem n)
  (if (zerop n)
      t
    (and (byte-writep (add (l) addr (times 2 (sub1 n))) mem)
         (movep-writep addr mem (sub1 n)))))

(defn movep-write (value addr mem n)
  (if (zerop n)
      mem
    (movep-write (tail value (b))
                 addr
                 (byte-write value (add (l) addr (times 2 (sub1 n))) mem)
                 (sub1 n))))

(defn movep-to-mem (addr oplen ins s)
  (if (movep-writep addr (mc-mem s) (op-sz oplen))
      (update-mem (movep-write (read-dn oplen (d_rn ins) s)
                               addr
                               (mc-mem s)
                               (op-sz oplen))
                  s)
    (halt (write-signal) s)))

; MOVEP moves alternate bytes in memory into a data register.
(defn movep-readp (addr mem n)
  (if (zerop n)
      t
    (and (byte-readp addr mem)
         (movep-readp (add (l) addr (wsz)) mem (sub1 n)))))

(defn movep-read (addr mem n)
  (if (zerop n)
      0
    (app (b)
         (byte-read (add (l) addr (times 2 (sub1 n))) mem)
```

```
        (movep-read addr mem (sub1 n)))))

(defn movep-to-reg (addr oplen ins s)
  (if (movep-readp addr (mc-mem s) (op-sz oplen))
      (write-dn oplen
                (movep-read addr (mc-mem s) (op-sz oplen))
                (d_rn ins)
                s)
    (halt (read-signal) s)))

(defn movep-ins (opmode ins s)
  (let ((s&addr (addr-disp (mc-pc s) (s_rn ins) s)))
    (if (mc-haltp (car s&addr))
        (car s&addr)
      (if (lessp opmode 6)
          (if (equal opmode 4)
              (movep-to-reg (cddr s&addr) (w) ins (car s&addr))
            (movep-to-reg (cddr s&addr) (l) ins (car s&addr)))
        (if (equal opmode 6)
            (movep-to-mem (cddr s&addr) (w) ins (car s&addr))
          (movep-to-mem (cddr s&addr) (l) ins (car s&addr)))))))

; Some functions for bit operations.
(defn bxxx-oplen (smode)
  (if (dn-direct-modep smode) (l) (b)))

(defn bxxx-num (smode bnum)
  (if (dn-direct-modep smode)
      (head bnum 5)
    (head bnum 3)))

(defn bxxx-opd (smode s&addr)
  (if (dn-direct-modep smode)
      (read-dn (l) (cddr s&addr) (car s&addr))
    (operand (b) (cdr s&addr) (car s&addr))))

; BCHG instruction.
(defn bchg-addr-modep (ins)
  (and (data-addr-modep (s_mode ins) (s_rn ins))
       (alterable-addr-modep (s_mode ins) (s_rn ins))))

(defn bchg-effect (bnum opd ccr)
  (cons (setn opd bnum (b-not (bitn opd bnum)))
        (set-z (b-not (bitn opd bnum)) ccr)))

(defn bchg-ins (bnum ins s)
  (if (bchg-addr-modep ins)
      (let ((s&addr (mc-instate (b) ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (mapping (bxxx-oplen (s_mode ins))
                   (bchg-effect (bxxx-num (s_mode ins) bnum)
                                (bxxx-opd (s_mode ins) s&addr)
                                (mc-ccr s))
```

```
                  s&addr)))
    (halt (mode-signal) s)))

; BCLR instruction.
(defn bclr-addr-modep (ins)
  (and (data-addr-modep (s_mode ins) (s_rn ins))
       (alterable-addr-modep (s_mode ins) (s_rn ins))))

(defn bclr-effect (bnum opd ccr)
  (cons (setn opd bnum (b0))
        (set-z (b-not (bitn opd bnum)) ccr)))

(defn bclr-ins (bnum ins s)
  (if (bclr-addr-modep ins)
      (let ((s&addr (mc-instate (b) ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (mapping (bxxx-oplen (s_mode ins))
                   (bclr-effect (bxxx-num (s_mode ins) bnum)
                                (bxxx-opd (s_mode ins) s&addr)
                                (mc-ccr s))
                   s&addr)))
    (halt (mode-signal) s)))

; BSET instruction.
(defn bset-addr-modep (ins)
  (and (data-addr-modep (s_mode ins) (s_rn ins))
       (alterable-addr-modep (s_mode ins) (s_rn ins))))

(defn bset-effect (bnum opd ccr)
  (cons (setn opd bnum (b1))
        (set-z (b-not (bitn opd bnum)) ccr)))

(defn bset-ins (bnum ins s)
  (if (bset-addr-modep ins)
      (let ((s&addr (mc-instate (b) ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
          (mapping (bxxx-oplen (s_mode ins))
                   (bset-effect (bxxx-num (s_mode ins) bnum)
                                (bxxx-opd (s_mode ins) s&addr)
                                (mc-ccr s))
                   s&addr)))
    (halt (mode-signal) s)))

; BTST instruction.
(defn btst-addr-modep (ins)
  (data-addr-modep (s_mode ins) (s_rn ins)))

(defn btst-ins (bnum ins s)
  (if (btst-addr-modep ins)
      (let ((s&addr (mc-instate (b) ins s)))
        (if (mc-haltp (car s&addr))
            (car s&addr)
```

```
          (update-ccr (set-z (b-not (bitn (bxxx-opd (s_mode ins) s&addr)
                                          (bxxx-num (s_mode ins) bnum)))
                             (mc-ccr s))
                      (car s&addr))))
    (halt (mode-signal) s)))

; bit-ins includes the BTST, BCLR, BCHG, and BSET instructions.
(defn bit-ins (bnum ins s)
  (let ((type (bits ins 6 7)))
    (if (lessp type 2)
        (if (equal type 0)
            (btst-ins bnum ins s)
          (bchg-ins bnum ins s))
      (if (equal type 2)
          (bclr-ins bnum ins s)
        (bset-ins bnum ins s)))))

; Dynamic bit operation.  BTST, BCLR, BCHG, and BSET instructions.
(defn d-bit-subgroup (ins s)
  (if (equal (s_mode ins) 1)
      (movep-ins (opmode-field ins) ins s)
    (bit-ins (read-dn (l) (d_rn ins) s) ins s)))

; Static bit operation.  BTST, BCLR, BCHG, and BSET instructions.
(defn s-bit-subgroup (ins s)
  (if (pc-word-readp (mc-pc s) (mc-mem s))
      (if (equal (pc-byte-read (mc-pc s) (mc-mem s)) 0)
          (bit-ins (pc-byte-read (add (l) (mc-pc s) (bsz)) (mc-mem s))
                   ins
                   (update-pc (add (l) (mc-pc s) (wsz)) s))
        (halt (reserved-signal) s))
    (halt (pc-signal) s)))

; ORI instruction.
(defn ori-addr-modep (ins)
  (and (data-addr-modep (s_mode ins) (s_rn ins))
       (alterable-addr-modep (s_mode ins) (s_rn ins))))

(defn ori-ins (oplen ins s)
  (let ((s&idata (immediate oplen (mc-pc s) s)))
    (if (mc-haltp (car s&idata))
        (car s&idata)
      (if (ori-addr-modep ins)
          (or-mapping (cddr s&idata) oplen ins (car s&idata))
        (halt (mode-signal) s)))))

; ORI to CCR instruction.
(defn ori-to-ccr-ins (pc s)
  (if (pc-word-readp pc (mc-mem s))
      (if (equal (pc-byte-read pc (mc-mem s)) 0)
          (update-ccr (logor (pc-byte-read (add (l) pc (bsz)) (mc-mem s))
                             (mc-ccr s))
                      (update-pc (add (l) pc (wsz)) s))
        (halt (reserved-signal) s))
```

```
    (halt (pc-signal) s)))

; ORI and ORI to CCR instructions.
; Detect ORI to SR, CMP2, and CHK2.
(defn ori-subgroup (oplen ins s)
  (if (equal oplen (q))
      (halt 'cmp2-chk2-unspecified s)
    (if (equal (head ins 6) 60)
        (if (equal oplen (b))
            (ori-to-ccr-ins (mc-pc s) s)
          (if (equal oplen (w))
              (halt 'ori-to-sr-privileged s)
            (halt (reserved-signal) s)))
      (ori-ins oplen ins s))))

; ANDI instruction.
(defn andi-addr-modep (ins)
  (and (data-addr-modep (s_mode ins) (s_rn ins))
       (alterable-addr-modep (s_mode ins) (s_rn ins))))

(defn andi-ins (oplen ins s)
  (let ((s&idata (immediate oplen (mc-pc s) s)))
    (if (mc-haltp s)
        (car s&idata)
      (if (andi-addr-modep ins)
          (and-mapping (cddr s&idata) oplen ins (car s&idata))
        (halt (mode-signal) s)))))

; ANDI to CCR instruction.
(defn andi-to-ccr-ins (pc s)
  (if (pc-word-readp pc (mc-mem s))
      (if (equal (pc-byte-read pc (mc-mem s)) 0)
          (update-ccr (logand (pc-byte-read (add (1) pc (bsz)) (mc-mem s))
                              (mc-ccr s))
                      (update-pc (add (1) pc (wsz)) s))
        (halt (reserved-signal) s))
    (halt (pc-signal) s)))

; ANDI and ANDI to CCR instructions.
; Detect ANDI to SR, CMP2 and CHK2.
(defn andi-subgroup (oplen ins s)
  (if (equal oplen (q))
      (halt 'cmp2-chk2-unspecified s)
    (if (equal (head ins 6) 60)
        (if (equal oplen (b))
            (andi-to-ccr-ins (mc-pc s) s)
          (if (equal oplen (w))
              (halt 'andi-to-sr-unspecified s)
            (halt (reserved-signal) s)))
      (andi-ins oplen ins s))))

; SUBI instruction. Detect CMP2 and CHK2.
(defn subi-addr-modep (ins)
  (and (data-addr-modep (s_mode ins) (s_rn ins))
```

```
       (alterable-addr-modep (s_mode ins) (s_rn ins))))

(defn subi-ins (oplen ins s)
  (let ((s&idata (immediate oplen (mc-pc s) s)))
    (if (mc-haltp (car s&idata))
        (car s&idata)
      (if (subi-addr-modep ins)
          (sub-mapping (cddr s&idata) oplen ins (car s&idata))
        (halt (mode-signal) s)))))

(defn subi-subgroup (oplen ins s)
  (if (equal oplen (q))
      (halt 'cmp2-chk2-unspecified s)
    (subi-ins oplen ins s)))

; ADDI instruction.  Detect RTM and CALLM.
(defn addi-addr-modep (ins)
  (and (data-addr-modep (s_mode ins) (s_rn ins))
       (alterable-addr-modep (s_mode ins) (s_rn ins))))

(defn addi-ins (oplen ins s)
  (let ((s&idata (immediate oplen (mc-pc s) s)))
    (if (mc-haltp (car s&idata))
        (car s&idata)
      (if (addi-addr-modep ins)
          (add-mapping (cddr s&idata) oplen ins (car s&idata))
        (halt (mode-signal) s)))))

(defn addi-subgroup (oplen ins s)
  (if (equal oplen (q))
      (halt 'rtm-callm-unspecified s)
    (addi-ins oplen ins s)))

; EORI instruction.
(defn eori-ins (oplen ins s)
  (let ((s&idata (immediate oplen (mc-pc s) s)))
    (if (mc-haltp (car s&idata))
        (car s&idata)
      (if (eor&eori-addr-modep ins)
          (eor-mapping (cddr s&idata) oplen ins (car s&idata))
        (halt (mode-signal) s)))))

; EORI to CCR instruction.
(defn eori-to-ccr-ins (pc s)
  (if (pc-word-readp pc (mc-mem s))
      (if (equal (pc-byte-read pc (mc-mem s)) 0)
          (update-ccr (logeor (pc-byte-read (add (1) pc (bsz)) (mc-mem s))
                              (mc-ccr s))
                      (update-pc (add (1) pc (wsz)) s))
        (halt (reserved-signal) s))
    (halt (pc-signal) s)))

; EORI and EORI to CCR instructions.
; Detect EORI to SR, CAS and CAS2 instructions!
```

```
(defn eori-subgroup (oplen ins s)
  (if (equal oplen (q))
      (halt 'cas-cas2-unspecified s)
    (if (equal (head ins 6) 60)
        (if (equal oplen (b))
            (eori-to-ccr-ins (mc-pc s) s)
          (if (equal oplen (w))
              (halt 'eori-to-sr-unspecified s)
            (halt (reserved-signal) s)))
      (eori-ins oplen ins s))))

; CMPI instruction.
(defn cmpi-addr-modep (ins)
  (and (data-addr-modep (s_mode ins) (s_rn ins))
       (not (idata-modep (s_mode ins) (s_rn ins)))))

(defn cmpi-mapping (idata oplen ins s)
  (let ((s&addr (mc-instate oplen ins s)))
    (if (mc-haltp (car s&addr))
        (car s&addr)
      (update-ccr (cmp-cvznx oplen
                             idata
                             (operand oplen (cdr s&addr) s)
                             (mc-ccr s))
                  (car s&addr)))))

(defn cmpi-ins (oplen ins s)
  (let ((s&idata (immediate oplen (mc-pc s) s)))
    (if (mc-haltp (car s&idata))
        (car s&idata)
      (if (cmpi-addr-modep ins)
          (cmpi-mapping (cddr s&idata) oplen ins (car s&idata))
        (halt (mode-signal) s)))))

; The CMPI subgroup includes only the CMPI instruction.
; Detect CAS and CAS2 instructions!
(defn cmpi-subgroup (oplen ins s)
  (if (equal oplen (q))
      (halt 'cas-cas2-unspecified s)
    (cmpi-ins oplen ins s)))

; Opcode 0000.
; This instruction group includes instructions ORI, ORI to CCR, BTST, BCLR,
; BCHG, BSET, MOVEP, ANDI, ANDI to CCR, SUBI, ADDI, EORI, EORI to CCR, CMPI.
(defn bit-group (ins s)
  (if (b0p (bitn ins 8))
      (if (b0p (bitn ins 11))
          (if (b0p (bitn ins 10))
              (if (b0p (bitn ins 9))
                  (ori-subgroup (op-len ins) ins s)
                (andi-subgroup (op-len ins) ins s))
            (if (b0p (bitn ins 9))
                (subi-subgroup (op-len ins) ins s)
              (addi-subgroup (op-len ins) ins s)))
```

```
        (if (b0p (bitn ins 10))
            (if (b0p (bitn ins 9))
                (s-bit-subgroup ins s)
              (eori-subgroup (op-len ins) ins s))
          (if (b0p (bitn ins 9))
              (cmpi-subgroup (op-len ins) ins s)
            (halt 'moves-cas-cas2-unspecified s))))
    (d-bit-subgroup ins s)))

; The opcode field.
(defn opcode-field (ins)
  (bits ins 12 15))

; Execute the current instruction.  See Table 3-14 of [41]
; for this classification.
(defn execute-ins (ins s)
  (let ((opcode (opcode-field ins)))
    (if (lessp opcode 8)
        (if (lessp opcode 4)
            (if (lessp opcode 2)
                (if (equal opcode 0)
                    (bit-group ins s)                       ; 0000
                  (move-ins (b) ins s))                     ; 0001
              (if (equal opcode 2)
                  (move-group (l) ins s)                    ; 0010
                (move-group (w) ins s)))                    ; 0011
          (if (lessp opcode 6)
              (if (equal opcode 4)
                  (misc-group ins s)                        ; 0100
                (scc-group ins s))                          ; 0101
            (if (equal opcode 6)
                (bcc-group (head ins (b)) ins s)            ; 0110
              (moveq-ins ins s))))                          ; 0111
      (if (lessp opcode 12)
          (if (lessp opcode 10)
              (if (equal opcode 8)
                  (or-group (op-len ins) ins s)             ; 1000
                (sub-group (opmode-field ins) ins s))       ; 1001
            (if (equal opcode 10)
                (halt (reserved-signal) s)                  ; 1010
              (cmp-group (op-len ins) ins s)))              ; 1011
        (if (lessp opcode 14)
            (if (equal opcode 12)
                (and-group (op-len ins) ins s)              ; 1100
              (add-group (opmode-field ins) ins s))         ; 1101
          (if (equal opcode 14)
              (s&r-group ins s)                             ; 1110
            (halt 'coprocessor-unspecified s))))))          ; 1111
 )

; current-ins is a function of two arguments, pc and s.  pc is the
; current value of the program counter, and s is the current state.
; current-ins returns the current instruction (a word, not including
; any possible extension words), that is, the word pointed to by pc.
```

```
; To determine what instruction we are to execute, this word may only
; provide partial information.  Many instructions require that we
; examine subsequent words to determine what to do.  But to figure out
; how many words we need, we must start with the first word.
(defn current-ins (pc s)
  (pc-word-read pc (mc-mem s)))

;                  Stepi and Stepn
; Word-aligned means that the least significant bit is 0.
(defn evenp (x)
  (b0p (bcar x)))

; stepi maps a machine state to the next machine state by executing
; the current instruction.
(defn stepi (s)
  (if (evenp (mc-pc s))
      (if (pc-word-readp (mc-pc s) (mc-mem s))
          (execute-ins (current-ins (mc-pc s) s)
                       (update-pc (add (1) (mc-pc s) (wsz)) s))
        (halt (pc-signal) s))
    (halt (pc-odd-signal) s)))

; stepn is a function of two arguments: s is the current state of the
; machine, and n is the number of instructions to execute.
(defn stepn (s n)
  (if (or (mc-haltp s) (zerop n))
      s
    (stepn (stepi s) (sub1 n)))
  ((lessp (count n))))

;                  Auxiliary Functions
; This section contains some auxiliary functions which are not needed
; to define stepn but are used only in the example of the next section.
; map-update updates the map in the memory.  The map is a binary tree
; with a list of keys in the key field.  By updating the map we assign
; new properties to the memory.
(defn cons-key-lst (key lst)
  (if (member key lst)
      lst
    (cons key lst)))

(defn key-field (map)
  (if (listp map) (car map) nil))

(defn make-map (key map)
  (make-bt (cons-key-lst key (key-field map))
           (branch0 map)
           (branch1 map)))

(defn map-update (key x n map)
  (if (zerop n)
      (make-map key map)
    (if (b0p (bitn x (sub1 n)))
        (make-bt (key-field map)
```

```
                (map-update key x (sub1 n) (branch0 map))
                (branch1 map))
      (make-bt (key-field map)
               (branch0 map)
               (map-update key x (sub1 n) (branch1 map))))))

; Load the values in the list into the memory starting from location addr.
(defn load-lst-mem (opsz lst addr mem)
  (if (listp lst)
      (load-lst-mem opsz
                    (cdr lst)
                    (add 32 addr opsz)
                    (write-mem (car lst) addr mem opsz))
    mem))

(compile-uncompiled-defns "tmp")

;                An Example of Simulation

; Here is an utterly concrete theorem about stepn.  Roughly speaking, the
; theorem states that if stepn executes 37 instructions starting in a state
; that contains machine code instructions for Euclid's GCD algorithm in ROM and
; the integers 54 and 42 on the stack, then the correct answer, 6, is the
; value of data register d0 in the resulting state.  This theorem has, of
; course, an utterly trivial proof: we just run stepn .  We present this
; trivial theorem here only to illustrate setting up stepn to run.

(prove-lemma gcd-example nil
  (implies (and
            (equal stack-pointer #XEFFFE40)
            (equal rfile '(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 #XEFFFE4C #XEFFFE40))
            (equal pc #X22B6)
            (equal ccr 0)
            (equal gcd-code
                   '(78      86      0       0       72      231     48      0
                     36      46      0       8       38      46      0       12
                     74      130     103     28      74      131     102     4
                     32      2       96      22      182     130     108     8
                     76      67      40      0       36      0       96      232
                     76      66      56      0       38      0       96      224
                     32      3       76      238     0       12      255     248
                     78      94      78      117))
            (equal empty-memory '((NIL (NIL (NIL (NIL (NIL ((ROM) NIL)))))))))
            (equal mem (load-lst-mem 4 '(#X22B2 54 42) stack-pointer
                                (load-lst-mem 1 gcd-code pc empty-memory)))
            (equal initial-state (mc-state 'running rfile pc ccr mem))
            (equal final-state (stepn initial-state 37)))
           (and (equal (mc-status final-state) 'running)
                (equal (mc-rfile final-state)
                       '(6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 #XEFFFE4C #XEFFFE44))
                (equal (mc-pc final-state) #X22B2))))

; Here is a paraphrase of the foregoing theorem.  The specific numbers in
; the theorem are derived from the compilation of a C program for GCD and
```

```
; from the result of loading that program on a Sun-3.
; If
;  1. stack-pointer = #XEFFFE40,
;  2. the register file rfile is all 0's excepting for A6 and SP, which
;     are #XEFFFE4C and stack-pointer, respectively,
;  3. the program counter pc = #X22B6 and the condition code register ccr = 0,
;  4. gcd-code is the long list of integers above beginning with 78 and
;     ending with 117,
;  5. empty-memory is a pair representing a 32-bit wide memory which has
;     a 0 byte at every address, which is of type ROM from address #x0 to
;     address #x7FFFFFF, and which is of type RAM at all other addresses,
;  6. mem is the result of first loading gcd-code into an empty memory at
;     pc and then further loading the two natural numbers 54 and 42 and
;     the return address of the caller (#X22B2) at the location pointed
;     to by stack-pointer,
;  7. initial-state is an mc-state whose five fields are 'running, rfile,
;     pc, ccr, and mem, respectively, and finally final-state is the result
;     of running stepn for 37 instructions starting with initial-state,
; then, if we examine final-state, we find:
;  1. the machine is still 'running,
;  2. the register file is '(6 0 0 0 0 0 0 0 0 0 0 0 0 0 #XEFFFE4C #XEFFFE44),
;     observing that d0 is equal to 6, the GCD of 54 and 42, and
;  3. the program counter is set to #X22B2, the return address to the caller.
;
; This theorem should not be confused with the much more general theorem
; stating the correctness of the same GCD program on all input, a theorem
; whose mechanical proof is described in [11].

(make-lib "mc20-1")
```

B.3 A Lemma Library For Machine Code Program Proving

This section contains the lemma library built for machine code program proving.

```
; ------------------------------------------------------------------------
; Date:      Jan, 1991
; Modified:  Sep 4, 1992.
; File:      mc20-2.events
; ------------------------------------------------------------------------
;
;------------------------------------------------------------------------
;                        PROVING PHASE
;
; we establish basic theory for program proving based upon our MC68020
; specification.
;------------------------------------------------------------------------
;
; load and compile the spec.
(note-lib "mc20-1")
```

```
(compile-uncompiled-defns "tmp")

; a segment of the memory is ROM, iff it is pc-readable(read-only).
(defn rom-addrp (addr mem n)
  (pc-read-memp addr mem n))

; a segment of the memory is RAM, iff it is writable(random).
(defn ram-addrp (addr mem n)
  (write-memp addr mem n))

; x is an element of lst wrt numberp.
(defn n-member (x lst)
  (if (nlistp lst)
      f
    (or (equal (fix x) (fix (car lst)))
        (n-member x (cdr lst)))))

; mod-eq returns t iff x and y are the "same" bit-vector.  It is often
; used as an induction hint to many lemmas.
(defn mod-eq (n x y)
  (if (zerop n)
      t
    (and (equal (bcar x) (bcar y))
         (mod-eq (sub1 n) (bcdr x) (bcdr y)))))

; mod32-eq is equivalent to mod-eq with n = 32. But we define it as follows.
(defn mod32-eq (x y)
  (equal (head x 32) (head y 32)))

; the equivalence is given by this lemma.
; (prove-lemma mod-eq-lemma (rewrite)
;      (equal (mod-eq 32 x y)
;             (mod32-eq x y)))

; the negation of a bit-vector.
(defn neg (n x)
  (sub n x 0))

; x, x+1, ..., x+(m-1) and y are disjoint.  We assume all the numbers are
; modulo 2^32.
(defn disjoint0 (x m y)
  (if (zerop m)
      t
    (and (not (mod32-eq (add 32 x (sub1 m)) y))
         (disjoint0 x (sub1 m) y))))

; x, x+1, ..., x+(m-1) and y, y+1, ..., y+(n-1) are disjoint.
(defn disjoint (x m y n)
  (if (zerop n)
      t
    (and (disjoint0 x m (add 32 y (sub1 n)))
         (disjoint x m y (sub1 n)))))

;                 BIT VECTOR AS LIST OF BIT
```

```
; A lower level layer.  This is part of my old spec that specifies the
; microprocessor at a relatively lower level.
(defn bit (bv)
  (if (nlistp bv) (b0) (if (b0p (car bv)) (b0) (b1))))

(defn vec (bv)
  (if (nlistp bv) nil (cdr bv)))

(defn fix-bv (bv)
  (if (nlistp bv)
      nil
    (cons (bit bv) (fix-bv (vec bv)))))

(defn bv-len (bv)
  (if (nlistp bv)
      0
    (add1 (bv-len (vec bv)))))

(defn bv-not (bv)
  (if (nlistp bv)
      nil
    (cons (b-not (bit bv)) (bv-not (vec bv)))))

(defn bv-head (bv n)
  (if (zerop n)
      nil
    (cons (bit bv) (bv-head (vec bv) (sub1 n)))))

(defn bv-tail (bv n)
  (if (zerop n)
      (fix-bv bv)
    (bv-tail (vec bv) (sub1 n))))

(defn bv-bitn (bv n)
  (if (zerop n)
      (bit bv)
    (bv-bitn (vec bv) (sub1 n))))

(defn bv-mbit (bv)
  (if (nlistp bv)
      (b0)
    (if (nlistp (vec bv))
        (bit bv)
      (bv-mbit (vec bv)))))

(defn bv-adder (c x y)
  (if (nlistp x)
      nil
    (cons (b-eor (b-eor (bit x) (bit y)) c)
          (bv-adder (b-or (b-and (bit x) (bit y))
                          (b-or (b-and (bit x) c) (b-and (bit y) c)))
                    (vec x)
                    (vec y)))))
```

```
; conversion functions.
(defn bv-to-nat (bv)
  (if (nlistp bv)
      0
    (plus (bit bv) (times 2 (bv-to-nat (vec bv))))))

(defn bv-sized-to-nat (bv size)
  (if (zerop size)
      0
    (plus (bit bv) (times 2 (bv-sized-to-nat (vec bv) (sub1 size))))))

(defn nat-to-bv (n)
  (if (zerop n)
      nil
    (cons (remainder n 2) (nat-to-bv (quotient n 2)))))

(defn nat-to-bv-sized (n size)
  (if (zerop size)
      nil
    (cons (remainder n 2) (nat-to-bv-sized (quotient n 2) (sub1 size)))))

;               LISTP
(prove-lemma bv-len-listp (rewrite)
     (equal (equal (bv-len x) 0) (nlistp x)))

(prove-lemma bv-head-listp (rewrite)
     (equal (listp (bv-head x n)) (not (zerop n)))
     ((expand (bv-head x 1))))

(prove-lemma bv-adder-listp (rewrite)
     (equal (listp (bv-adder c x y)) (listp x)))

;               BV-LEN
(prove-lemma bv-head-len (rewrite)
     (equal (bv-len (bv-head x n)) (fix n)))

(prove-lemma nat-to-bv-sized-len (rewrite)
     (equal (bv-len (nat-to-bv-sized x n)) (fix n)))

(prove-lemma bv-adder-len (rewrite)
     (equal (bv-len (bv-adder c x y)) (bv-len x)))

;                 CONVERSION THEOREMS
; nat-to-bv-sized only considers the first k binary digits of natural number n.
(prove-lemma nat-to-bv-sized-la0 (rewrite)
     (equal (nat-to-bv-sized (remainder n (exp 2 k)) k)
            (nat-to-bv-sized n k)))

; rewrite nat-to-bv-sized into nat-to-bv.
(prove-lemma nat-to-bv-sized-head (rewrite)
     (equal (nat-to-bv-sized m n)
            (bv-head (nat-to-bv m) n)))

; rewrite bv-sized-to-nat into bv-to-nat.
```

```
(prove-lemma bv-sized-to-nat-head (rewrite)
     (equal (bv-sized-to-nat x n)
            (bv-to-nat (bv-head x n))))

(prove-lemma bv-to-nat-to-bv (rewrite)
     (equal (bv-head (nat-to-bv (bv-to-nat x)) n)
            (bv-head x n))
     ((disable plus-add1)))

(prove-lemma nat-to-bv-to-nat (rewrite)
     (equal (bv-to-nat (nat-to-bv n)) (fix n)))

(prove-lemma bv-head-nat (rewrite)
     (equal (bv-to-nat (bv-head x n))
            (if (lessp (bv-to-nat x) (exp 2 n))
                (bv-to-nat x)
              (remainder (bv-to-nat x) (exp 2 n)))))

(prove-lemma bv-to-nat-to-bv-sized (rewrite)
     (equal (nat-to-bv-sized (bv-to-nat x) n)
            (bv-head x n)))

(prove-lemma nat-to-bv-sized-to-nat (rewrite)
     (equal (bv-to-nat (nat-to-bv-sized m n))
            (if (lessp m (exp 2 n))
                (fix m)
              (remainder m (exp 2 n)))))

(prove-lemma bv-sized-to-nat-to-bv-sized (rewrite)
     (equal (nat-to-bv-sized (bv-sized-to-nat x n) n)
            (bv-head x n)))

(prove-lemma nat-to-bv-sized-sized-to-nat (rewrite)
     (equal (bv-sized-to-nat (nat-to-bv-sized m n) n)
            (if (lessp m (exp 2 n))
                (fix m)
              (remainder m (exp 2 n)))))

(disable bv-sized-to-nat-head)
(disable nat-to-bv-sized-head)

;            BV-BITN and BV-NOT
(prove-lemma bv-bitn-not (rewrite)
     (equal (bv-bitn (bv-not x) i)
            (if (lessp i (bv-len x))
                (b-not (bv-bitn x i))
              (b0))))

(prove-lemma bv-to-nat-rangep (rewrite)
     (nat-rangep (bv-to-nat x) (bv-len x)))

;            BV-ADDER
; a bridge function.
(defn bv-adder&carry (c x y)
```

```
  (if (nlistp x)
      (cons (fix-bit c) nil)
    (cons (b-eor c (b-eor (bit x) (bit y)))
          (bv-adder&carry (b-or (b-and (bit x) (bit y))
                                (b-or (b-and (bit x) c) (b-and (bit y) c)))
                          (vec x)
                          (vec y)))))

(prove-lemma bv-adder&carry-len (rewrite)
     (equal (bv-len (bv-adder&carry c x y)) (add1 (bv-len x))))

(prove-lemma fix-bv-adder&carry (rewrite)
     (equal (fix-bv (bv-adder&carry c x y))
            (bv-adder&carry c x y)))

; the relation between bv-adder&carry and bv-adder.  We have successfully
; constructed the bridge!
(prove-lemma bv-adder-bridge (rewrite)
     (equal (bv-adder c x y)
            (bv-head (bv-adder&carry c x y) (bv-len x))))

;                  lifting lemmas
(prove-lemma bv-not-lognot (rewrite)
     (equal (bv-to-nat (bv-not x))
            (lognot (bv-len x) (bv-to-nat x)))
     ((disable remainder quotient)))

(prove-lemma bv-bitn-bitn (rewrite)
     (equal (bv-bitn x n)
            (bitn (bv-to-nat x) n))
     ((disable remainder quotient)))

(prove-lemma bv-mbit-bitn (rewrite)
     (equal (bv-mbit x)
            (if (equal (bv-len x) 0)
                0
              (bitn (bv-to-nat x) (sub1 (bv-len x)))))
     ((disable remainder quotient)))

(prove-lemma bv-adder&carry-nat (rewrite)
     (implies (equal (bv-len x) (bv-len y))
              (equal (bv-to-nat (bv-adder&carry c x y))
                     (plus (fix-bit c) (bv-to-nat x) (bv-to-nat y)))))

(prove-lemma bv-adder-nat (rewrite)
     (implies (and (equal (bv-len x) (bv-len y))
                   (bitp c))
              (equal (bv-to-nat (bv-adder c x y))
                     (adder (bv-len x) c (bv-to-nat x) (bv-to-nat y))))
     ((disable remainder)))

;                    EVENP
; add-evenp states that the sum of two even numbers is still even.
(prove-lemma add-evenp (rewrite)
```

```
     (implies (and (evenp x) (evenp y))
              (evenp (add n x y))))

(disable evenp)

;                      LOGNOT
(prove-lemma lognot-0 (rewrite)
     (implies (not (numberp x))
              (and (equal (lognot x y) (lognot 0 y))
                   (equal (lognot y x) (lognot y 0)))))

(prove-lemma lognot-nat-rangep (rewrite)
     (nat-rangep (lognot n x) n))

(prove-lemma lognot-lognot (rewrite)
     (equal (lognot n (lognot n x))
            (head x n)))

(prove-lemma lognot-cancel (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n))
              (equal (equal (lognot n x) (lognot n y))
                     (equal (fix x) (fix y)))))

; the integer interpretation of lognot.
(prove-lemma lognot-int (rewrite)
     (implies (and (nat-rangep x n)
                   (not (zerop n)))
              (equal (nat-to-int (lognot n x) n)
                     (iplus -1 (ineg (nat-to-int x n))))))

(prove-lemma adder-lognot (rewrite)
     (implies (bitp c)
              (equal (lognot n (adder n c x y))
                     (adder n (b-not c) (lognot n x) (lognot n y))))
     ((disable plus difference)))

(disable lognot)

;               BASICS ABOUT HEAD & TAIL
(prove-lemma head-0 (rewrite)
     (equal (head 0 n) 0))

(prove-lemma tail-0 (rewrite)
     (equal (tail 0 n) 0))

(prove-lemma head-of-0 (rewrite)
     (equal (head x 0) 0))

(prove-lemma tail-of-0 (rewrite)
     (equal (tail x 0) (fix x)))

(prove-lemma head-lessp (rewrite)
     (lessp (head x n) (exp 2 n)))
```

```
(prove-lemma tail-lessp (rewrite)
     (equal (lessp (tail x n) y)
            (lessp x (times (exp 2 n) y))))

(prove-lemma head-leq (rewrite)
     (not (lessp x (head x n))))

(prove-lemma tail-leq (rewrite)
     (not (lessp x (tail x n))))

(prove-lemma tail-equal-0 (rewrite)
     (equal (equal (tail x n) 0)
            (nat-rangep x n)))

(prove-lemma head-lemma (rewrite)
     (implies (nat-rangep x n)
              (equal (head x n) (fix x))))

(prove-lemma tail-lemma (rewrite)
     (implies (nat-rangep x n)
              (equal (tail x n) 0)))

(prove-lemma replace-0 (rewrite)
     (and (equal (replace 0 x y) (fix y))
          (equal (replace n x 0) (head x n))
          (implies (and (nat-rangep x n)
                        (nat-rangep y n))
                   (equal (replace n x y) (fix x)))))

(prove-lemma app-0 (rewrite)
     (and (equal (app n x 0) (head x n))
          (equal (app n 0 y) (times y (exp 2 n)))
          (equal (app 0 x y) (fix y))))

;                THEOREMS ABOUT SHIFT AND ROTATE
; the key events are LSL-UINT, LSR-UINT, ASL-INT, and ASR-INT.
; ASL-INT and ASR-INT are pretty hard.  It took me a few days to get the
; proofs.  It would be a good challenge to other provers.
(prove-lemma lsl-uint (rewrite)
     (implies (uint-rangep x (difference n cnt))
              (equal (nat-to-uint (lsl n x cnt))
                     (times (nat-to-uint x) (exp 2 cnt)))))

(prove-lemma lsr-uint (rewrite)
     (equal (nat-to-uint (lsr x cnt))
            (quotient (nat-to-uint x) (exp 2 cnt))))

; two lemmas for asl-int.
(prove-lemma remainder-diff-la ()
     (implies (and (equal (remainder x z) 0)
                   (lessp y z))
              (equal (remainder (difference x y) z)
                     (if (leq x y)
```

```
                      0
                    (if (zerop y) 0 (difference z y)))))
     ((use (remainder-plus2 (i (difference x z)) (x (difference z y)) (j z)))))

(prove-lemma asl-int-crock1 (rewrite)
     (implies (lessp x (exp 2 (sub1 (difference n s))))
              (equal (head (times x (exp 2 s)) n)
                     (times x (exp 2 s)))))

(prove-lemma asl-int-crock2 (rewrite)
     (implies (and (leq (times (exp 2 s) (difference (exp 2 n) x))
                        (exp 2 (sub1 n)))
                   (lessp x (exp 2 n)))
              (equal (head (times x (exp 2 s)) n)
                     (difference (exp 2 n)
                                 (times (exp 2 s)
                                        (difference (exp 2 n) x)))))
     ((use (remainder-diff-la (x (times (exp 2 n) (exp 2 s)))
                              (y (difference (times (exp 2 n) (exp 2 s))
                                             (times x (exp 2 s))))
                              (z (exp 2 n))))
      (disable exp-plus)))

(prove-lemma asl-int (rewrite)
     (implies (and (nat-rangep x n)
                   (int-rangep (nat-to-int x n) (difference n s)))
              (equal (nat-to-int (asl n x s) n)
                     (itimes (nat-to-int x n) (exp 2 s))))
     ((disable head)))

(disable asl-int-crock1)
(disable asl-int-crock2)

(prove-lemma quotient-diff-la ()
     (implies (and (equal (remainder x z) 0)
                   (lessp y z))
              (equal (quotient (difference x y) z)
                     (if (zerop y) (quotient x z) (sub1 (quotient x z)))))
     ((use (quotient-plus2 (i (difference x z)) (x (difference z y)) (j z)))))

(prove-lemma remainder-diff (rewrite)
     (implies (equal (remainder x z) 0)
              (equal (remainder (difference x y) z)
                     (if (or (leq x y) (equal (remainder y z) 0))
                           0
                         (difference z (remainder y z)))))
     ((use (remainder-diff-la (x (difference x (times z (quotient y z))))
                              (y (remainder y z))))))

(prove-lemma quotient-diff (rewrite)
     (implies (equal (remainder x z) 0)
              (equal (quotient (difference x y) z)
                     (if (equal (remainder y z) 0)
                         (difference (quotient x z) (quotient y z))
```

```
                    (sub1 (difference (quotient x z) (quotient y z))))))
     ((use (quotient-diff-la (x (difference x (times z (quotient y z))))
                             (y (remainder y z))))))

(prove-lemma quotient-exp-lessp (rewrite)
     (implies (leq cnt n)
              (equal (lessp (quotient x (exp 2 cnt))
                            (exp 2 (difference n cnt)))
                     (lessp x (exp 2 n)))))

(prove-lemma lessp-times-exp-1s (rewrite)
     (implies (equal (plus m n) (fix k))
              (equal (lessp (plus x (times (exp 2 m) (sub1 (exp 2 n))))
                            (exp 2 (sub1 k)))
                     (if (zerop n) (lessp x (exp 2 (sub1 (plus m n)))) f)))
     ((induct (plus x y)))))

(prove-lemma lessp-app-1s (rewrite)
     (implies (equal (plus m n) (fix k))
              (equal (lessp (app m x (sub1 (exp 2 n))) (exp 2 (sub1 k)))
                     (if (zerop n)
                         (lessp (head x m) (exp 2 (sub1 (plus m n))))
                       f)))))

(prove-lemma times-sub1 (rewrite)
     (equal (times (sub1 y) x)
            (difference (times x y) x))
     ((use (times-distributes-difference (z 1)))))

(prove-lemma difference-app-1s (rewrite)
     (implies (equal (plus m n) (fix k))
              (equal (difference (exp 2 k) (app m x (sub1 (exp 2 n))))
                     (difference (exp 2 m) (head x m)))))

(disable times-sub1)

(prove-lemma asr-int-crock (rewrite)
     (implies (and (lessp x (exp 2 n))
                   (not (lessp x (exp 2 (sub1 n))))
                   (leq cnt n))
              (equal (lessp (app (difference n cnt)
                                 (quotient x (exp 2 cnt))
                                 (sub1 (exp 2 cnt)))
                            (exp 2 (sub1 n)))
                     f))
     ((disable app))))

(prove-lemma asr-int (rewrite)
     (implies (nat-rangep x n)
              (equal (nat-to-int (asr n x i) n)
                     (if (negativep (nat-to-int x n))
                         (if (equal (iremainder (nat-to-int x n) (exp 2 i)) 0)
                             (iquotient (nat-to-int x n) (exp 2 i))
                           (iplus -1 (iquotient (nat-to-int x n) (exp 2 i))))
```

```
                     (iquotient (nat-to-int x n) (exp 2 i)))))
     ((disable app exp quotient-times-lessp)))

(disable asr-int-crock)
(disable quotient-exp-lessp)
(disable remainder-diff)
(disable quotient-diff)

(prove-lemma lsl-0 (rewrite)
     (equal (lsl n x 0) (head x n)))

(prove-lemma lsl-nat-rangep (rewrite)
     (nat-rangep (lsl n x cnt) n))

(prove-lemma lsl-lsl (rewrite)
     (equal (lsl n (lsl n x cnt1) cnt2)
            (lsl n x (plus cnt1 cnt2))))

(prove-lemma lsr-0 (rewrite)
     (equal (lsr x 0) (fix x)))

(prove-lemma lsr-nat-rangep (rewrite)
     (implies (nat-rangep x n)
              (nat-rangep (lsr x cnt) n)))

(prove-lemma lsr-lsr (rewrite)
     (equal (lsr (lsr x cnt1) cnt2)
            (lsr x (plus cnt1 cnt2))))

(prove-lemma asl-0 (rewrite)
     (equal (asl n x 0) (head x n)))

(prove-lemma asl-nat-rangep (rewrite)
     (nat-rangep (asl n x cnt) n))

(prove-lemma asl-asl (rewrite)
     (equal (asl n (asl n x cnt1) cnt2)
            (asl n x (plus cnt1 cnt2))))

(prove-lemma asr-0 (rewrite)
     (implies (nat-rangep x n)
              (equal (asr n x 0) (fix x))))

(prove-lemma asr-nat-rangep (rewrite)
     (implies (nat-rangep x n)
              (nat-rangep (asr n x cnt) n))
     ((use (lessp-plus-times-exp2 (x (quotient x (exp 2 cnt)))
                                  (y (sub1 (exp 2 cnt)))
                                  (i (difference n cnt))))))

; unproved!  Not useful yet.
; (prove-lemma asr-asr (rewrite)
;     (implies (and (nat-rangep x n)
;                   (not (zerop n)))
```

```
;              (equal (asr n (asr n x cnt1) cnt2)
;                     (asr n x (plus cnt1 cnt2)))))

; the integer interpretation of ext.
(prove-lemma mbit-means-lessp (rewrite)
     (implies (nat-rangep x n)
              (equal (bitn x (sub1 n))
                     (if (lessp x (exp 2 (sub1 n))) 0 1))))

(prove-lemma exp2-lessp (rewrite)
     (implies (lessp i j)
              (lessp (exp 2 i) (exp 2 j))))

; note that ext-lemma takes care of the other aspects about ext.
(prove-lemma ext-int (rewrite)
     (implies (and (nat-rangep x n)
                   (lessp n size))
              (equal (nat-to-int (ext n x size) size)
                     (nat-to-int x n)))
     ((disable app exp plus)))

(disable exp2-lessp)

;                       INTEGERS!
; functions with a postfix 'end' will eventually be disabled.  We use them
; to ''tell'' the prover that something unexpected happens.  They can
; somehow force the prover to stop proving and print out some information.
(defn adder-int-end (n c x y)
  (nat-to-int (remainder (plus (fix-bit c)
                               (int-to-nat x n)
                               (int-to-nat y n))
                         (exp 2 n))
              n))

(prove-lemma plus-to-iplus (rewrite)
     (implies (and (integerp x)
                   (integerp y)
                   (int-rangep x n)
                   (int-rangep y n))
              (equal (adder-int-end n c x y)
                     (if (int-rangep (iplus x (iplus y (fix-bit c))) n)
                         (iplus x (iplus y (fix-bit c)))
                       (if (negativep x)
                           (iplus x (iplus y (iplus (fix-bit c) (exp 2 n))))
                         (iplus x (iplus y (iplus (fix-bit c)
                                                  (minus (exp 2 n)))))))))
     ((disable times-distributes-plus-new correctness-of-cancel-lessp-plus)
      (enable times-distributes-plus)))

(disable plus-to-iplus)

(prove-lemma iplus-with-carry-negativep (rewrite)
     (implies (and (integerp x)
                   (integerp y)
```

```
                (bitp z)
                (negativep x)
                (negativep y))
           (negativep (iplus x (iplus y z)))))

(prove-lemma iplus-with-carry-non-negativep (rewrite)
     (implies (and (integerp x)
                   (integerp y)
                   (bitp z)
                   (not (negativep x))
                   (not (negativep y)))
              (not (negativep (iplus x (iplus y z))))))

; two lemmas for addx-v.  They will be disabled right after we use them.
(prove-lemma addx-v-crock1 (rewrite)
     (implies (and (integerp x)
                   (integerp y)
                   (bitp z)
                   (int-rangep x n)
                   (int-rangep y n))
              (not (negativep (iplus x (iplus y (iplus z (exp 2 n))))))))

(prove-lemma addx-v-crock2 (rewrite)
     (implies (and (integerp x)
                   (integerp y)
                   (bitp z)
                   (not (zerop n))
                   (int-rangep x n)
                   (int-rangep y n))
              (negativep (iplus x (iplus y (iplus z (minus (exp 2 n))))))))

; two lemmas for BMI of add.  They will be disabled right after we
; use them.
(prove-lemma add-bmi-crock1 (rewrite)
     (implies (and (integerp x)
                   (integerp y)
                   (int-rangep x n)
                   (int-rangep y n))
              (not (negativep (iplus x (iplus y (exp 2 n)))))))

(prove-lemma add-bmi-crock2 (rewrite)
     (implies (and (integerp x)
                   (integerp y)
                   (int-rangep x n)
                   (int-rangep y n))
              (negativep (iplus x (iplus y (minus (exp 2 n)))))))

; two lemmas for BMI of sub.  They will be disabled right after we
; use them.
(prove-lemma ilessp-crock1 (rewrite)
     (implies (and (integerp x)
                   (integerp y)
                   (int-rangep x n)
                   (int-rangep y n))
```

```
              (not (ilessp (iplus y (exp 2 n)) x))))

(prove-lemma ilessp-crock2 (rewrite)
     (implies (and (integerp x)
                   (integerp y)
                   (int-rangep x n)
                   (int-rangep y n))
              (ilessp (iplus y (minus (exp 2 n))) x)))

; the addition of two opposite sign integers will never overflow, provided
; the two integers are in "good" range.
(prove-lemma iplus-int-rangep1 (rewrite)
     (implies (and (integerp x)
                   (integerp y)
                   (bitp z)
                   (int-rangep x n)
                   (int-rangep y n)
                   (not (negativep x))
                   (negativep y))
              (int-rangep (iplus x (iplus y z)) n)))

(prove-lemma iplus-int-rangep2 (rewrite)
     (implies (and (integerp x)
                   (integerp y)
                   (bitp z)
                   (int-rangep x n)
                   (int-rangep y n)
                   (negativep x)
                   (not (negativep y)))
              (int-rangep (iplus x (iplus y z)) n)))

(prove-lemma idifference-int-rangep1 (rewrite)
     (implies (and (int-rangep x n)
                   (int-rangep y n)
                   (not (ilessp x y))
                   (not (negativep y)))
              (int-rangep (idifference y x) n)))

(prove-lemma idifference-int-rangep2 (rewrite)
     (implies (and (integerp x)
                   (integerp y)
                   (int-rangep x n)
                   (int-rangep y n)
                   (not (ilessp y x))
                   (negativep y))
              (int-rangep (idifference y x) n)))

; two lemmas about uint-rangep and int-rangep. We introduce them before any
; lemmas about uint-rangep and int-rangep.
(prove-lemma uint-rangep-la (rewrite)
     (implies (lessp x (exp 2 n))
              (uint-rangep x n)))

(prove-lemma int-rangep-la (rewrite)
```

```
     (implies (and (numberp x)
                   (lessp x (exp 2 (sub1 n))))
              (int-rangep x n)))

;                NAT-TO-UINT & UINT-TO-NAT
(prove-lemma nat-to-uint-to-nat (rewrite)
     (equal (uint-to-nat (nat-to-uint x))
            (fix x)))

(prove-lemma uint-to-nat-to-uint (rewrite)
     (equal (nat-to-uint (uint-to-nat x))
            (fix x)))

(prove-lemma uint-to-nat-rangep (rewrite)
     (implies (lessp x (exp 2 n))
              (nat-rangep (uint-to-nat x) n))
     ((enable nat-rangep uint-to-nat)))

(prove-lemma nat-to-uint-rangep (rewrite)
     (implies (nat-rangep x n)
              (uint-rangep (nat-to-uint x) n)))

;                NAT-TO-INT & INT-TO-NAT
(prove-lemma nat-to-int-0 (rewrite)
     (equal (nat-to-int 0 n) 0))

(prove-lemma int-to-nat-0 (rewrite)
     (equal (int-to-nat 0 n) 0))

; nat-to-int always returns an integer, provided the input is a "good"
; natural number.
(prove-lemma nat-to-int-integerp (rewrite)
     (equal (integerp (nat-to-int x n))
            (nat-rangep x n)))

; nat-to-int always returns a "good" integer.
(prove-lemma nat-to-int-rangep (rewrite)
     (int-rangep (nat-to-int x n) n))

; int-to-nat always returns a "good" natural number, provided the input
; is a "good" integer.
(prove-lemma int-to-nat-rangep (rewrite)
     (implies (and (integerp x)
                   (int-rangep x n))
              (nat-rangep (int-to-nat x n) n)))

(prove-lemma nat-to-int-to-nat (rewrite)
     (implies (nat-rangep x n)
              (equal (int-to-nat (nat-to-int x n) n)
                     (fix x))))

(prove-lemma int-to-nat-to-int (rewrite)
     (implies (and (integerp x)
                   (int-rangep x n))
```

```
              (equal (nat-to-int (int-to-nat x n) n)
                     (fix-int x))))

(prove-lemma int-to-nat=0 (rewrite)
     (implies (and (int-rangep x n)
                   (integerp x))
              (equal (equal (int-to-nat x n) 0)
                     (equal x 0))))

; /x/ = /y/ --> x = y.
(prove-lemma nat-to-int= (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n))
              (equal (equal (nat-to-int x n) (nat-to-int y n))
                     (equal (fix x) (fix y)))))

; nat-to-int and int-to-nat are disabled.
(disable nat-to-int)
(disable int-to-nat)

; theorems about int-rangep.
(prove-lemma abs-lessp-int-rangep (rewrite)
     (implies (and (int-rangep x n)
                   (lessp (abs y) (abs x)))
              (int-rangep y n)))

; 0 is always in "good" integer range.
(prove-lemma int-rangep-of-0 (rewrite)
     (int-rangep 0 n))

(prove-lemma sub1-int-rangep (rewrite)
     (implies (int-rangep x n)
              (int-rangep (sub1 x) n)))

(prove-lemma difference-int-rangep (rewrite)
     (implies (int-rangep x n)
              (int-rangep (difference x y) n)))

(prove-lemma quotient-int-rangep (rewrite)
     (implies (int-rangep x n)
              (int-rangep (quotient x y) n)))

(prove-lemma remainder-int-rangep (rewrite)
     (implies (and (int-rangep y n)
                   (not (zerop y)))
              (int-rangep (remainder x y) n)))

; iquotient is almost always in the integer range.
(prove-lemma iquotient-int-rangep (rewrite)
     (implies (int-rangep x n)
              (equal (int-rangep (iquotient x y) n)
                     (if (equal y -1)
                         (not (equal x (minus (exp 2 (sub1 n)))))
                       t))))
```

```
; iremainder is almost always in the integer range.
(prove-lemma iremainder-int-rangep (rewrite)
     (implies (and (int-rangep y n)
                   (integerp y)
                   (not (equal y 0)))
              (int-rangep (iremainder x y) n)))

; lemmas about integerp.
(prove-lemma integerp-fix-int (rewrite)
     (implies (integerp x)
              (equal (fix-int x) x)))

(prove-lemma fix-int-integerp (rewrite)
     (integerp (fix-int x)))

(prove-lemma minus-integerp (rewrite)
     (equal (integerp (minus x))
            (not (zerop x))))

(prove-lemma integerp-minus0 (rewrite)
     (implies (integerp x)
              (not (equal x (minus 0)))))

(prove-lemma negativep-guts0 (rewrite)
     (implies (and (integerp x)
                   (negativep x))
              (not (equal (negative-guts x) 0))))

(prove-lemma numberp-integerp (rewrite)
     (implies (numberp x)
              (integerp x)))

(prove-lemma iplus-integerp (rewrite)
     (integerp (iplus x y)))

(prove-lemma ineg-integerp (rewrite)
     (integerp (ineg x)))

(prove-lemma idifference-integerp (rewrite)
     (integerp (idifference x y)))

(prove-lemma itimes-integerp (rewrite)
     (integerp (itimes x y)))

(prove-lemma iremainder-integerp (rewrite)
     (integerp (iremainder x y)))

(prove-lemma iquotient-integerp (rewrite)
     (integerp (iquotient x y)))

; theorems about iplus, idifference, and ineg.
(prove-lemma iplus-commutativity (rewrite)
     (equal (iplus x y) (iplus y x)))
```

```
(prove-lemma iplus-commutativity1 (rewrite)
     (equal (iplus x (iplus y z))
            (iplus y (iplus x z))))

(prove-lemma iplus-associativity (rewrite)
     (equal (iplus (iplus x y) z)
            (iplus x (iplus y z)))
     ((disable iplus)))

(prove-lemma ineg-iplus (rewrite)
     (equal (ineg (iplus x y))
            (iplus (ineg x) (ineg y))))

(prove-lemma iplus-0 (rewrite)
     (implies (izerop x)
              (and (equal (iplus x y) (fix-int y))
                   (equal (iplus y x) (fix-int y)))))

(prove-lemma iplus-1--1 (rewrite)
     (equal (iplus 1 (iplus -1 x))
            (fix-int x)))

(prove-lemma idifference-x-x (rewrite)
     (equal (idifference x x) 0))

(prove-lemma idifference-negativep (rewrite)
     (equal (negativep (idifference x y))
            (ilessp x y)))

; lemmas about ilessp.
(prove-lemma ilessp-reflex (rewrite)
     (not (ilessp x x)))

(prove-lemma ilessp-entails-ileq (rewrite)
     (implies (ilessp x y)
              (not (ilessp y x))))

; lemmas about itimes.
(prove-lemma itimes-0 (rewrite)
     (equal (itimes 0 x) 0))

(prove-lemma itimes-equal-0 (rewrite)
     (equal (equal (itimes x y) 0)
            (or (izerop x) (izerop y))))

(prove-lemma itimes-commutativity (rewrite)
     (equal (itimes x y)
            (itimes y x)))

(prove-lemma itimes-associativity (rewrite)
     (equal (itimes (itimes x y) z)
            (itimes x (itimes y z))))
```

```
(prove-lemma itimes-equal-cancellation (rewrite)
     (implies (not (izerop x))
              (equal (equal (itimes x y) (itimes x z))
                     (equal (fix-int y) (fix-int z)))))

(prove-lemma itimes-sign (rewrite)
     (implies (and (integerp x)
                   (integerp y))
              (equal (negativep (itimes x y))
                     (or (and (numberp x)
                              (not (equal x 0))
                              (negativep y))
                         (and (negativep x)
                              (numberp y)
                              (not (equal y 0)))))))

; lemmas about iremainder and iquotient.
(prove-lemma iremainder-wrt-1 (rewrite)
     (equal (iremainder x 1) 0))

(prove-lemma iquotient-wrt-1 (rewrite)
     (equal (iquotient x 1) (fix-int x)))

(prove-lemma iremainder-wrt--1 (rewrite)
     (equal (iremainder x -1) 0))

(prove-lemma iquotient-wrt--1 (rewrite)
     (equal (iquotient x -1) (ineg x)))

(disable iplus)
(disable itimes)
(disable iremainder)
(disable iquotient)
(disable integerp)

; unsigned interpretation of mulu.
(prove-lemma times-lessp_1 (rewrite)
     (implies (and (lessp x x1)
                   (lessp y y1))
              (lessp (times x y) (times x1 y1))))

; three instances of mulu-nat.
(prove-lemma mulu_1632-nat (rewrite)
     (implies (and (nat-rangep x 16)
                   (nat-rangep y 16))
              (equal (nat-to-uint (mulu 32 x y 16))
                     (times (nat-to-uint x) (nat-to-uint y))))
     ((use (times-lessp_1 (x1 (exp 2 16)) (y1 (exp 2 16))))))

(prove-lemma mulu_3264-nat (rewrite)
     (implies (and (nat-rangep x 32)
                   (nat-rangep y 32))
              (equal (nat-to-uint (mulu 64 x y 32))
                     (times (nat-to-uint x) (nat-to-uint y)))))
```

```
     ((use (times-lessp_1 (x1 (exp 2 32)) (y1 (exp 2 32))))))

(prove-lemma mulu_3232-nat (rewrite)
     (implies (uint-rangep (times (nat-to-uint x) (nat-to-uint y)) 32)
              (equal (nat-to-uint (mulu 32 x y 32))
                     (times (nat-to-uint x) (nat-to-uint y)))))

; signed interpretation of muls.
(prove-lemma exp2-lessp-crock (rewrite)
     (implies (and (not (zerop i))
                   (if (lessp k i) f t))
              (lessp (exp 2 (sub1 i)) (exp 2 k))))

(prove-lemma head-int-crock (rewrite)
     (implies (and (integerp x)
                   (int-rangep x n)
                   (leq n j))
              (equal (head (int-to-nat x j) n)
                     (int-to-nat x n)))
     ((enable remainder-diff integerp int-to-nat)))

(disable exp2-lessp-crock)

(prove-lemma times-lessp_2 (rewrite)
     (implies (and (leq x x1)
                   (leq y y1)
                   (not (zerop x1))
                   (not (zerop y1)))
              (lessp (times x y) (times 2 x1 y1))))

(prove-lemma times-lessp_3 (rewrite)
     (implies (and (leq x (exp 2 (sub1 n)))
                   (leq y (exp 2 (sub1 n)))
                   (not (zerop n)))
              (equal (lessp (times x y) (exp 2 (sub1 (plus n n))))
                     t))
     ((use (times-lessp_2 (x1 (exp 2 (sub1 n))) (y1 (exp 2 (sub1 n)))))))

(prove-lemma times-lessp_4 (rewrite)
     (implies (and (leq x (exp 2 (sub1 n)))
                   (leq y (exp 2 (sub1 n)))
                   (not (zerop n)))
              (equal (lessp (exp 2 (sub1 (plus n n))) (times x y))
                     f))
     ((use (times-lessp_2 (x1 (exp 2 (sub1 n))) (y1 (exp 2 (sub1 n)))))))

(prove-lemma muls-crock (rewrite)
     (implies (and (int-rangep x n)
                   (int-rangep y n)
                   (not (zerop n)))
              (int-rangep (itimes x y) (plus n n)))
     ((enable itimes integerp)))

; three instances of muls-int.
```

```
(prove-lemma muls_1632-int (rewrite)
     (implies (and (nat-rangep x 16)
                   (nat-rangep y 16))
              (equal (nat-to-int (muls 32 x y 16) 32)
                     (itimes (nat-to-int x 16) (nat-to-int y 16))))
     ((use (muls-crock (x (nat-to-int x 16)) (y (nat-to-int y 16)) (n 16)))))

(prove-lemma muls_3264-int (rewrite)
     (implies (and (nat-rangep x 32)
                   (nat-rangep y 32))
              (equal (nat-to-int (muls 64 x y 32) 64)
                     (itimes (nat-to-int x 32) (nat-to-int y 32))))
     ((use (muls-crock (x (nat-to-int x 32)) (y (nat-to-int y 32)) (n 32)))))

(prove-lemma muls_3232-int (rewrite)
     (implies (int-rangep (itimes (nat-to-int x 32) (nat-to-int y 32)) 32)
              (equal (nat-to-int (muls 32 x y 32) 32)
                     (itimes (nat-to-int x 32) (nat-to-int y 32)))))

(disable times-lessp_1)
(disable times-lessp_2)
(disable times-lessp_3)
(disable times-lessp_4)
(disable muls-crock)

; signed interpretation of irem.
; notice that we only consider the case that the divisor and the result
; have the same boundary n.  Since it is enough to cover the situations
; in the DIVS instructions.
(prove-lemma irem-int (rewrite)
     (implies (and (nat-rangep x n)
                   (not (equal (nat-to-int x n) 0)))
              (equal (nat-to-int (irem n x n y j) n)
                     (iremainder (nat-to-int y j) (nat-to-int x n)))))

; signed interpretation of iquot.
; notice that we only consider the case that the length of the result
; is at most the length of the dividend.  Since it is enough to cover
; the situations in DIVS instruction.
(prove-lemma iquot-int (rewrite)
     (implies (and (int-rangep (iquotient (nat-to-int y j) (nat-to-int x i))
                               n)
                   (leq n j))
              (equal (nat-to-int (iquot n x i y j) n)
                     (iquotient (nat-to-int y j) (nat-to-int x i)))))

(disable head-int-crock)

; unsigned interpratation of rem.
(prove-lemma rem-nat (rewrite)
     (implies (nat-rangep x n)
              (equal (nat-to-uint (rem n x y))
                     (if (equal (nat-to-uint x) 0)
                         (remainder (nat-to-uint y) (exp 2 n))
```

```
                    (remainder (nat-to-uint y) (nat-to-uint x))))))

; unsigned interpretation of quot.
(prove-lemma quot-nat (rewrite)
     (implies (nat-rangep y n)
              (equal (nat-to-uint (quot n x y))
                     (quotient (nat-to-uint y) (nat-to-uint x)))))

(disable int-rangep)

; the only correct execution of DIV is when NO overflow occurs.  We
; need to pass the guard in the specification.  The following few
; lemmas are introduced for this purpose.
(prove-lemma divu-overflow (rewrite)
     (equal (divu-v n x y)
            (if (uint-rangep (quotient (nat-to-uint y) (nat-to-uint x)) n)
                0 1)))

(prove-lemma divs-overflow (rewrite)
     (equal (divs-v n x i y j)
            (if (int-rangep (iquotient (nat-to-int y j)
                                       (nat-to-int x i))
                            n)
                0 1)))

(prove-lemma divs_3232-overflow (rewrite)
     (equal (divs-v 32 x 32 y 32)
            (if (and (equal (nat-to-int y 32) -2147483648)
                     (equal (nat-to-int x 32) -1))
                1 0)))

(disable divu-v)
(disable divs-v)

;          UNSIGNED INTERPRETATION OF ADDITION AND SUBTRACTION
;
(prove-lemma add-nat-la (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n))
              (equal (add n x y)
                     (if (lessp (plus x y) (exp 2 n))
                         (plus x y)
                       (difference (plus x y) (exp 2 n))))))

(prove-lemma sub-nat-la (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n) )
              (equal (sub n x y)
                     (if (lessp y x)
                         (difference (plus y (exp 2 n)) x)
                       (difference y x)))))

(prove-lemma adder-nat-la (rewrite)
     (implies (and (nat-rangep x n)
```

```
                (nat-rangep y n)
                (bitp c))
           (equal (adder n c x y)
                  (if (lessp (plus c x y) (exp 2 n))
                      (plus c x y)
                    (difference (plus c x y) (exp 2 n))))))

(prove-lemma subtracter-nat-la (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (bitp c))
              (equal (subtracter n c x y)
                     (if (lessp y (plus c x))
                         (difference (plus y (exp 2 n)) (plus c x))
                       (difference y (plus c x)))))
     ((enable lognot)))

(prove-lemma plus-numberp (rewrite)
     (implies (not (numberp y))
              (equal (plus x y) (fix x))))

; unsigned interpretation of adder.
(prove-lemma adder-uint (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (bitp c))
              (equal (nat-to-uint (adder n c x y))
                     (if (lessp (plus c (nat-to-uint x) (nat-to-uint y))
                                (exp 2 n))
                         (plus c (nat-to-uint x) (nat-to-uint y))
                       (difference (plus c (nat-to-uint x) (nat-to-uint y))
                                   (exp 2 n)))))
     ((disable adder)))

; unsigned interpretation of subtracter.
(prove-lemma subtracter-uint (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (bitp c))
              (equal (nat-to-uint (subtracter n c x y))
                     (if (lessp (nat-to-uint y) (plus c (nat-to-uint x)))
                         (difference (plus (nat-to-uint y) (exp 2 n))
                                     (plus c (nat-to-uint x)))
                       (difference (nat-to-uint y) (plus c (nat-to-uint x))))))
     ((disable subtracter)))

; unsigned interpretation of add.
(prove-lemma add-uint (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n))
              (equal (nat-to-uint (add n x y))
                     (if (lessp (plus (nat-to-uint x) (nat-to-uint y))
                                (exp 2 n))
                         (plus (nat-to-uint x) (nat-to-uint y))
```

```
                    (difference (plus (nat-to-uint x) (nat-to-uint y))
                                (exp 2 n))))))

; unsigned interpretation of sub.
(prove-lemma sub-uint (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n))
              (equal (nat-to-uint (add n x (neg n y)))
                     (if (lessp (nat-to-uint x) (nat-to-uint y))
                         (difference (plus (nat-to-uint x) (exp 2 n))
                                     (nat-to-uint y))
                       (difference (nat-to-uint x) (nat-to-uint y)))))
     ((disable add)))

(disable plus-numberp)
(disable add-nat-la)
(disable sub-nat-la)
(disable adder-nat-la)
(disable subtracter-nat-la)

;                  THEOREMS OF ADDER
;
(prove-lemma adder-shift-carry (rewrite)
     (equal (adder n c x (adder n d y z))
            (adder n d x (adder n c y z)))
     ((disable remainder plus remainder-exit)))

(prove-lemma adder-commutativity (rewrite)
     (equal (adder n c x y) (adder n c y x)))

(prove-lemma adder-associativity (rewrite)
     (equal (adder n d (adder n c x y) z)
            (adder n c x (adder n d y z)))
     ((disable remainder plus remainder-exit)))

; another commutativity theorem about adder.
(prove-lemma adder-commutativity1 (rewrite)
     (equal (adder n c x (adder n d y z))
            (adder n c y (adder n d x z)))
     ((disable remainder plus remainder-exit)))

;               THEOREMS ABOUT ADD AND SUB
;
; trivial!
(prove-lemma add-of-0 (rewrite)
     (equal (add 0 x y) 0))

; x + 0 = x!
(prove-lemma add-0 (rewrite)
     (and (equal (add n x 0) (head x n))
          (equal (add n 0 x) (head x n))))

; x - 0 = x!
(prove-lemma sub-0 (rewrite)
```

```
     (equal (sub n 0 x) (head x n)))

; x - x = 0!
(prove-lemma sub-x-x (rewrite)
     (equal (sub n x x) 0))

; commutativity of addition : x + y = y + x.
(prove-lemma add-commutativity (rewrite)
    (equal (add n x y) (add n y x)))

; associativity of addition: (x + y) + z = x + (y + z).
(prove-lemma add-associativity (rewrite)
     (equal (add n (add n x y) z)
            (add n x (add n y z)))
     ((disable remainder plus remainder-exit)))

; another commutativity of addition: x + (y + z) = y + (x + z).
(prove-lemma add-commutativity1 (rewrite)
     (equal (add n x (add n y z))
            (add n y (add n x z)))
     ((disable remainder plus remainder-exit)))

(prove-lemma remainder-leq (rewrite)
        (not (lessp x (remainder x y))))

(prove-lemma add-leq (rewrite)
     (not (lessp (plus x y) (add n x y)))
     ((enable add)))

(prove-lemma sub-leq-la ()
     (implies (leq x y)
              (not (lessp (difference y x) (sub n x y))))
     ((use (remainder-plus2 (i (exp 2 n))
                            (j (exp 2 n))
                            (x (difference y x))))
      (disable remainder-exit)))

(disable remainder-leq)

; addition and subtraction are always in "good" range.
(prove-lemma adder-nat-rangep (rewrite)
     (nat-rangep (adder n c x y) n))

(prove-lemma subtracter-nat-rangep (rewrite)
     (nat-rangep (subtracter n c y x) n)
     ((disable adder nat-rangep)))

(prove-lemma add-nat-rangep (rewrite)
     (nat-rangep (add n x y) n))

(prove-lemma sub-nat-rangep (rewrite)
     (nat-rangep (sub n x y) n))

(prove-lemma adder-head (rewrite)
```

```
     (and (equal (adder n c (head x n) y)
                 (adder n c x y))
          (equal (adder n c x (head y n))
                 (adder n c x y))))

(prove-lemma add-head (rewrite)
     (and (equal (add n (head x n) y) (add n x y))
          (equal (add n x (head y n)) (add n x y))))

; x + y = x - z <==> y + z = 0.
(prove-lemma add-sub-cancel (rewrite)
     (implies (and (nat-rangep y n)
                   (nat-rangep z n))
              (equal (equal (add n x y) (sub n z x))
                     (equal (add n y z) 0)))
     ((enable lognot)))

; x - y = 0 <==> x = y.
(prove-lemma sub-equal-0 (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n))
              (equal (equal (sub n x y) 0)
                     (equal (fix x) (fix y))))
     ((enable lognot)))

; a bridge from add to adder.
(prove-lemma add-adder (rewrite)
     (equal (add n x y) (adder n 0 x y)))

(prove-lemma plus-add1-sub1 (rewrite)
     (implies (not (zerop y))
              (equal (add1 (plus x (sub1 y)))
                     (plus x y))))

; a bridge from sub to adder.
(prove-lemma sub-adder (rewrite)
     (equal (sub n y x)
            (adder n 1 x (lognot n y)))
     ((enable lognot)))

(prove-lemma add-neg-adder (rewrite)
     (equal (add n x (neg n y))
            (adder n 1 x (lognot n y)))
     ((enable lognot)))

; plus-add1-sub1 is a very ``dangerous'' event.  To use it,  you need
; to enable it first.
(disable plus-add1-sub1)

(disable add)
(disable sub)

; (x + y) -z = x + (y - z).
(prove-lemma add-sub (rewrite)
```

```
     (equal (sub n z (add n x y))
            (add n x (sub n z y)))
     ((disable adder)))

; (x - y) + z = x + (z - y).
(prove-lemma sub-add (rewrite)
     (equal (add n (sub n y x) z)
            (add n x (sub n y z)))
     ((disable adder)))

; (x - y) - z = x - (y + z).
(prove-lemma sub-sub (rewrite)
     (equal (sub n z (sub n y x))
            (sub n (add n y z) x))
     ((disable adder)))

; x - (y - z) = x + (z - y).
(prove-lemma sub-sub1 (rewrite)
     (equal (sub n (sub n z y) x)
            (add n x (sub n y z)))
     ((disable adder head)))

; another commutativity of addition: x + (y - z) = y + (x - z).
(prove-lemma add-commutativity2 (rewrite)
     (equal (add n x (sub n z y))
            (add n y (sub n z x)))
     ((disable adder)))

; x + y = x <==> y = 0.
(prove-lemma add-cancel0 (rewrite)
     (equal (equal (add n x y) x)
            (and (numberp x)
                 (nat-rangep x n)
                 (equal (head y n) 0))))

; x - y = x <==> y = 0.
(prove-lemma sub-cancel0 (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n))
              (equal (equal (sub n y x) x)
                     (and (numberp x) (zerop y))))
     ((enable lognot)))

(prove-lemma adder-cancel (rewrite)
     (equal (equal (adder n c x y) (adder n c x z))
            (equal (head y n) (head z n))))

; x + y = x + z <==> y = z.
(prove-lemma add-cancel (rewrite)
     (equal (equal (add n x y) (add n x z))
            (equal (head y n) (head z n))))

; x - y = x - z <==> y = z.
(prove-lemma sub-cancel (rewrite)
```

```
     (implies (and (nat-rangep y n)
                   (nat-rangep z n))
              (equal (equal (sub n y x) (sub n z x))
                     (equal (fix y) (fix z))))
     ((disable adder)))

(disable add-adder)
(disable sub-adder)

; x + (y - y) = x!
(prove-lemma addy-y (rewrite)
     (implies (nat-rangep x n)
              (equal (add n x (sub n y y))
                     (fix x))))

; y + (x - y) = x!
(prove-lemma addy-y1 (rewrite)
     (implies (nat-rangep x n)
              (equal (add n y (sub n y x))
                     (fix x))))

;           INTEGER INTERPRETATION OF ADDITION AND SUBTRACTION

; a bridge for adder-int.
(prove-lemma adder-int-bridge (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (bitp c))
              (equal (nat-to-int (adder n c x y) n)
                     (adder-int-end n c (nat-to-int x n) (nat-to-int y n))))
     ((disable remainder remainder-add1)))

(disable adder-int-end)
(disable adder)

; integer interpretation of adder.
(prove-lemma adder-int (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (bitp c))
              (equal (nat-to-int (adder n c x y) n)
                     (if (int-rangep (iplus (nat-to-int x n)
                                            (iplus (nat-to-int y n) c))
                                     n)
                         (iplus (nat-to-int x n) (iplus (nat-to-int y n) c))
                       (if (negativep (nat-to-int x n))
                           (iplus (nat-to-int x n)
                                  (iplus (nat-to-int y n)
                                         (iplus c (exp 2 n))))
                         (iplus (nat-to-int x n)
                                (iplus (nat-to-int y n)
                                       (iplus c (minus (exp 2 n)))))))))
     ((enable plus-to-iplus)))
```

```
; integer interpretation of subtracter.
(prove-lemma subtracter-int (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (bitp c)
                   (not (zerop n)))
              (equal (nat-to-int (subtracter n c x y) n)
                     (if (int-rangep (idifference (nat-to-int y n)
                                                  (iplus (nat-to-int x n) c))
                                     n)
                         (idifference (nat-to-int y n)
                                      (iplus (nat-to-int x n) c))
                       (if (negativep (nat-to-int y n))
                           (idifference (iplus (nat-to-int y n) (exp 2 n))
                                        (iplus (nat-to-int x n) c))
                         (idifference (iplus (nat-to-int y n)
                                             (minus (exp 2 n)))
                                      (iplus (nat-to-int x n) c)))))))

(disable subtracter)

; integer interpretation of add.
(prove-lemma add-int (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n))
              (equal (nat-to-int (add n x y) n)
                     (if (int-rangep (iplus (nat-to-int x n) (nat-to-int y n))
                                     n)
                         (iplus (nat-to-int x n) (nat-to-int y n))
                       (if (negativep (nat-to-int x n))
                           (iplus (nat-to-int x n)
                                  (iplus (nat-to-int y n) (exp 2 n)))
                         (iplus (nat-to-int x n)
                                (iplus (nat-to-int y n)
                                       (minus (exp 2 n)))))))))
     ((enable add-adder)))

; integer interpretation of sub.
(prove-lemma sub-int (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (not (zerop n)))
              (equal (nat-to-int (add n x (neg n y)) n)
                     (if (int-rangep (idifference (nat-to-int x n)
                                                  (nat-to-int y n))
                                     n)
                         (idifference (nat-to-int x n) (nat-to-int y n))
                       (if (negativep (nat-to-int x n))
                           (idifference (iplus (nat-to-int x n) (exp 2 n))
                                        (nat-to-int y n))
                         (idifference (iplus (nat-to-int x n)
                                             (minus (exp 2 n)))
                                      (nat-to-int y n)))))))
     ((disable neg)))
```

```
(disable add-neg-adder)

;            THEOREMS ABOUT HEAD, TAIL, APP, AND REPLACE
;
(prove-lemma head-head (rewrite)
     (equal (head (head x m) n)
            (if (lessp n m) (head x n) (head x m))))

(prove-lemma tail-tail (rewrite)
     (equal (tail (tail x m) n)
            (tail x (plus m n))))

(prove-lemma head-plus-cancel0 (rewrite)
     (equal (equal (head (plus i j) n) (head i n))
            (equal (head j n) 0))
     ((use (remainder-plus-cancel (k 0) (n (exp 2 n))))
      (disable remainder-plus-cancel)))

(prove-lemma head-plus-cancel (rewrite)
     (equal (equal (head (plus x y) n) (head (plus x z) n))
            (equal (head y n) (head z n))))

(prove-lemma head-plus-head (rewrite)
     (equal (head (plus x (head y n)) n)
            (head (plus x y) n)))

(prove-lemma remainder-plus-times-exp2-1 (rewrite)
     (implies (leq j i)
              (equal (remainder (plus x (times y (exp 2 i)))
                                (exp 2 j))
                     (remainder x (exp 2 j))))
     ((use (remainder-plus1 (i (times y (exp 2 i))) (j (exp 2 j))))))

(prove-lemma remainder2-plus-times-exp2 (rewrite)
     (implies (not (zerop i))
              (equal (remainder (plus x (times y (exp 2 i))) 2)
                     (remainder x 2)))
     ((use (remainder-plus-times-exp2-1 (j 1)))))

(prove-lemma remainder-plus-times-exp2-2 (rewrite)
     (implies (and (lessp x (exp 2 i))
                   (leq i j))
              (equal (remainder (plus x (times y (exp 2 i)))
                                (exp 2 j))
                     (if (lessp i j)
                         (plus x
                               (times (remainder y (exp 2 (difference j i)))
                                      (exp 2 i)))
                       (fix x))))
     ((enable times-distributes-plus exp2-lessp)
      (disable plus-commutativity)))

(prove-lemma quotient-plus-times-exp2-1 (rewrite)
```

```
     (implies (lessp j i)
              (equal (quotient (plus x (times y (exp 2 i)))
                               (exp 2 j))
                     (plus (quotient x (exp 2 j))
                           (times y (exp 2 (difference i j))))))
     ((use (quotient-plus1 (i (times y (exp 2 i))) (j (exp 2 j))))))

(prove-lemma exp2-leq (rewrite)
     (implies (leq i j)
              (not (lessp (exp 2 j) (exp 2 i)))))

(prove-lemma quotient-plus-times-exp2-2 (rewrite)
     (implies (and (leq i j)
                   (lessp x (exp 2 i)))
              (equal (quotient (plus x (times y (exp 2 i))) (exp 2 j))
                     (quotient y (exp 2 (difference j i)))))
     ((enable times-distributes-plus)))

(prove-lemma remainder-quotient-exp2 (rewrite)
     (equal (quotient (remainder x (exp 2 i)) (exp 2 j))
            (if (lessp j i)
                (remainder (quotient x (exp 2 j))
                           (exp 2 (difference i j)))
              0)))

(disable exp2-leq)

(prove-lemma head-replace (rewrite)
     (equal (head (replace m x y) n)
            (if (lessp m n)
                (replace m x (head y n))
              (head x n)))
     ((disable times-commutativity remainder quotient)))

(prove-lemma tail-head (rewrite)
     (equal (tail (head x m) n)
            (head (tail x n) (difference m n)))
     ((disable remainder quotient)))

(prove-lemma tail-app (rewrite)
     (equal (tail (app i x y) j)
            (if (lessp j i)
                (app (difference i j) (tail x j) y)
              (tail y (difference j i))))
     ((disable remainder quotient)))

(prove-lemma tail-replace (rewrite)
     (equal (tail (replace i x y) j)
            (if (lessp j i)
                (replace (difference i j) (tail x j) (tail y j))
              (tail y j)))
     ((disable times-commutativity remainder quotient)))

(prove-lemma replace-reflex (rewrite)
```

```
     (equal (replace n x x) (fix x)))

(prove-lemma replace-head (rewrite)
     (implies (leq n m)
              (equal (replace n (head x m) x)
                     (fix x))))

(prove-lemma replace-associativity (rewrite)
     (equal (replace n x (replace n y z))
            (replace n (replace n x y) z)))

(prove-lemma replace-leq1 (rewrite)
     (implies (leq j i)
              (equal (replace i x (replace j y z))
                     (replace i x z)))
     ((disable times-commutativity)))

(prove-lemma replace-leq (rewrite)
     (implies (leq i j)
              (equal (replace i (replace j x y) z)
                     (replace i x z)))
     ((disable times-commutativity)))

(prove-lemma head-app (rewrite)
     (implies (leq i j)
              (equal (head (app j x y) i) (head x i))))

; a few basic lemmas about bcar and bcdr.  They are useful in induction
; proofs with bcar and bcdr disabled.
(prove-lemma bcar-nonnumberp (rewrite)
     (implies (not (numberp x))
              (equal (bcar x) 0)))

(prove-lemma bcdr-nonnumberp (rewrite)
     (implies (not (numberp x))
              (equal (bcdr x) 0)))

(prove-lemma bcar-1 (rewrite)
     (and (equal (bcar (times 2 x)) 0)
          (equal (bcar (add1 (times 2 x))) 1)))

(prove-lemma bcdr-1 (rewrite)
     (and (equal (bcdr (times 2 x)) (fix x))
          (equal (bcdr (add1 (times 2 x))) (fix x))))

(prove-lemma bcar-2 (rewrite)
     (equal (bcar (plus x (times 2 y))) (bcar x)))

(prove-lemma bcdr-2 (rewrite)
     (equal (bcdr (plus x (times 2 y))) (plus (bcdr x) y)))

; the first "bit" of head.
(prove-lemma bcar-head (rewrite)
     (equal (bcar (head x n))
```

```
           (if (zerop n) 0 (bcar x))))

; the first "bit" of app.
(prove-lemma bcar-app (rewrite)
     (equal (bcar (app n x y))
            (if (zerop n) (bcar y) (bcar x)))
     ((disable times-commutativity)))

; the first "bit" of replace.
(prove-lemma bcar-replace (rewrite)
     (equal (bcar (replace n x y))
            (if (zerop n) (bcar y) (bcar x)))
     ((disable times-commutativity remainder quotient)))

(prove-lemma bcar-lessp (rewrite)
     (lessp (bcar x) 2))

(prove-lemma bcdr-lessp (rewrite)
     (equal (lessp (bcdr x) y)
            (lessp x (times 2 y))))

(prove-lemma app-associativity (rewrite)
     (equal (app n1 x (app n2 y z))
            (app (plus n1 n2) (app n1 x y) z))
     ((enable times-distributes-plus)))

(prove-lemma app-head-tail (rewrite)
     (equal (app n (head x n) (tail x n))
            (fix x)))

(prove-lemma head-app-head-tail (rewrite)
     (equal (app m (head x m) (head (tail x m) n))
            (head x (plus m n)))
     ((use (app-head-tail (n m) (x (head x (plus m n)))))
      (disable remainder quotient)))

(disable head)
(disable tail)
(disable bcar)
(disable bcdr)

(prove-lemma app-nat-rangep (rewrite)
     (implies (leq m n)
              (equal (nat-rangep (app m x y) n)
                     (nat-rangep y (difference n m))))
     ((enable exp2-leq)))

(prove-lemma replace-nat-rangep (rewrite)
     (implies (leq m n)
              (equal (nat-rangep (replace m x y) n)
                     (nat-rangep y n)))
     ((enable equal-iff exp2-leq)
      (disable times-commutativity)))
```

```
(disable app)
(disable replace)

;                     BITN
(prove-lemma bitn-0 (rewrite)
     (and (equal (bitn 0 n) 0)
          (implies (not (numberp x))
                   (equal (bitn x i) 0))))

(prove-lemma bitn-0-1 (rewrite)
     (lessp (bitn x n) 2))

(prove-lemma bitn-head (rewrite)
     (equal (bitn (head x i) j)
            (if (lessp j i) (bitn x j) 0)))

(prove-lemma bitn-tail (rewrite)
     (equal (bitn (tail x i) j)
            (bitn x (plus i j))))

(prove-lemma bitn-app (rewrite)
     (equal (bitn (app n x y) k)
            (if (lessp k n)
                (bitn x k)
              (bitn y (difference k n))))
     ((disable times-commutativity)))

(prove-lemma bitn-replace (rewrite)
     (equal (bitn (replace n x y) k)
            (if (lessp k n) (bitn x k) (bitn y k)))
     ((disable times-commutativity)))

(disable bitn)

(prove-lemma bitn-lognot (rewrite)
     (implies (nat-rangep x n)
              (equal (bitn (lognot n x) i)
                     (if (lessp i n) (b-not (bitn x i)) 0)))
     ((use (bv-bitn-not (x (nat-to-bv-sized x n)) (n i)))))

;               LOGAND, LOGOR, AND LOGEOR.
(prove-lemma logor-equal-0 (rewrite)
     (equal (equal (logor x y) 0)
            (and (zerop x) (zerop y)))
     ((enable bcar bcdr)))

(prove-lemma logeor-equal-0 (rewrite)
     (equal (equal (logeor x y) 0)
            (equal (fix x) (fix y)))
     ((enable bcar bcdr)
      (disable remainder quotient)))

(prove-lemma logand-commutativity (rewrite)
     (equal (logand x y) (logand y x)))
```

```
(prove-lemma logor-commutativity (rewrite)
     (equal (logor x y) (logor y x)))

(prove-lemma logeor-commutativity (rewrite)
     (equal (logeor x y) (logeor y x)))

(prove-lemma logand-logor (rewrite)
     (equal (logand x (logor y z))
            (logor (logand x y) (logand x z))))

(prove-lemma logand-logeor (rewrite)
     (equal (logand x (logeor y z))
            (logeor (logand x y) (logand x z))))

(prove-lemma logand-uint-la (rewrite)
     (equal (nat-to-uint (logand (sub1 (exp 2 i)) y))
            (remainder (nat-to-uint y) (exp 2 i)))
     ((induct (mod-eq i x y))
      (enable bcar bcdr)
      (disable remainder quotient)))

(prove-lemma logand-uint (rewrite)
     (implies (equal (exp 2 (log 2 (add1 x))) (add1 x))
              (equal (nat-to-uint (logand x y))
                     (remainder (nat-to-uint y) (add1 x))))
     ((use (logand-uint-la (i (log 2 (add1 x)))))
      (disable logand-uint-la)))

;            HEAD WITH READ/WRITE AND READM/WRITEM
(prove-lemma head-readp (rewrite)
     (implies (leq n k)
              (equal (readp (head x k) n map)
                     (readp x n map))))

(prove-lemma head-pc-readp (rewrite)
     (implies (leq n k)
              (equal (pc-readp (head x k) n map)
                     (pc-readp x n map))))

(prove-lemma head-writep (rewrite)
     (implies (leq n k)
              (equal (writep (head x k) n map)
                     (writep x n map))))

(prove-lemma head-read (rewrite)
     (implies (leq n k)
              (equal (read (head x k) n bt)
                     (read x n bt))))

(prove-lemma head-pc-read (rewrite)
     (implies (leq n k)
              (equal (pc-read (head x k) n bt)
                     (pc-read x n bt))))
```

```
(prove-lemma head-write (rewrite)
     (implies (leq n k)
              (equal (write value (head x k) n bt)
                     (write value x n bt))))

(prove-lemma head-byte-readp (rewrite)
     (equal (byte-readp (head x 32) mem)
            (byte-readp x mem))
     ((enable byte-readp)))

(prove-lemma head-pc-byte-readp (rewrite)
     (equal (pc-byte-readp (head x 32) mem)
            (pc-byte-readp x mem))
     ((enable pc-byte-readp)))

(prove-lemma head-byte-writep (rewrite)
     (equal (byte-writep (head x 32) mem)
            (byte-writep x mem))
     ((enable byte-writep)))

(prove-lemma head-byte-read (rewrite)
     (equal (byte-read (head x 32) mem)
            (byte-read x mem)))

(prove-lemma head-byte-pc-read (rewrite)
     (equal (pc-byte-read (head x 32) mem)
            (pc-byte-read x mem)))

(prove-lemma head-byte-write (rewrite)
     (equal (byte-write value (head x 32) mem)
            (byte-write value x mem)))

(prove-lemma head-read-memp (rewrite)
     (equal (read-memp (head x 32) mem k)
            (read-memp x mem k)))

(prove-lemma head-pc-read-memp (rewrite)
     (equal (pc-read-memp (head x 32) mem k)
            (pc-read-memp x mem k)))

(prove-lemma head-write-memp (rewrite)
     (equal (write-memp (head x 32) mem k)
            (write-memp x mem k)))

(prove-lemma head-read-mem (rewrite)
     (equal (read-mem (head x 32) mem k)
            (read-mem x mem k)))

(prove-lemma head-pc-read-mem (rewrite)
     (equal (pc-read-mem (head x 32) mem k)
            (pc-read-mem x mem k)))

(prove-lemma head-write-mem (rewrite)
```

```
     (equal (write-mem value (head x 32) mem k)
            (write-mem value x mem k)))

(prove-lemma head-readm-mem (rewrite)
     (equal (readm-mem opsz (head x 32) mem n)
            (readm-mem opsz x mem n)))

;                  NAT-RANGEP
(prove-lemma nat-rangep-of-0 (rewrite)
     (nat-rangep 0 n))

(prove-lemma nat-rangep-ub (rewrite)
     (equal (nat-rangep (sub1 (exp 2 k)) n)
            (not (lessp n k))))

(prove-lemma nat-rangep-0 (rewrite)
     (equal (nat-rangep x 0) (zerop x)))

(prove-lemma nat-plus-rangep (rewrite)
     (implies (nat-rangep x k)
              (and (nat-rangep x (plus k j))
                   (nat-rangep x (plus j k)))))

(prove-lemma sub1-nat-rangep (rewrite)
     (implies (nat-rangep x n)
              (nat-rangep (sub1 x) n)))

(prove-lemma sub1-times2-nat-rangep (rewrite)
     (equal (nat-rangep (sub1 (times 2 x)) (add1 n))
            (nat-rangep (sub1 x) n)))

(prove-lemma difference-nat-rangep (rewrite)
     (implies (nat-rangep x n)
              (nat-rangep (difference x y) n)))

(prove-lemma quotient-nat-rangep (rewrite)
     (implies (nat-rangep x n)
              (nat-rangep (quotient x y) n)))

(prove-lemma times-exp2-nat-rangep (rewrite)
     (equal (nat-rangep (times x (exp 2 i)) n)
            (nat-rangep x (difference n i))))

(prove-lemma logand-nat-rangep (rewrite)
     (implies (or (nat-rangep x n)
                  (nat-rangep y n))
              (nat-rangep (logand x y) n))
     ((induct (mod-eq n x y))))

(prove-lemma logor-nat-rangep (rewrite)
     (equal (nat-rangep (logor x y) n)
            (and (nat-rangep x n)
                 (nat-rangep y n)))
     ((induct (mod-eq n x y))))
```

```
(prove-lemma logeor-nat-rangep (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n))
              (nat-rangep (logeor x y) n))
     ((induct (mod-eq n x y))))

(prove-lemma head-nat-rangep (rewrite)
     (implies (leq m n)
              (nat-rangep (head x m) n)))

(prove-lemma tail-nat-rangep (rewrite)
     (equal (nat-rangep (tail x m) n)
            (nat-rangep x (plus m n))))

(prove-lemma read-rn-nat-rangep (rewrite)
     (nat-rangep (read-rn n rn rfile) n))

(prove-lemma byte-read-nat-rangep (rewrite)
     (implies (leq 8 n)
              (nat-rangep (byte-read x mem) n)))

(prove-lemma pc-byte-read-nat-rangep (rewrite)
     (nat-rangep (pc-byte-read x mem) 8))

(disable nat-rangep)

(prove-lemma mulu-nat-rangep (rewrite)
     (nat-rangep (mulu n x y i) n))

(prove-lemma muls-nat-rangep (rewrite)
     (nat-rangep (muls n x y i) n))

(prove-lemma quot-nat-rangep (rewrite)
     (nat-rangep (quot n x y) n))

(prove-lemma rem-nat-rangep (rewrite)
     (nat-rangep (rem n x y) n))

(prove-lemma iquot-nat-rangep (rewrite)
     (nat-rangep (iquot n x i y j) n))

(prove-lemma irem-nat-rangep (rewrite)
     (nat-rangep (irem n x i y j) n))

(prove-lemma neg-nat-rangep (rewrite)
     (nat-rangep (neg n x) n))

;                          EXT
(prove-lemma ext-0 (rewrite)
     (equal (ext n 0 size) 0))

(prove-lemma ext-lemma (rewrite)
     (implies (leq size n)
```

```
             (equal (ext n x size) (head x size))))

(prove-lemma head-ext (rewrite)
     (implies (and (leq i j) (leq j k))
              (equal (head (ext j x k) i)
                     (head x i))))

(prove-lemma ext-nat-rangep (rewrite)
     (nat-rangep (ext n x size) size)
     ((use (exp2-leq (i n) (j size)))))

(disable ext)

;                      CVZNX-FLAGS
(prove-lemma set-cvznx-c (rewrite)
     (equal (ccr-c (set-cvznx (cvznx c v z n x) ccr))
            (fix-bit c)))

(prove-lemma set-cvznx-v (rewrite)
     (equal (ccr-v (set-cvznx (cvznx c v z n x) ccr))
            (fix-bit v)))

(prove-lemma set-cvznx-z (rewrite)
     (equal (ccr-z (set-cvznx (cvznx c v z n x) ccr))
            (fix-bit z)))

(prove-lemma set-cvznx-n (rewrite)
     (equal (ccr-n (set-cvznx (cvznx c v z n x) ccr))
            (fix-bit n)))

(prove-lemma set-cvznx-x (rewrite)
     (equal (ccr-x (set-cvznx (cvznx c v z n x) ccr))
            (fix-bit x)))

; the new cvznx-flags replaces the old ones.
(prove-lemma set-set-cvznx1 (rewrite)
     (equal (set-cvznx x (set-cvznx y ccr))
            (set-cvznx x ccr)))

(prove-lemma set-cvznx-ccr (rewrite)
     (equal (set-cvznx ccr ccr)
            (fix ccr)))

(prove-lemma set-set-cvznx2 (rewrite)
     (equal (set-cvznx (set-cvznx x ccr) ccr)
            (set-cvznx x ccr)))

(prove-lemma set-cvznx-nat-rangep (rewrite)
     (equal (nat-rangep (set-cvznx x ccr) 8)
            (nat-rangep ccr 8)))

(disable ccr-c)
(disable ccr-v)
(disable ccr-z)
```

```
(disable ccr-n)
(disable ccr-x)
(disable cvznx)
(disable set-cvznx)

;                       MC ACCESSORS
(prove-lemma mc-status-rewrite (rewrite)
     (equal (mc-status (mc-state status rfile pc ccr mem))
            status))

(prove-lemma mc-rfile-rewrite (rewrite)
     (equal (mc-rfile (mc-state status rfile pc ccr mem))
            rfile))

(prove-lemma mc-pc-rewrite (rewrite)
     (implies (nat-rangep pc 32)
              (equal (mc-pc (mc-state status rfile pc ccr mem))
                     (fix pc))))

(prove-lemma mc-ccr-rewrite (rewrite)
     (implies (nat-rangep ccr 8)
              (equal (mc-ccr (mc-state status rfile pc ccr mem))
                     (fix ccr))))

(prove-lemma mc-mem-rewrite (rewrite)
     (equal (mc-mem (mc-state status rfile pc ccr mem))
            mem))

; lemmas about those mc accessors.
(prove-lemma mc-pc-rangep (rewrite)
     (nat-rangep (mc-pc s) 32))

(prove-lemma mc-ccr-rangep (rewrite)
     (nat-rangep (mc-ccr s) 8))

(disable mc-status)
(disable mc-rfile)
(disable mc-pc)
(disable mc-ccr)
(disable mc-mem)
(disable mc-state)

;           THE BASIC GET-NTH/PUT-NTH RELATIONS
;
(prove-lemma get-nth-0 (rewrite)
     (implies (not (numberp i))
              (equal (get-nth i lst) (get-nth 0 lst))))

(prove-lemma put-nth-0 (rewrite)
     (implies (not (numberp i))
              (equal (put-nth v i lst) (put-nth v 0 lst))))

(disable get-nth-0)
(disable put-nth-0)
```

```
(prove-lemma get-nth-nil (rewrite)
     (and (equal (get-nth n nil) 0)
          (equal (get-nth n 0) 0)))

(prove-lemma get-put (rewrite)
     (equal (get-nth n (put-nth value m lst))
            (if (equal (fix m) (fix n))
                value
              (get-nth n lst))))

(prove-lemma put-put (rewrite)
     (equal (put-nth value1 n (put-nth value n lst))
            (put-nth value1 n lst)))

(prove-lemma put-get (rewrite)
     (implies (lessp n (len lst))
              (equal (put-nth (get-nth n lst) n lst) lst)))

(prove-lemma put-nth-len (rewrite)
     (equal (len (put-nth value n lst))
            (if (lessp n (len lst)) (len lst) (add1 n))))

;            THE BASIC READ-RN/WRITE-RN RELATIONS
;
(prove-lemma head-read-rn (rewrite)
     (implies (leq n1 n2)
              (equal (head (read-rn n2 rn rfile) n1)
                     (read-rn n1 rn rfile))))

; stop proving.
(defn read-write-rn-end (n2 rn n1 value rm rfile)
  (read-rn n2 rn (write-rn n1 value rm rfile)))

(prove-lemma read-write-rn (rewrite)
     (equal (read-rn n2 rn (write-rn n1 value rm rfile))
            (if (equal (fix rm) (fix rn))
                (if (leq n2 n1)
                    (head value n2)
                  (replace n1 value (read-rn n2 rn rfile)))
              (read-rn n2 rn rfile))))

(prove-lemma write-write-rn (rewrite)
     (implies (leq n1 n2)
              (equal (write-rn n2 v2 rn (write-rn n1 v1 rn rfile))
                     (write-rn n2 v2 rn rfile)))))

(prove-lemma write-rn-len (rewrite)
     (equal (len (write-rn oplen value rn rfile))
            (if (lessp rn (len rfile)) (len rfile) (add1 rn))))

(disable read-rn)
(disable write-rn)
```

```
;             THE BASIC READM-RN/WRITEM-RN EVENTS
;
(prove-lemma readm-rn-len (rewrite)
     (equal (len (readm-rn oplen rnlst rfile))
            (len rnlst)))

(defn get-vlst (oplen value rn rnlst vlst)
  (if (nlistp rnlst)
      value
    (if (equal (fix rn) (fix (car rnlst)))
        (get-vlst oplen (head (car vlst) oplen) rn (cdr rnlst) (cdr vlst))
      (get-vlst oplen value rn (cdr rnlst) (cdr vlst)))))

; we deliberately put on the hypothesis (numberp value) to restrict the use
; of this lemma.  The dead loop situation is thereby avoided.
(prove-lemma get-vlst-member (rewrite)
     (implies (numberp value)
              (equal (get-vlst oplen value rn rnlst vlst)
                     (if (n-member rn rnlst)
                         (get-vlst oplen nil rn rnlst vlst)
                       value))))

; stop proving.
(defn read-writem-rn-end (n1 rn n2 vlst rnlst rfile)
  (read-rn n1 rn (writem-rn n2 vlst rnlst rfile)))

(prove-lemma read-writem-rn (rewrite)
     (equal (read-rn n1 rn (writem-rn n2 vlst rnlst rfile))
            (if (n-member rn rnlst)
                (if (and (leq n1 n2)
                         (leq n2 32))
                    (get-vlst n1 0 rn rnlst vlst)
                  (read-writem-rn-end n1 rn n2 vlst rnlst rfile))
              (read-rn n1 rn rfile)))
     ((induct (writem-rn n2 vlst rnlst rfile))))

(disable read-writem-rn-end)

(prove-lemma readm-write-rn (rewrite)
     (implies (not (n-member rn rnlst))
              (equal (readm-rn n1 rnlst (write-rn n2 value rn rfile))
                     (readm-rn n1 rnlst rfile))))

(prove-lemma read-rn-0 (rewrite)
     (implies (not (numberp rn))
              (equal (read-rn oplen rn rfile)
                     (read-rn oplen 0 rfile)))
     ((enable read-rn)))

(prove-lemma get-vlst-readm-rn (rewrite)
     (implies (leq n1 n2)
              (equal (get-vlst n1 0 rn rnlst (readm-rn n2 rnlst rfile))
                     (if (n-member rn rnlst)
                         (read-rn n1 rn rfile)
```

```
                    0))))

(disable read-rn-0)
(disable get-vlst-member)
(disable get-vlst)

;           THE BASIC READ-MEM/WRITE-MEM EVENTS
;
(prove-lemma pc-readp->readp (rewrite)
     (implies (pc-readp x n map)
              (readp x n map)))

(prove-lemma writep->readp (rewrite)
     (implies (writep x n map)
              (readp x n map)))

; a read right after a write at the same location returns the new value
; just writen.  But a read right after a write at a different location
; has the same effect as a read made on the original binary tree.
(defn modn-eq (n x y)
  (if (zerop n)
      t
    (and (equal (bitn x (sub1 n)) (bitn y (sub1 n)))
         (modn-eq (sub1 n) x y))))

(prove-lemma read-write (rewrite)
     (equal (read y n (write value x n bt))
            (if (modn-eq n x y)
                value
              (read y n bt))))

; pc should always be able to go through the memory.
(prove-lemma pc-read-write (rewrite)
     (implies (and (writep x n map)
                   (pc-readp y n map))
              (equal (pc-read y n (write value x n bt))
                     (pc-read y n bt))))

; write twice at the same location is equivalent to the second write on
; the original binary tree.
(prove-lemma write-write-la ()
     (equal (write v2 y n (write v1 x n bt))
            (if (not (modn-eq n x y))
                (write v1 x n (write v2 y n bt))
              (write v2 y n bt))))

(prove-lemma write-write (rewrite)
     (equal (write v2 x n (write v1 x n bt))
            (write v2 x n bt)))

(disable pc-read)

(prove-lemma plus-times-equal (rewrite)
     (implies (and (lessp a k)
```

```
                (lessp b k))
           (equal (equal (plus a (times i k)) (plus b (times j k)))
                  (and (equal (fix a) (fix b))
                       (equal (fix i) (fix j)))))
     ((induct (difference i j))))

(prove-lemma app-cancel (rewrite)
     (equal (equal (app n x y) (app n x1 y1))
            (and (equal (head x n) (head x1 n))
                 (equal (fix y) (fix y1))))
     ((enable app)))

(disable plus-times-equal)

(prove-lemma head-app-cancel (rewrite)
     (equal (equal (head x n) (app n x1 y1))
            (and (equal (head x n) (head x1 n))
                 (zerop y1)))
     ((use (app-cancel (y 0)))))

(prove-lemma head-recursion (rewrite)
     (equal (head x (add1 n))
            (app n (head x n) (bitn x n)))
     ((enable head bitn app tail bcar)))

(prove-lemma modn-eq-equal (rewrite)
     (equal (modn-eq n x y)
            (equal (head x n) (head y n))))

(prove-lemma ext-equal (rewrite)
     (implies (and (leq n size)
                   (not (zerop n)))
              (equal (equal (ext n x size) (ext n y size))
                     (equal (head x n) (head y n))))
     ((enable ext)))

(prove-lemma ext-equal-0 (rewrite)
     (implies (and (leq n size)
                   (not (zerop n)))
              (equal (equal (ext n x size) 0)
                     (equal (head x n) 0)))
     ((use (ext-equal (y 0)))))

(disable head-recursion)

;               BYTE-READ/BYTE-WRITE
;
; pc-byte-readp --> byte-readp.
(prove-lemma pc-byte-readp->byte-readp (rewrite)
     (implies (pc-byte-readp x mem)
              (byte-readp x mem)))

; byte-writep --> byte-readp.
(prove-lemma byte-writep->readp (rewrite)
```

```
     (implies (byte-writep x mem)
              (byte-readp x mem)))

(prove-lemma byte-read-write (rewrite)
     (equal (byte-read x (byte-write v y mem))
            (if (mod32-eq x y)
                (if (nat-rangep v 8) (fix v) (head v 8))
              (byte-read x mem))))

(prove-lemma byte-write-write-la ()
     (equal (byte-write v2 y (byte-write v1 x mem))
            (if (not (mod32-eq x y))
                (byte-write v1 x (byte-write v2 y mem))
              (byte-write v2 y mem)))
     ((use (write-write-la (v1 (head v1 8)) (v2 (head v2 8))
                           (n 32) (bt (cdr mem))))))

(prove-lemma byte-write-write (rewrite)
     (equal (byte-write v2 x (byte-write v1 x mem))
            (byte-write v2 x mem)))

(prove-lemma pc-byte-read-write (rewrite)
     (implies (and (byte-writep x mem)
                   (pc-byte-readp y mem))
              (equal (pc-byte-read y (byte-write value x mem))
                     (pc-byte-read y mem))))

; write on memory does not change the properties of the memory.
(prove-lemma byte-write-maintain-pc-byte-readp (rewrite)
     (equal (pc-byte-readp x (byte-write value y mem))
            (pc-byte-readp x mem)))

(prove-lemma byte-write-maintain-byte-readp (rewrite)
     (equal (byte-readp x (byte-write value y mem))
            (byte-readp x mem)))

(prove-lemma byte-write-maintain-byte-writep (rewrite)
     (equal (byte-writep x (byte-write value y mem))
            (byte-writep x mem)))

; a lemma useful to deal with read-mem/write-mem.
(prove-lemma byte-write-app (rewrite)
     (implies (leq 8 n)
              (equal (byte-write (app n x y) addr mem)
                     (byte-write x addr mem))))

(disable pc-byte-readp)
(disable byte-readp)
(disable byte-writep)
(disable pc-byte-read)
(disable byte-read)
(disable byte-write)

;             THE BASIC READ-MEM/WRITE-MEM EVENTS
```

```
(defn mem-induct0 (k n)
  (if (zerop k)
      0
    (mem-induct0 (sub1 k) (difference n 8))))

(prove-lemma pc-read-mem-nat-rangep (rewrite)
     (implies (equal n (times 8 k))
              (nat-rangep (pc-read-mem x mem k) n)))

(prove-lemma read-mem-nat-rangep (rewrite)
     (implies (leq (times 8 k) n)
              (nat-rangep (read-mem x mem k) n))
     ((induct (mem-induct0 k n))))

; write on memory does not change the properties of the memory. i.e. RAM is
; still RAM, ROM is still ROM, and UNAVAILABLE is still unavailable.
(prove-lemma write-mem-maintain-pc-byte-readp (rewrite)
     (equal (pc-byte-readp addr (write-mem value x mem k))
            (pc-byte-readp addr mem)))

(prove-lemma write-mem-maintain-byte-readp (rewrite)
     (equal (byte-readp addr (write-mem value x mem k))
            (byte-readp addr mem)))

(prove-lemma write-mem-maintain-byte-writep (rewrite)
     (equal (byte-writep addr (write-mem value x mem k))
            (byte-writep addr mem)))

(prove-lemma byte-write-maintain-pc-read-memp (rewrite)
     (equal (pc-read-memp addr (byte-write value x mem) n)
            (pc-read-memp addr mem n)))

(prove-lemma byte-write-maintain-read-memp (rewrite)
     (equal (read-memp addr (byte-write value x mem) n)
            (read-memp addr mem n)))

(prove-lemma byte-write-maintain-write-memp (rewrite)
     (equal (write-memp addr (byte-write value x mem) n)
            (write-memp addr mem n)))

(prove-lemma write-mem-maintain-pc-read-memp (rewrite)
     (equal (pc-read-memp addr (write-mem value x mem m) n)
            (pc-read-memp addr mem n)))

(prove-lemma write-mem-maintain-read-memp (rewrite)
     (equal (read-memp addr (write-mem value x mem m) n)
            (read-memp addr mem n)))

(prove-lemma write-mem-maintain-write-memp (rewrite)
     (equal (write-memp addr (write-mem value x mem m) n)
            (write-memp addr mem n)))

; if it is pc-readable, then it must be readable.
(prove-lemma pc-read-memp->read-memp (rewrite)
```

```
     (implies (pc-read-memp x mem n)
              (read-memp x mem n)))

; if it is writeable, then it must be readable.
(prove-lemma write-memp->read-memp (rewrite)
     (implies (write-memp x mem n)
              (read-memp x mem n)))

; program segments are never changed as the memory are changed.
(prove-lemma pc-byte-read-write-mem (rewrite)
     (implies (and (write-memp x mem n)
                   (pc-byte-readp y mem))
              (equal (pc-byte-read y (write-mem value x mem n))
                     (pc-byte-read y mem)))
     ((induct (write-mem value x mem n))))

(prove-lemma pc-read-mem-byte-write (rewrite)
     (implies (and (byte-writep x mem)
                   (pc-read-memp y mem n))
              (equal (pc-read-mem y (byte-write value x mem) n)
                     (pc-read-mem y mem n))))

(prove-lemma pc-read-mem-write-mem (rewrite)
     (implies (and (write-memp x mem m)
                   (pc-read-memp y mem n))
              (equal (pc-read-mem y (write-mem value x mem m) n)
                     (pc-read-mem y mem n))))

; a byte-read vs a multi-write.
; stop proving.
(defn byte-read-write-mem-end (x value y mem k)
  (byte-read x (write-mem value y mem k)))

(prove-lemma byte-read-write-mem (rewrite)
     (equal (byte-read x (write-mem value y mem k))
            (if (disjoint x 1 y k)
                (byte-read x mem)
              (byte-read-write-mem-end x value y mem k)))
     ((induct (write-mem value y mem k))))

; a multi-read vs a byte-write.
; stop proving.
(defn read-mem-byte-write-end (x n value y mem)
  (read-mem x (byte-write value y mem) n))

(prove-lemma read-mem-byte-write (rewrite)
     (equal (read-mem x (byte-write value y mem) n)
            (if (disjoint0 x n y)
                (read-mem x mem n)
              (read-mem-byte-write-end x n value y mem))))

; a multi-read vs a multi-write.
; stop proving.
(defn read-write-mem-end (x value y mem m n)
```

```
  (read-mem x (write-mem value y mem m) n))

(prove-lemma read-write-mem1 (rewrite)
     (equal (read-mem x (write-mem value y mem k) n)
            (if (disjoint x n y k)
                (read-mem x mem n)
              (read-write-mem-end x value y mem k n))))

(prove-lemma head-sub1-lessp (rewrite)
     (equal (lessp (sub1 (head x n)) x)
            (not (zerop x))))

(prove-lemma byte-read-write-mem-lemma (rewrite)
     (implies (and (nat-rangep y 32)
                   (leq n y))
              (equal (byte-read (add 32 x y) (write-mem value x mem n))
                     (byte-read (add 32 x y) mem)))
     ((induct (write-mem value x mem n))
      (enable nat-rangep)))

(prove-lemma read-write-mem2 (rewrite)
     (implies (uint-rangep n 32)
              (equal (read-mem x (write-mem value x mem n) n)
                     (head value (times 8 n))))
     ((induct (write-mem value x mem n))
      (enable nat-rangep) (disable read-write-mem1)))

(disable read-write-mem-end)
(disable read-mem-byte-write-end)
(disable byte-read-write-mem-end)

; write to the same location twice, only the second counts.
(prove-lemma write-mem-byte-write (rewrite)
     (equal (write-mem v2 y (byte-write v1 x mem) n)
            (if (disjoint0 y n x)
                (byte-write v1 x (write-mem v2 y mem n))
              (write-mem v2 y mem n)))
     ((induct (write-mem v2 y mem n))
      (use (byte-write-write-la (y (add 32 y (sub1 n)))))))

(defn write-write-induct (v1 v2 n)
  (if (zerop n)
      t
    (write-write-induct (tail v1 (b)) (tail v2 (b)) (sub1 n))))

(prove-lemma write-write-mem (rewrite)
     (equal (write-mem v2 x (write-mem v1 x mem n) n)
            (write-mem v2 x mem n))
     ((induct (write-write-induct v1 v2 n))))

;                LEMMAS ABOUT BITP
(prove-lemma bitp-fix-bit (rewrite)
     (implies (bitp x)
              (equal (fix-bit x) x)))
```

```
(prove-lemma fix-bit-bitp (rewrite)
     (bitp (fix-bit b)))

(prove-lemma bitn-bitp (rewrite)
     (bitp (bitn x n)))

(prove-lemma add-c-bitp (rewrite)
     (bitp (add-c n x y)))

(prove-lemma add-v-bitp (rewrite)
     (bitp (add-v n x y)))

(prove-lemma add-z-bitp (rewrite)
     (bitp (add-z n x y)))

(prove-lemma add-n-bitp (rewrite)
     (bitp (add-n n x y)))

(prove-lemma addx-c-bitp (rewrite)
     (bitp (addx-c n x s d)))

(prove-lemma addx-v-bitp (rewrite)
     (bitp (addx-v n x s d)))

(prove-lemma addx-z-bitp (rewrite)
     (bitp (addx-z n z x s d)))

(prove-lemma addx-n-bitp (rewrite)
     (bitp (addx-n n x s d)))

(prove-lemma sub-c-bitp (rewrite)
     (bitp (sub-c n x y)))

(prove-lemma sub-v-bitp (rewrite)
     (bitp (sub-v n x y)))

(prove-lemma sub-z-bitp (rewrite)
     (bitp (sub-z n x y)))

(prove-lemma sub-n-bitp (rewrite)
     (bitp (sub-n n x y)))

(prove-lemma subx-c-bitp (rewrite)
     (bitp (subx-c n x s d)))

(prove-lemma subx-v-bitp (rewrite)
     (bitp (subx-v n x s d)))

(prove-lemma subx-z-bitp (rewrite)
     (bitp (subx-z n z x s d)))

(prove-lemma subx-n-bitp (rewrite)
     (bitp (subx-n n x s d)))
```

```
(prove-lemma and-z-bitp (rewrite)
     (bitp (and-z n s d)))

(prove-lemma and-n-bitp (rewrite)
     (bitp (and-n n s d)))

(prove-lemma mulu-v-bitp (rewrite)
     (bitp (mulu-v n s d i)))

(prove-lemma mulu-z-bitp (rewrite)
     (bitp (mulu-z n s d i)))

(prove-lemma mulu-n-bitp (rewrite)
     (bitp (mulu-n n s d i)))

(prove-lemma muls-v-bitp (rewrite)
     (bitp (muls-v n s d i)))

(prove-lemma muls-z-bitp (rewrite)
     (bitp (muls-z n s d i)))

(prove-lemma muls-n-bitp (rewrite)
     (bitp (muls-n n s d i)))

(prove-lemma or-z-bitp (rewrite)
     (bitp (or-z n s d)))

(prove-lemma or-n-bitp (rewrite)
     (bitp (or-n n s d)))

(prove-lemma divs-z-bitp (rewrite)
     (bitp (divs-z n s i d j)))

(prove-lemma divs-n-bitp (rewrite)
     (bitp (divs-n n s i d j)))

(prove-lemma divu-z-bitp (rewrite)
     (bitp (divu-z n s d)))

(prove-lemma divu-n-bitp (rewrite)
     (bitp (divu-n n s d)))

(prove-lemma rol-c-bitp (rewrite)
     (bitp (rol-c len x cnt)))

(prove-lemma rol-z-bitp (rewrite)
     (bitp (rol-z len x cnt)))

(prove-lemma rol-n-bitp (rewrite)
     (bitp (rol-n len x cnt)))

(prove-lemma ror-c-bitp (rewrite)
     (bitp (ror-c len x cnt)))
```

```
(prove-lemma ror-z-bitp (rewrite)
     (bitp (ror-z len x cnt)))

(prove-lemma ror-n-bitp (rewrite)
     (bitp (ror-n len x cnt)))

(prove-lemma lsl-c-bitp (rewrite)
     (bitp (lsl-c len x cnt)))

(prove-lemma lsl-z-bitp (rewrite)
     (bitp (lsl-z len x cnt)))

(prove-lemma lsl-n-bitp (rewrite)
     (bitp (lsl-n len x cnt)))

(prove-lemma lsr-c-bitp (rewrite)
     (bitp (lsr-c len x cnt)))

(prove-lemma lsr-z-bitp (rewrite)
     (bitp (lsr-z len x cnt)))

(prove-lemma lsr-n-bitp (rewrite)
     (bitp (lsr-n len x cnt)))

(prove-lemma asl-c-bitp (rewrite)
     (bitp (asl-c len x cnt)))

(prove-lemma asl-v-bitp (rewrite)
     (bitp (asl-v len x cnt)))

(prove-lemma asl-z-bitp (rewrite)
     (bitp (asl-z len x cnt)))

(prove-lemma asl-n-bitp (rewrite)
     (bitp (asl-n len x cnt)))

(prove-lemma asr-c-bitp (rewrite)
     (bitp (asr-c len x cnt)))

(prove-lemma asr-z-bitp (rewrite)
     (bitp (asr-z len x cnt)))

(prove-lemma asr-n-bitp (rewrite)
     (bitp (asr-n len x cnt)))

(prove-lemma roxl-c-bitp (rewrite)
     (bitp (roxl-c len opd cnt x)))

(prove-lemma roxl-z-bitp (rewrite)
     (bitp (roxl-z len opd cnt x))
     ((disable roxl)))

(prove-lemma roxl-n-bitp (rewrite)
```

```
     (bitp (roxl-n len opd cnt x))
     ((disable roxl)))

(prove-lemma roxr-c-bitp (rewrite)
     (bitp (roxr-c len opd cnt x)))

(prove-lemma roxr-z-bitp (rewrite)
     (bitp (roxr-z len opd cnt x))
     ((disable roxr)))

(prove-lemma roxr-n-bitp (rewrite)
     (bitp (roxr-n len opd cnt x))
     ((disable roxr)))

(prove-lemma move-z-bitp (rewrite)
     (bitp (move-z oplen sopd)))

(prove-lemma move-n-bitp (rewrite)
     (bitp (move-n oplen sopd)))

(prove-lemma ext-z-bitp (rewrite)
     (bitp (ext-z oplen opd size)))

(prove-lemma ext-n-bitp (rewrite)
     (bitp (ext-n oplen opd size)))

(prove-lemma swap-z-bitp (rewrite)
     (bitp (swap-z opd)))

(prove-lemma swap-n-bitp (rewrite)
     (bitp (swap-n opd)))

(prove-lemma not-z-bitp (rewrite)
     (bitp (not-z oplen opd)))

(prove-lemma not-n-bitp (rewrite)
     (bitp (not-n oplen opd)))

(prove-lemma eor-z-bitp (rewrite)
     (bitp (eor-z n s d)))

(prove-lemma eor-n-bitp (rewrite)
     (bitp (eor-n n s d)))

;                          NEG
; after replacing sub by add and neg.  We can completely get rid of sub.
(prove-lemma neg-head (rewrite)
     (equal (neg n (head x n)) (neg n x))
     ((enable sub)))

(prove-lemma neg-neg (rewrite)
     (equal (neg n (neg n x)) (head x n)))

(prove-lemma neg-cancel (rewrite)
```

```
     (equal (add n x (neg n x)) 0))

(prove-lemma neg-add (rewrite)
     (equal (neg n (add n x y))
            (add n (neg n x) (neg n y))))

(prove-lemma sub-neg (rewrite)
     (equal (sub n y x)
            (add n x (neg n y))))

(disable neg)

; x - y = 0 <==> x = y.
(prove-lemma add-neg-0 (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n))
              (equal (equal (add n y (neg n x)) 0)
                     (equal (fix x) (fix y))))
     ((use (sub-equal-0))))

; y + (x - y) = x.
(prove-lemma add-neg-cancel (rewrite)
     (and (equal (add n y (add n x (neg n y)))
                 (head x n))
          (equal (add n y (add n (neg n y) x))
                 (head x n))))

(prove-lemma sub-leq-1 (rewrite)
     (implies (leq (neg n x) y)
              (and (not (lessp (difference y (neg n x)) (add n y x)))
                   (not (lessp (difference y (neg n x)) (add n x y))))))
     ((use (sub-leq-1a (x (neg n x))))))

(prove-lemma sub-leq-2 (rewrite)
     (implies (leq x y)
              (and (not (lessp (difference y x) (add n y (neg n x))))
                   (not (lessp (difference y x) (add n (neg n x) y))))))
     ((use (sub-leq-1a))))

(defn add-fringe (n x)
  (if (and (listp x)
           (equal (car x) 'add)
           (equal (cadr x) n))
      (append (add-fringe n (caddr x))
              (add-fringe n (cadddr x)))
    (cons x nil)))

(defn add-tree (n l)
  (if (nlistp l)
      ''0
    (if (nlistp (cdr l))
        (list 'head (car l) n)
      (if (nlistp (cddr l))
          (list 'add n (car l) (cadr l))
```

```
        (list 'add n (car l) (add-tree n (cdr l)))))))

(prove-lemma numberp-eval$-add (rewrite)
     (implies (equal (car x) 'add)
              (numberp (eval$ t x a))))

(prove-lemma numberp-eval$-add-tree (rewrite)
     (numberp (eval$ t (add-tree n l) a)))

(prove-lemma add-equal-cancel-1 (rewrite)
       (equal (equal (add n a b) (add n c (add n a d)))
              (equal (head b n) (add n c d))))

(prove-lemma eval$-add-member (rewrite)
     (implies (member e x)
              (equal (eval$ t (add-tree n x) a)
                     (add (eval$ t n a)
                          (eval$ t e a)
                          (eval$ t (add-tree n (delete e x)) a)))))

(prove-lemma add-tree-nat-rangep (rewrite)
     (nat-rangep (eval$ t (add-tree n x) a)
                 (eval$ t n a)))

(prove-lemma add-tree-append (rewrite)
     (equal (eval$ t (add-tree n (append x y)) a)
            (add (eval$ t n a)
                 (eval$ t (add-tree n x) a)
                 (eval$ t (add-tree n y) a))))

(prove-lemma add-tree-add-fringe (rewrite)
     (equal (eval$ t (add-tree n (add-fringe n x)) a)
            (head (eval$ t x a) (eval$ t n a)))
     ((induct (add-fringe n x))))

(defn add-tree-delete-cond (n x y)
  (list 'and
        (list 'numberp x)
        (list 'and
              (list 'nat-rangep x n)
              (list 'equal (add-tree n (delete x y)) ''0))))

(prove-lemma add-tree-delete-equal (rewrite)
     (implies (member x y)
              (equal (eval$ t (add-tree-delete-cond n x y) a)
                     (equal (eval$ t (add-tree n y) a)
                            (eval$ t x a)))))

(prove-lemma add-tree-bagdiff (rewrite)
     (implies (and (subbagp x y)
                   (subbagp x z))
              (equal (equal (eval$ t (add-tree n (bagdiff y x)) a)
                            (eval$ t (add-tree n (bagdiff z x)) a))
                     (equal (eval$ t (add-tree n y) a)
```

```
                          (eval$ t (add-tree n z) a)))))

(defn cancel-equal-add (x)
  (if (and (listp x) (equal (car x) 'equal))
      (if (and (listp (cadr x)) (equal (caadr x) 'add)
               (listp (caddr x)) (equal (caaddr x) 'add))
          (if (equal (cadadr x) (cadaddr x))
              (list 'equal
                    (add-tree
                     (cadadr x)
                     (bagdiff (add-fringe (cadadr x) (cadr x))
                              (bagint (add-fringe (cadadr x) (cadr x))
                                      (add-fringe (cadadr x) (caddr x)))))
                    (add-tree
                     (cadadr x)
                     (bagdiff (add-fringe (cadadr x) (caddr x))
                              (bagint (add-fringe (cadadr x) (cadr x))
                                      (add-fringe (cadadr x) (caddr x))))))
              x)
        (if (and (listp (cadr x)) (equal (caadr x) 'add)
                 (member (caddr x) (add-fringe (cadadr x) (cadr x))))
            (add-tree-delete-cond (cadadr x)
                                  (caddr x)
                                  (add-fringe (cadadr x) (cadr x)))
          (if (and (listp (caddr x)) (equal (caaddr x) 'add)
                   (member (cadr x) (add-fringe (cadaddr x) (caddr x))))
              (add-tree-delete-cond (cadaddr x)
                                    (cadr x)
                                    (add-fringe (cadaddr x) (caddr x)))
            x)))
    x))

(prove-lemma correctness-of-cancel-equal-add ((meta equal))
     (equal (eval$ t x a)
            (eval$ t (cancel-equal-add x) a))
     ((disable add-tree-delete-cond)))

(prove-lemma add-tree-delete (rewrite)
     (implies (member x y)
              (equal (eval$ t (add-tree n (delete x y)) a)
                     (sub (eval$ t n a)
                          (eval$ t x a)
                          (eval$ t (add-tree n y) a)))))

(defn cancel-add-neg (x)
  (if (and (listp x)
           (equal (car x) 'add)
           (listp (cdr x))
           (listp (cddr x)))
      (if (and (listp (caddr x))
               (listp (cdaddr x))
               (listp (cddaddr x))
               (equal (caaddr x) 'neg)
               (equal (cadaddr x) (cadr x))
```

```
              (member (caddaddr x) (add-fringe (cadr x) (cadddr x))))
         (add-tree (cadr x)
                   (delete (caddaddr x) (add-fringe (cadr x) (cadddr x))))
       (if (member (list 'neg (cadr x) (caddr x))
                   (add-fringe (cadr x) (cadddr x)))
           (add-tree (cadr x)
                     (delete (list 'neg (cadr x) (caddr x))
                             (add-fringe (cadr x) (cadddr x))))
         x))
    x))

(prove-lemma correctness-of-cancel-add-neg ((meta add))
     (equal (eval$ t x a)
            (eval$ t (cancel-add-neg x) a)))

(compile-uncompiled-defns "tmp")

;                  CONDITION CODES
; this section considers the various condition codes in Bcc and Scc
; instructions. The condition codes are specified as follows:
;  CC  carry clear       ~C                LS  low or same     C+Z
;  CS  carry set         C                 LT  less than       N*~V+~N*V
;  EQ  equal             Z                 MI  minus           N
;  F   never true        0                 NE  not equal       ~Z
;  GE  greater or equal  N*V+~N*~V         PL  plus            ~N
;  GT  greater than      N*V*~Z+~N*~V*~Z   T   always true     1
;  HI  high              ~C*~Z             VC  overflow clear  ~V
;  LE  less or equal     Z+N*~V+~N*V       VS  overflow set    V

;                           BHI/BLS
; two bridge lemmas.
; the relation between add-c and addx-c.
(prove-lemma add-addx-c (rewrite)
     (equal (add-c n x y) (addx-c n 0 x y))
     ((enable add-adder)))

; the relation between sub-c and subx-c.
(prove-lemma sub-subx-c (rewrite)
     (equal (sub-c n x y) (subx-c n 0 x y))
     ((enable add-adder subtracter sub-adder)
      (disable sub-neg)))

; two lemmas for addx-c and subx-c.
(prove-lemma addx-c-la (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (bitp c)
                   (not (zerop n)))
              (equal (addx-c n c x y)
                     (if (lessp (plus c x y) (exp 2 n)) 0 1)))
     ((enable nat-rangep adder-nat-la)))

(prove-lemma subx-c-la (rewrite)
     (implies (and (nat-rangep x n)
```

```
                (nat-rangep y n)
                (bitp c)
                (not (zerop n)))
           (equal (subx-c n c x y)
                  (if (lessp y (plus c x)) 1 0)))
     ((enable nat-rangep subtracter-nat-la)))

(disable addx-c)
(disable add-c)
(disable subx-c)
(disable sub-c)
(disable mbit-means-lessp)

(prove-lemma sub-z-la (rewrite)
     (implies (and (nat-rangep x n)
                (nat-rangep y n))
           (equal (sub-z n x y)
                  (if (equal (fix x) (fix y)) 1 0))))

(defn between-ileq (x y z)
  (and (ileq x y) (ileq y z)))

(prove-lemma sub-bhi-int (rewrite)
     (implies (and (nat-rangep x n)
                (nat-rangep y n)
                (not (zerop n))
                (not (negativep (nat-to-int x n))))
           (equal (bhi (sub-c n x y) (sub-z n x y))
                  (if (between-ileq 0 (nat-to-int y n) (nat-to-int x n))
                      0 1)))
     ((enable nat-to-int nat-rangep)))

(prove-lemma sub-bhi-0 (rewrite)
     (implies (and (nat-rangep x n)
                (nat-rangep y n)
                (not (lessp (nat-to-uint x) (nat-to-uint y)))
                (not (zerop n)))
           (equal (bhi (sub-c n x y) (sub-z n x y)) 0)))

(prove-lemma sub-bhi-1 (rewrite)
     (implies (and (nat-rangep x n)
                (nat-rangep y n)
                (lessp (nat-to-uint x) (nat-to-uint y))
                (not (zerop n)))
           (equal (bhi (sub-c n x y) (sub-z n x y)) 1)))

(prove-lemma sub-bls (rewrite)
     (implies (and (nat-rangep x n)
                (nat-rangep y n)
                (not (zerop n)))
           (equal (bls (sub-c n x y) (sub-z n x y))
                  (if (lessp (nat-to-uint x) (nat-to-uint y)) 0 1))))

; the trivial relation between bls and bhi.
```

```
(prove-lemma bls-bhi (rewrite)
     (equal (bls c z) (b-not (bhi c z))))

(disable bhi)
(disable bls)

;                          BEQ/BNE
; the z-flag of move.
; the unsigned integer interpretation.
(prove-lemma move-beq-uint (rewrite)
     (implies (nat-rangep x n)
              (equal (beq (move-z n x))
                     (if (equal (nat-to-uint x) 0) 1 0))))

; the signed integer interpretation.
(prove-lemma move-beq-int-0 (rewrite)
     (implies (and (nat-rangep x n)
                   (not (equal (nat-to-int x n) 0)))
              (equal (beq (move-z n x)) 0))
     ((enable nat-to-int)))

(prove-lemma move-beq-int-1 (rewrite)
     (implies (and (nat-rangep x n)
                   (equal (nat-to-int x n) 0))
              (equal (beq (move-z n x)) 1))
     ((enable nat-to-int)))

; zero testing on a sign-extended bit vector is equivalent to
; testing on the original value.
(prove-lemma move-beq-ext (rewrite)
     (implies (and (nat-rangep x n)
                   (leq n size)
                   (not (zerop n)))
              (equal (move-z size (ext n x size)) (move-z n x))))

; the z-flag of sub.
; the unsigned integer interpretation.
(prove-lemma sub-beq-uint (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n))
              (equal (beq (sub-z n x y))
                     (if (equal (nat-to-uint x) (nat-to-uint y)) 1 0))))

; the signed integer interpretation.
(prove-lemma sub-beq-int-0 (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (not (equal (nat-to-int x n) (nat-to-int y n))))
              (equal (beq (sub-z n x y)) 0)))

(prove-lemma sub-beq-int-1 (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (equal (nat-to-int x n) (nat-to-int y n)))
```

```
             (equal (beq (sub-z n x y)) 1)))

; equality testing on two sign-extended bit vectors is equivalent to
; testing on their original values.
(prove-lemma sub-beq-ext (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (leq n size)
                   (not (zerop n)))
              (equal (sub-z size (ext n x size) (ext n y size))
                     (sub-z n x y))))

; the z flag of logor.
(prove-lemma logor-beq-uint (rewrite)
     (equal (beq (or-z n x y))
            (if (and (equal (nat-to-uint x) 0)
                     (equal (nat-to-uint y) 0))
                1 0)))

(prove-lemma logor-beq-int-0 (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (or (not (equal (nat-to-int x n) 0))
                       (not (equal (nat-to-int y n) 0))))
              (equal (beq (or-z n x y)) 0))
     ((enable nat-to-int)))

(prove-lemma logor-beq-int-1 (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (equal (nat-to-int x n) 0)
                   (equal (nat-to-int y n) 0))
              (equal (beq (or-z n x y)) 1))
     ((enable nat-to-int)))

; the z flag of logeor.
(prove-lemma logeor-beq-uint (rewrite)
     (equal (beq (eor-z n x y))
            (if (equal (nat-to-uint x) (nat-to-uint y))
                1 0)))

(prove-lemma logeor-beq-int-0 (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (not (equal (nat-to-int x n) (nat-to-int y n))))
              (equal (beq (eor-z n x y)) 0))
     ((enable nat-to-int)))

(prove-lemma logeor-beq-int-1 (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (equal (nat-to-int x n) (nat-to-int y n)))
              (equal (beq (eor-z n x y)) 1))
     ((enable nat-to-int nat-rangep)))
```

```
; the z flag of logand.
(prove-lemma logand-beq-uint (rewrite)
     (implies (equal (exp 2 (log 2 (add1 x))) (add1 x))
              (equal (beq (and-z n x y))
                     (if (equal (remainder (nat-to-uint y) (add1 x)) 0)
                         1 0)))
     ((use (logand-uint))
      (disable logand-uint)))

(prove-lemma logand-or-beq-uint (rewrite)
     (implies (equal (exp 2 (log 2 (add1 x))) (add1 x))
              (equal (beq (and-z n x (logor y z)))
                     (if (and (equal (remainder (nat-to-uint y) (add1 x)) 0)
                              (equal (remainder (nat-to-uint z) (add1 x)) 0))
                         1 0)))
     ((use (logand-uint) (logand-uint (y z)))
      (disable logand-uint)))

(prove-lemma logand-eor-beq-uint (rewrite)
     (implies (equal (exp 2 (log 2 (add1 x))) (add1 x))
              (equal (beq (and-z n x (logeor y z)))
                     (if (equal (remainder (nat-to-uint y) (add1 x))
                                (remainder (nat-to-uint z) (add1 x)))
                         1 0)))
     ((use (logand-uint) (logand-uint (y z)))
      (disable logand-uint)))

; the z-flag of mulu.  There are three cases to handle.
(prove-lemma mulu_1632-beq (rewrite)
     (implies (and (nat-rangep x 16)
                   (nat-rangep y 16))
              (equal (beq (mulu-z 32 x y 16))
                     (if (or (equal (nat-to-uint x) 0)
                             (equal (nat-to-uint y) 0))
                         1 0)))
     ((use (mulu_1632-nat))))

(prove-lemma mulu_3264-beq (rewrite)
     (implies (and (nat-rangep x 32)
                   (nat-rangep y 32))
              (equal (beq (mulu-z 64 x y 32))
                     (if (or (equal (nat-to-uint x) 0)
                             (equal (nat-to-uint y) 0))
                         1 0)))
     ((use (mulu_3264-nat))))

(prove-lemma mulu_3232-beq (rewrite)
     (implies (uint-rangep (times (nat-to-uint x) (nat-to-uint y)) 32)
              (equal (beq (mulu-z 32 x y 32))
                     (if (or (equal (nat-to-uint x) 0)
                             (equal (nat-to-uint y) 0))
                         1 0)))
     ((enable nat-rangep)))
```

```
; the z-flag of muls.  There are three cases to handle.
(prove-lemma muls_1632-beq (rewrite)
     (implies (and (nat-rangep x 16)
                   (nat-rangep y 16))
              (equal (beq (muls-z 32 x y 16))
                     (if (or (equal (nat-to-int x 16) 0)
                             (equal (nat-to-int y 16) 0))
                         1 0)))
     ((use (muls_1632-int))))

(prove-lemma muls_3264-beq (rewrite)
     (implies (and (nat-rangep x 32)
                   (nat-rangep y 32))
              (equal (beq (muls-z 64 x y 32))
                     (if (or (equal (nat-to-int x 32) 0)
                             (equal (nat-to-int y 32) 0))
                         1 0)))
     ((use (muls_3264-int))))

(prove-lemma muls_3232-beq (rewrite)
     (implies (and (int-rangep (itimes (nat-to-int x 32) (nat-to-int y 32)) 32)
                   (nat-rangep x 32)
                   (nat-rangep y 32))
              (equal (beq (muls-z 32 x y 32))
                     (if (or (equal (nat-to-int x 32) 0)
                             (equal (nat-to-int y 32) 0))
                         1 0)))
     ((use (muls_3232-int))))

; the z-flag of divu.
(prove-lemma divu-beq (rewrite)
     (implies (and (nat-rangep y n)
                   (not (equal (nat-to-uint x) 0)))
              (equal (beq (divu-z n x y))
                     (if (lessp (nat-to-uint y) (nat-to-uint x)) 1 0))))

; the z-flag of divs.
(prove-lemma iquotient=0 (rewrite)
     (implies (and (integerp y)
                   (not (equal y 0)))
              (equal (equal (iquotient x y) 0)
                     (lessp (abs x) (abs y))))
     ((enable integerp iquotient)))

(prove-lemma divs-beq (rewrite)
     (implies (and (int-rangep (iquotient (nat-to-int y j) (nat-to-int x i)) n)
                   (nat-rangep x i)
                   (not (equal (nat-to-int x i) 0))
                   (leq n j))
              (equal (beq (divs-z n x i y j))
                     (if (lessp (abs (nat-to-int y j))
                                (abs (nat-to-int x i)))
                         1 0)))
```

```
     ((enable head-int-crock)))

; the z-flag of lsl.
(prove-lemma lsl-beq (rewrite)
     (implies (uint-rangep (nat-to-uint x) (difference n cnt))
              (equal (beq (lsl-z n x cnt))
                     (if (equal (nat-to-uint x) 0) 1 0)))
     ((enable nat-rangep)))

; the z-flag of lsr.
(prove-lemma lsr-beq (rewrite)
     (equal (beq (lsr-z n x cnt))
            (if (lessp (nat-to-uint x) (exp 2 cnt)) 1 0))
     ((enable nat-rangep)))

(prove-lemma z-flag-la ()
     (implies (numberp x)
              (equal (equal (nat-to-int x n) 0)
                     (equal x 0)))
     ((enable nat-to-int)))

; the z-flag of asl.
(prove-lemma asl-beq (rewrite)
     (implies (and (nat-rangep x n)
                   (int-rangep (nat-to-int x n) (difference n cnt)))
              (equal (beq (asl-z n x cnt))
                     (if (equal (nat-to-int x n) 0) 1 0)))
     ((use (z-flag-la (x (asl n x cnt))))
      (disable asl)))

; the z-flag of asr.
(prove-lemma asr-beq (rewrite)
     (implies (nat-rangep x n)
              (equal (beq (asr-z n x cnt))
                     (if (negativep (nat-to-int x n))
                         0
                       (if (lessp (nat-to-int x n) (exp 2 cnt))
                           1 0))))
     ((use (z-flag-la (x (asr n x cnt))))
      (enable iplus iquotient iremainder)
      (disable asr)))

; the z-flag of ext.
(prove-lemma ext-beq-uint (rewrite)
     (implies (and (nat-rangep x n)
                   (leq n size))
              (equal (beq (ext-z n x size))
                     (if (equal (nat-to-uint x) 0) 1 0)))
     ((enable app ext nat-rangep mbit-means-lessp)))

(prove-lemma ext-beq-int-0 (rewrite)
     (implies (and (nat-rangep x n)
                   (lessp n size)
                   (not (equal (nat-to-int x n) 0)))
```

```
              (equal (beq (ext-z n x size)) 0))
     ((use (z-flag-la (x (ext n x size)) (n size)))))

(prove-lemma ext-beq-int-1 (rewrite)
     (implies (and (nat-rangep x n)
                   (lessp n size)
                   (equal (nat-to-int x n) 0))
              (equal (beq (ext-z n x size)) 1))
     ((use (z-flag-la (x (ext n x size)) (n size)))))

(disable iquotient=0)
(disable int-to-nat=0)
(disable beq)

;                          BCS/BCC
; BCS/BCC means carry is set/cleared.
; the c-flag of sub.
(prove-lemma sub-bcs&cc (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (not (zerop n)))
              (equal (bcs (sub-c n x y))
                     (if (lessp (nat-to-uint y) (nat-to-uint x)) 1 0))))

; the c-flag of add.
(prove-lemma add-bcs&cc (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (not (zerop n)))
              (equal (bcs (add-c n x y))
                     (if (lessp (plus (nat-to-uint x) (nat-to-uint y))
                                (exp 2 n))
                         0 1))))

(prove-lemma tail-mbit (rewrite)
     (implies (lessp x (exp 2 n))
              (equal (tail x (sub1 n))
                     (if (lessp x (exp 2 (sub1 n))) 0 1)))
     ((enable tail)))

; the c-flag of lsl.  But we only consider special cases: cnt = 0 or 1.
; I don't know what the meaning of c-flag is when cnt > 1.
(prove-lemma lsl-c-0 (rewrite)
     (equal (lsl-c n x 0) 0))

(prove-lemma lsl-1-bcs&cc (rewrite)
     (implies (and (nat-rangep x n)
                   (lessp 1 n))
              (equal (bcs (lsl-c n x 1))
                     (if (uint-rangep (nat-to-uint x) (sub1 n)) 0 1)))
     ((enable bitn nat-rangep)))

; the c-flag of lsr.  But we only consider special cases: cnt = 0 or 1.
; I don't know what the meaning of c-flag is when cnt > 1.
```

```
(prove-lemma lsr-c-0 (rewrite)
     (equal (lsr-c n x 0) 0))

(prove-lemma lsr-1-bcs&cc (rewrite)
     (implies (lessp 1 n)
              (equal (bcs (lsr-c n x 1))
                     (remainder (nat-to-uint x) 2)))
     ((enable bitn bcar)))

(disable tail-mbit)
(disable bcs)

;                       BVS/BVC
; BVS/BVC means overflow is set/cleared.
; three bridge lemmas.
(prove-lemma add-addx-v (rewrite)
     (equal (add-v n x y) (addx-v n 0 x y))
     ((enable add-adder)))

(prove-lemma subx-addx-v (rewrite)
     (implies (and (nat-rangep x n)
                   (not (zerop n)))
              (equal (subx-v n z x y)
                     (addx-v n (b-not z) y (lognot n x))))
     ((enable subtracter)))

(prove-lemma sub-subx-v (rewrite)
     (equal (sub-v n x y) (subx-v n 0 x y))
     ((enable add-adder subtracter sub-adder)
      (disable sub-neg)))

; lemmas for addx-v and subx-v.
(prove-lemma mbit-means-negativep (rewrite)
     (implies (nat-rangep x n)
              (equal (bitn x (sub1 n))
                     (if (negativep (nat-to-int x n)) 1 0)))
     ((enable nat-to-int bitn bcar tail nat-rangep)))

(prove-lemma addx-v-la (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (bitp c)
                   (not (zerop n)))
              (equal (addx-v n c x y)
                     (if (int-rangep (iplus (nat-to-int x n)
                                            (iplus (nat-to-int y n) c))
                                     n)
                         0 1)))
     ((disable bitp iplus-commutativity iplus-commutativity1)))

(prove-lemma subx-v-la (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (bitp c)
```

```
                (not (zerop n)))
           (equal (subx-v n c x y)
                  (if (int-rangep (idifference (nat-to-int y n)
                                               (iplus (nat-to-int x n) c))
                                  n)
                      0 1))))

(disable addx-v-crock1)
(disable addx-v-crock2)

; the v-flag of sub.
(prove-lemma sub-bvs&vc (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (not (zerop n)))
              (equal (bvs (sub-v n x y))
                     (if (int-rangep (idifference (nat-to-int y n)
                                                  (nat-to-int x n))
                                     n)
                         0 1))))

; the v-flag of add.
(prove-lemma add-bvs&vc (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (not (zerop n)))
              (equal (bvs (add-v n x y))
                     (if (int-rangep (iplus (nat-to-int x n) (nat-to-int y n))
                                     n)
                         0 1))))

; three cases for the v-flag of mulu.
(prove-lemma mulu_1632-bvs (rewrite)
     (implies (and (nat-rangep x 16)
                   (nat-rangep y 16))
              (equal (mulu-v 32 x y 16) 0))
     ((use (times-lessp_1 (x1 (exp 2 16)) (y1 (exp 2 16))))
      (enable nat-rangep)))

(prove-lemma mulu_3264-bvs (rewrite)
     (implies (and (nat-rangep x 32)
                   (nat-rangep y 32))
              (equal (mulu-v 64 x y 32) 0))
     ((use (times-lessp_1 (x1 (exp 2 32)) (y1 (exp 2 32))))
      (enable nat-rangep)))

(prove-lemma mulu_3232-bvs (rewrite)
     (equal (bvs (mulu-v 32 x y 32))
            (if (lessp (times (nat-to-uint x) (nat-to-uint y)) (exp 2 32))
                0 1)))

; three cases for the v-flag of muls.
(prove-lemma muls_1632-bvs (rewrite)
     (implies (and (nat-rangep x 16)
```

```
                (nat-rangep y 16))
           (equal (muls-v 32 x y 16) 0))
     ((use (muls-crock (x (nat-to-int x 16)) (y (nat-to-int y 16)) (n 16)))))

(prove-lemma muls_3264-bvs (rewrite)
     (implies (and (nat-rangep x 32)
                (nat-rangep y 32))
           (equal (muls-v 64 x y 32) 0))
     ((use (muls-crock (x (nat-to-int x 32)) (y (nat-to-int y 32)) (n 32)))))

(prove-lemma muls_3232-bvs (rewrite)
     (equal (bvs (muls-v 32 x y 32))
            (if (int-rangep (itimes (nat-to-int x 32) (nat-to-int y 32)) 32)
                0 1)))

; the v-flag of asl.  It is an easy copy of the definition of asl-v.
; sometimes, it seems to be better to define those functions directly
; using the intended meaning.  But there are several problems:
; 1. the meaning is not completely clear. It is vague.
; 2. it is impossible to have a clean definition.
; 3. it is error prone. It does not always work the way you think.
;    Your intended interpretation is just a special case.
; In some sense,  a formal specification should be more syntax-oriented.
; syntax rules are clear and accurate. e.g. M68000 are well-documented.
; But it gives one a hard time to do formal proofs.  We need to assign
; meanings to these definitions.  To make sure our "intended" meanings
; are consistent with the specification, we need to prove the equivalence.
; this is done formally by a theorem prover, and sometimes, is very hard.
(prove-lemma asl-bvs (rewrite)
     (equal (bvs (asl-v n x cnt))
            (if (int-rangep (nat-to-int x n) (difference n cnt))
                0 1)))

(disable bvs)

;                      BMI/BPL
; the n-flag of move.
(prove-lemma move-bmi (rewrite)
     (implies (nat-rangep x n)
              (equal (bmi (move-n n x))
                     (if (negativep (nat-to-int x n))
                         1 0))))

; the n-flag of sub.
(prove-lemma sub-bmi (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (not (zerop n)))
              (equal (bmi (sub-n n x y))
                     (if (int-rangep (idifference (nat-to-int y n)
                                                  (nat-to-int x n))
                                     n)
                         (if (ilessp (nat-to-int y n) (nat-to-int x n)) 1 0)
                       (if (ilessp (nat-to-int y n) (nat-to-int x n)) 0 1)))))
```

```
     ((disable idifference ilessp iplus-commutativity)))

; the n-flag of add.
(prove-lemma add-bmi (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (not (zerop n)))
              (equal (bmi (add-n n x y))
                     (if (int-rangep (iplus (nat-to-int x n) (nat-to-int y n))
                                     n)
                         (if (negativep (iplus (nat-to-int x n)
                                               (nat-to-int y n)))
                             1 0)
                       (if (negativep (nat-to-int x n)) 0 1))))
     ((disable iplus-commutativity)))

; the n-flag of muls.  There are three cases to handle.
(prove-lemma muls_1632-bmi (rewrite)
     (implies (and (nat-rangep x 16)
                   (nat-rangep y 16))
              (equal (bmi (muls-n 32 x y 16))
                     (if (or (equal (nat-to-int x 16) 0)
                             (equal (nat-to-int y 16) 0))
                         0
                       (if (negativep (nat-to-int x 16))
                           (if (negativep (nat-to-int y 16))
                               0 1)
                         (if (negativep (nat-to-int y 16))
                             1 0))))))
     ((use (mbit-means-negativep (x (muls 32 x y 16)) (n 32)))
      (disable muls)))

(prove-lemma muls_3264-bmi (rewrite)
     (implies (and (nat-rangep x 32)
                   (nat-rangep y 32))
              (equal (bmi (muls-n 64 x y 32))
                     (if (or (equal (nat-to-int x 32) 0)
                             (equal (nat-to-int y 32) 0))
                         0
                       (if (negativep (nat-to-int x 32))
                           (if (negativep (nat-to-int y 32)) 0 1)
                         (if (negativep (nat-to-int y 32)) 1 0))))))
     ((use (mbit-means-negativep (x (muls 64 x y 32)) (n 64)))
      (disable muls)))

(prove-lemma muls_3232-bmi (rewrite)
     (implies (and (int-rangep (itimes (nat-to-int x 32) (nat-to-int y 32)) 32)
                   (nat-rangep x 32)
                   (nat-rangep y 32))
              (equal (bmi (muls-n 32 x y 32))
                     (if (or (equal (nat-to-int x 32) 0)
                             (equal (nat-to-int y 32) 0))
                         0
                       (if (negativep (nat-to-int x 32))
```

```
                          (if (negativep (nat-to-int y 32))
                              0 1)
                         (if (negativep (nat-to-int y 32))
                             1 0)))))
     ((use (mbit-means-negativep (x (muls 32 x y 32)) (n 32)))
      (disable muls)))

; the n-flag of asl.
(prove-lemma asl-bmi (rewrite)
     (implies (and (nat-rangep x n)
                   (int-rangep (nat-to-int x n) (difference n cnt)))
              (equal (bmi (asl-n n x cnt))
                     (if (negativep (nat-to-int x n)) 1 0)))
     ((disable asl)))

; the n-flag of asr.
(prove-lemma asr-bmi (rewrite)
     (implies (nat-rangep x n)
              (equal (bmi (asr-n n x cnt))
                     (if (negativep (nat-to-int x n))
                         1 0)))
     ((enable iplus iquotient iremainder)
      (disable asr)))

; the n-flag of ext.
(prove-lemma ext-bmi (rewrite)
     (implies (and (nat-rangep x n)
                   (lessp n size))
              (equal (bmi (ext-n n x size))
                     (if (negativep (nat-to-int x n)) 1 0))))

;                    BGE/BLT
(prove-lemma bge-v0 (rewrite)
     (equal (bge 0 n) (b-not (bmi n))))

(prove-lemma blt-v0 (rewrite)
     (equal (blt 0 n) (bmi n)))

; bge of sub.
(prove-lemma sub-bge (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (not (zerop n)))
              (equal (bge (sub-v n x y) (sub-n n x y))
                     (if (ilessp (nat-to-int y n) (nat-to-int x n))
                         0 1)))
     ((disable ilessp idifference iplus-commutativity)))

; blt of sub.
(prove-lemma sub-blt (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (not (zerop n)))
              (equal (blt (sub-v n x y) (sub-n n x y))
```

```
                    (if (ilessp (nat-to-int y n) (nat-to-int x n))
                        1 0)))
     ((disable ilessp idifference iplus-commutativity)))

; the trivial relation between BGE and BLT.
(prove-lemma blt-bge (rewrite)
     (equal (blt v n) (b-not (bge v n))))

(disable bge)
(disable blt)

;                       BGT/BLE
(prove-lemma sub-z-la1 (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n))
              (equal (sub-z n x y)
                     (if (equal (nat-to-int x n) (nat-to-int y n))
                         1 0))))

(disable sub-z)

; bgt of sub.
(prove-lemma sub-bgt (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (not (zerop n)))
              (equal (bgt (sub-v n x y) (sub-z n x y) (sub-n n x y))
                     (if (ilessp (nat-to-int x n) (nat-to-int y n))
                         1 0)))
     ((disable idifference iplus-commutativity nat-to-int=)))

; ble of sub.
(prove-lemma sub-ble (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (not (zerop n)))
              (equal (ble (sub-v n x y) (sub-z n x y) (sub-n n x y))
                     (if (ilessp (nat-to-int x n) (nat-to-int y n))
                         0 1)))
     ((disable idifference iplus-commutativity nat-to-int=)))

; bgt of move.
(prove-lemma move-bgt (rewrite)
     (implies (nat-rangep x n)
              (equal (bgt 0 (move-z n x) (move-n n x))
                     (if (lessp 0 (nat-to-int x n)) 1 0)))
     ((enable nat-to-int)))

; ble of move.
(prove-lemma move-ble (rewrite)
     (implies (nat-rangep x n)
              (equal (ble 0 (move-z n x) (move-n n x))
                     (if (lessp 0 (nat-to-int x n)) 0 1)))
     ((enable nat-to-int)))
```

```
; the trivial relation between ble and bgt.
(prove-lemma ble-bgt (rewrite)
     (equal (ble v z n) (b-not (bgt v z n))))

(disable ble)
(disable bgt)

; the end of this subsection.  Some lemmas should be disabled, for we
; no longer need them.
(disable mbit-means-negativep)
(disable bmi)
(disable add-addx-c)
(disable sub-subx-c)
(disable addx-c-la)
(disable subx-c-la)
(disable add-addx-v)
(disable subx-addx-v)
(disable sub-subx-v)
(disable addx-v-la)
(disable subx-v-la)
(disable sub-z-la)
(disable sub-z-la1)
(disable ilessp-crock1)
(disable ilessp-crock2)
(disable add-bmi-crock1)
(disable add-bmi-crock2)
(disable nat-to-int=)

;                           STEPN
; stepping the state s (m+n) steps is equivalent to stepping s m steps and
; n steps further.
(defn splus (x y) (plus x y))

(prove-lemma stepn-lemma (rewrite)
     (equal (stepn s (splus m n))
            (stepn (stepn s m) n))
     ((disable stepi)))

(prove-lemma stepn-rewriter0 (rewrite)
     (equal (stepn s 0) s))

(prove-lemma stepn-rewriter (rewrite)
     (implies (equal (mc-status s) 'running)
              (equal (stepn s (add1 n))
                     (stepn (stepi s) n)))
     ((disable stepi)))

;                     THEOREMS ABOUT THE MEMORY
; if a portion of memory is ROM, then it still is ROM after a write on memory.
(prove-lemma byte-write-maintain-rom-addrp (rewrite)
     (equal (rom-addrp addr (byte-write value x mem) n)
            (rom-addrp addr mem n)))
```

```
(prove-lemma write-mem-maintain-rom-addrp (rewrite)
     (equal (rom-addrp addr (write-mem value x mem m) n)
            (rom-addrp addr mem n)))

; if a portion of memory is RAM, then it still is RAM after a write on memory.
(prove-lemma byte-write-maintain-ram-addrp (rewrite)
     (equal (ram-addrp addr (byte-write value x mem) n)
            (ram-addrp addr mem n)))

(prove-lemma write-mem-maintain-ram-addrp (rewrite)
     (equal (ram-addrp addr (write-mem value x mem m) n)
            (ram-addrp addr mem n)))

; mcode-addrp claims that the machine code program lst is stored in the
; memory starting at addr.
(defn mcode-addrp (addr mem lst)
  (if (listp lst)
      (if (equal (car lst) -1)
          (mcode-addrp (add 32 addr 1) mem (cdr lst))
        (and (equal (pc-byte-read addr mem) (car lst))
             (mcode-addrp (add 32 addr 1) mem (cdr lst))))
    t))

(prove-lemma add-non-numberp (rewrite)
     (implies (not (numberp i))
              (equal (add n x i) (head x n)))
     ((enable add head)))

(prove-lemma add-plus (rewrite)
     (equal (add n x (add n y z))
            (add n x (plus y z)))
     ((enable add)))

; four lemmas about pc-read-memp.
(prove-lemma pc-read-memp-la0 (rewrite)
     (implies (and (pc-read-memp addr mem k)
                   (not (zerop k)))
              (pc-byte-readp addr mem)))

(prove-lemma pc-read-memp-la1 (rewrite)
     (implies (and (pc-read-memp addr mem k)
                   (lessp j k))
              (pc-byte-readp (add 32 addr j) mem)))

(prove-lemma pc-read-memp-la2 (rewrite)
     (implies (and (pc-read-memp addr mem k)
                   (leq j k))
              (pc-read-memp addr mem j))
     ((use (pc-read-memp-la1 (j (sub1 j))))))

(prove-lemma pc-read-memp-la3 (rewrite)
     (implies (and (pc-read-memp addr mem k)
                   (leq (plus i j) k))
              (pc-read-memp (add 32 addr i) mem j))
```

```
     ((induct (pc-read-memp addr mem j))))

(prove-lemma write-memp-la0 (rewrite)
     (implies (and (write-memp addr mem n)
                   (not (zerop n)))
              (byte-writep addr mem)))

(prove-lemma write-memp-la1 (rewrite)
     (implies (and (write-memp addr mem n)
                   (lessp m n))
              (byte-writep (add 32 addr m) mem)))

(prove-lemma write-memp-la2 (rewrite)
     (implies (and (write-memp addr mem n)
                   (leq m n))
              (write-memp addr mem m))
     ((use (write-memp-la1 (m (sub1 m))))))

(prove-lemma write-memp-la3 (rewrite)
     (implies (and (write-memp addr mem n)
                   (leq (plus i j) n))
              (write-memp (add 32 addr i) mem j))
     ((induct (write-memp addr mem j))))

(disable add-plus)

; array index.  There are some good reasons to distinguish this concept
; with add and sub.
(defn index-n (x y)
  (sub 32 y x))

; ROM is pc-readable.
(prove-lemma pc-byte-readp-rom0 (rewrite)
     (implies (and (rom-addrp addr mem k)
                   (not (zerop k)))
              (pc-byte-readp addr mem)))

(prove-lemma pc-byte-readp-rom1 (rewrite)
     (implies (and (rom-addrp addr mem k)
                   (lessp j k))
              (pc-byte-readp (add 32 addr j) mem)))

(prove-lemma pc-byte-readp-rom2 (rewrite)
     (implies (and (rom-addrp (add 32 addr i) mem k)
                   (lessp (index-n 0 i) k))
              (pc-byte-readp addr mem)))

(prove-lemma pc-byte-readp-rom3 (rewrite)
     (implies (and (rom-addrp (add 32 addr i) mem k)
                   (lessp (index-n j i) k))
              (pc-byte-readp (add 32 addr j) mem))
     ((use (pc-byte-readp-rom1 (addr (add 32 addr i)) (j (index-n j i))))))

(prove-lemma pc-read-memp-rom0 (rewrite)
```

```
     (implies (and (rom-addrp addr mem k)
                   (leq j k))
              (pc-read-memp addr mem j)))

(prove-lemma pc-read-memp-rom1 (rewrite)
     (implies (and (rom-addrp addr mem k)
                   (leq (plus i j) k))
              (pc-read-memp (add 32 addr i) mem j)))

(prove-lemma pc-read-memp-rom2 (rewrite)
     (implies (and (rom-addrp (add 32 addr i) mem k)
                   (leq (plus (index-n 0 i) j) k))
              (pc-read-memp addr mem j))
     ((use (pc-read-memp-rom1 (addr (add 32 addr i)) (i (index-n 0 i))))))

(prove-lemma pc-read-memp-rom3 (rewrite)
     (implies (and (rom-addrp (add 32 addr i) mem k)
                   (leq (plus (index-n h i) j) k))
              (pc-read-memp (add 32 addr h) mem j))
     ((use (pc-read-memp-rom1 (addr (add 32 addr i)) (i (index-n h i))))))

; starting at the same address, a portion of the memory is ROM if a bigger
; portion of the memory is ROM.
(prove-lemma rom-addrp-la1 (rewrite)
     (implies (and (rom-addrp addr mem n)
                   (leq m n))
              (rom-addrp addr mem m)))

; a portion of the memory is also ROM, if a bigger portion of the memory
; is ROM.
(prove-lemma rom-addrp-la2 (rewrite)
     (implies (and (rom-addrp addr mem k)
                   (leq (plus j (index-n x addr)) k))
              (rom-addrp x mem j))
     ((use (pc-read-memp-la3 (i (index-n x addr))))))

(disable rom-addrp)

; ROM is readable.
(prove-lemma byte-readp-rom0 (rewrite)
     (implies (and (rom-addrp addr mem k)
                   (not (zerop k)))
              (byte-readp addr mem)))

(prove-lemma byte-readp-rom1 (rewrite)
     (implies (and (rom-addrp addr mem k)
                   (lessp j k))
              (byte-readp (add 32 addr j) mem)))

(prove-lemma byte-readp-rom2 (rewrite)
     (implies (and (rom-addrp (add 32 addr i) mem k)
                   (lessp (index-n 0 i) k))
              (byte-readp addr mem)))
```

```
(prove-lemma byte-readp-rom3 (rewrite)
     (implies (and (rom-addrp (add 32 addr i) mem k)
                   (lessp (index-n j i) k))
              (byte-readp (add 32 addr j) mem)))

(prove-lemma read-memp-rom0 (rewrite)
     (implies (and (rom-addrp addr mem k)
                   (leq j k))
              (read-memp addr mem j)))

(prove-lemma read-memp-rom1 (rewrite)
     (implies (and (rom-addrp addr mem k)
                   (leq (plus i j) k))
              (read-memp (add 32 addr i) mem j)))

(prove-lemma read-memp-rom2 (rewrite)
     (implies (and (rom-addrp (add 32 addr i) mem k)
                   (leq (plus (index-n 0 i) j) k))
              (read-memp addr mem j)))

(prove-lemma read-memp-rom3 (rewrite)
     (implies (and (rom-addrp (add 32 addr i) mem k)
                   (leq (plus (index-n h i) j) k))
              (read-memp (add 32 addr h) mem j)))

; integer view.
(prove-lemma read-memp-rom1-int (rewrite)
     (implies (and (rom-addrp addr mem n)
                   (lessp (plus (nat-to-int x 32) j) n)
                   (numberp (nat-to-int x 32)))
              (read-memp (add 32 addr x) mem j))
     ((enable nat-to-int)))

; RAM is writable.
(prove-lemma byte-writep-ram0 (rewrite)
     (implies (and (ram-addrp addr mem k)
                   (not (zerop k)))
              (byte-writep addr mem)))

(prove-lemma byte-writep-ram1 (rewrite)
     (implies (and (ram-addrp addr mem k)
                   (lessp j k))
              (byte-writep (add 32 addr j) mem)))

(prove-lemma byte-writep-ram2 (rewrite)
     (implies (and (ram-addrp (add 32 addr i) mem k)
                   (lessp (index-n 0 i) k))
              (byte-writep addr mem)))

(prove-lemma byte-writep-ram3 (rewrite)
     (implies (and (ram-addrp (add 32 addr i) mem k)
                   (lessp (index-n j i) k))
              (byte-writep (add 32 addr j) mem))
     ((use (byte-writep-ram1 (addr (add 32 addr i)) (j (index-n j i))))))
```

```
(prove-lemma write-memp-ram0 (rewrite)
     (implies (and (ram-addrp addr mem k)
                   (leq j k))
              (write-memp addr mem j)))

(prove-lemma write-memp-ram1 (rewrite)
     (implies (and (ram-addrp addr mem k)
                   (leq (plus i j) k))
              (write-memp (add 32 addr i) mem j)))

(prove-lemma write-memp-ram1-int (rewrite)
     (implies (and (ram-addrp addr mem k)
                   (numberp (nat-to-int i 32))
                   (leq (plus (nat-to-int i 32) j) k))
              (write-memp (add 32 addr i) mem j))
     ((enable nat-to-int)))

(prove-lemma write-memp-ram2 (rewrite)
     (implies (and (ram-addrp (add 32 addr i) mem k)
                   (leq (plus (index-n 0 i) j) k))
              (write-memp addr mem j))
     ((use (write-memp-ram1 (addr (add 32 addr i)) (i (index-n 0 i))))))

(prove-lemma write-memp-ram3 (rewrite)
     (implies (and (ram-addrp (add 32 addr i) mem k)
                   (leq (plus (index-n h i) j) k))
              (write-memp (add 32 addr h) mem j))
     ((use (write-memp-ram1 (addr (add 32 addr i)) (i (index-n h i))))))

; a portion of a memory is also RAM, if a "bigger" portion of the memory
; is RAM.
(prove-lemma ram-addrp-la1 (rewrite)
     (implies (and (ram-addrp addr mem n)
                   (leq m n))
              (ram-addrp addr mem m)))

(prove-lemma ram-addrp-la2 (rewrite)
     (implies (and (ram-addrp addr mem n)
                   (leq (plus j (index-n x addr)) n))
              (ram-addrp x mem j))
     ((use (write-memp-la3 (i (index-n x addr))))))

(disable ram-addrp)

; RAM is readable.
(prove-lemma byte-readp-ram0 (rewrite)
     (implies (and (ram-addrp addr mem k)
                   (not (zerop k)))
              (byte-readp addr mem)))

(prove-lemma byte-readp-ram1 (rewrite)
     (implies (and (ram-addrp addr mem k)
                   (lessp j k))
```

```
              (byte-readp (add 32 addr j) mem)))

(prove-lemma byte-readp-ram2 (rewrite)
     (implies (and (ram-addrp (add 32 addr i) mem k)
                   (lessp (index-n 0 i) k))
              (byte-readp addr mem)))

(prove-lemma byte-readp-ram3 (rewrite)
     (implies (and (ram-addrp (add 32 addr i) mem k)
                   (lessp (index-n j i) k))
              (byte-readp (add 32 addr j) mem)))

(prove-lemma read-memp-ram0 (rewrite)
     (implies (and (ram-addrp addr mem k)
                   (leq j k))
              (read-memp addr mem j)))

(prove-lemma read-memp-ram1 (rewrite)
     (implies (and (ram-addrp addr mem k)
                   (leq (plus i j) k))
              (read-memp (add 32 addr i) mem j)))

(prove-lemma read-memp-ram1-int (rewrite)
     (implies (and (ram-addrp addr mem k)
                   (numberp (nat-to-int i 32))
                   (leq (plus (nat-to-int i 32) j) k))
              (read-memp (add 32 addr i) mem j))
     ((enable nat-to-int)))

(prove-lemma read-memp-ram2 (rewrite)
     (implies (and (ram-addrp (add 32 addr i) mem k)
                   (leq (plus (index-n 0 i) j) k))
              (read-memp addr mem j)))

(prove-lemma read-memp-ram3 (rewrite)
     (implies (and (ram-addrp (add 32 addr i) mem k)
                   (leq (plus (index-n h i) j) k))
              (read-memp (add 32 addr h) mem j)))

; please use the right induction.
(defn mem-induct1 (i addr lst)
  (if (zerop i)
      t
    (mem-induct1 (sub1 i) (add 32 addr 1) (cdr lst))))

;            OBTAIN MACHINE CODE FROM MEMORY
; please read the right thing.
(prove-lemma pc-byte-read-mcode0 (rewrite)
     (implies (and (mcode-addrp addr mem (cons x lst))
                   (not (equal x -1)))
              (equal (pc-byte-read addr mem) x)))

(prove-lemma pc-byte-read-mcode1 (rewrite)
     (implies (and (mcode-addrp addr mem lst)
```

```
                (lessp i (len lst))
                (not (equal (get-nth i lst) -1)))
           (equal (pc-byte-read (add 32 addr i) mem)
                  (get-nth i lst)))
     ((induct (mem-induct1 i addr lst))
      (enable add-plus)))

(prove-lemma pc-byte-read-mcode2 (rewrite)
     (implies (and (mcode-addrp (add 32 addr i) mem lst)
                (lessp (index-n 0 i) (len lst))
                (not (equal (get-nth (index-n 0 i) lst) -1)))
           (equal (pc-byte-read addr mem)
                  (get-nth (index-n 0 i) lst)))
     ((use (pc-byte-read-mcode1 (addr (add 32 addr i)) (i (index-n 0 i))))))

(prove-lemma pc-byte-read-mcode3 (rewrite)
     (implies (and (mcode-addrp (add 32 addr i) mem lst)
                (lessp (index-n j i) (len lst))
                (not (equal (get-nth (index-n j i) lst) -1)))
           (equal (pc-byte-read (add 32 addr j) mem)
                  (get-nth (index-n j i) lst)))
     ((use (pc-byte-read-mcode1 (addr (add 32 addr i)) (i (index-n j i))))))

(defn lst-numberp0 (lst n)
  (if (zerop n)
      0
    (and (numberp (get-nth (sub1 n) lst))
         (lst-numberp0 lst (sub1 n)))))

(defn read-lst0 (lst n)
  (if (zerop n)
      0
    (app (b)
         (get-nth (sub1 n) lst)
         (read-lst0 lst (sub1 n)))))

(defn tail-lst (lst n)
  (if (zerop n)
      lst
    (tail-lst (cdr lst) (sub1 n))))

(defn lst-numberp (m lst n)
  (lst-numberp0 (tail-lst lst m) n))

(defn read-lst (m lst n)
  (read-lst0 (tail-lst lst m) n))

(prove-lemma get-nth-tail-lst (rewrite)
     (equal (get-nth n (tail-lst x m))
            (get-nth (plus m n) x)))

(prove-lemma pc-read-mem-mcode0 (rewrite)
     (implies (and (mcode-addrp addr mem lst)
                (leq j (len lst))
```

```
                (lst-numberp 0 lst j))
           (equal (pc-read-mem addr mem j)
                  (read-lst 0 lst j))))

(prove-lemma pc-read-mem-mcode1 (rewrite)
     (implies (and (mcode-addrp addr mem lst)
                   (leq (plus i j) (len lst))
                   (lst-numberp i lst j))
              (equal (pc-read-mem (add 32 addr i) mem j)
                     (read-lst i lst j)))
     ((enable add-plus)))

(disable read-lst)
(disable lst-numberp)

(prove-lemma pc-read-mem-mcode2 (rewrite)
     (implies (and (mcode-addrp (add 32 addr i) mem lst)
                   (leq (plus (index-n 0 i) k) (len lst))
                   (lst-numberp (index-n 0 i) lst k))
              (equal (pc-read-mem addr mem k)
                     (read-lst (index-n 0 i) lst k)))
     ((use (pc-read-mem-mcode1 (addr (add 32 addr i))
                               (i (index-n 0 i)) (j k)))))

(prove-lemma pc-read-mem-mcode3 (rewrite)
     (implies (and (mcode-addrp (add 32 addr i) mem lst)
                   (leq (plus (index-n j i) k) (len lst))
                   (lst-numberp (index-n j i) lst k))
              (equal (pc-read-mem (add 32 addr j) mem k)
                     (read-lst (index-n j i) lst k)))
     ((use (pc-read-mem-mcode1 (addr (add 32 addr i))
                               (i (index-n j i)) (j k)))))

; sometimes,  we obtain machine code by read-mem.
(prove-lemma read->pc-read-mem (rewrite)
     (equal (read-mem x mem k)
            (pc-read-mem x mem k))
     ((enable read-mem pc-read-mem pc-byte-read byte-read pc-read)))

(prove-lemma read-mem-mcode1-int (rewrite)
     (implies (and (mcode-addrp addr mem lst)
                   (leq (plus (nat-to-int i 32) j) (len lst))
                   (numberp (nat-to-int i 32))
                   (lst-numberp (nat-to-int i 32) lst j))
              (equal (read-mem (add 32 addr i) mem j)
                     (read-lst (nat-to-int i 32) lst j)))
     ((enable nat-to-int)))

(prove-lemma read-mem-mcode2 (rewrite)
     (implies (and (mcode-addrp (add 32 addr i) mem lst)
                   (leq (plus (index-n 0 i) k) (len lst))
                   (lst-numberp (index-n 0 i) lst k))
              (equal (read-mem addr mem k)
                     (read-lst (index-n 0 i) lst k))))
```

```
(prove-lemma read-mem-mcode3 (rewrite)
     (implies (and (mcode-addrp (add 32 addr i) mem lst)
                   (leq (plus (index-n j i) k) (len lst))
                   (lst-numberp (index-n j i) lst k))
              (equal (read-mem (add 32 addr j) mem k)
                     (read-lst (index-n j i) lst k))))

(disable read->pc-read-mem)

; program segment remains unchanged after any write on memory,  because
; program is in ROM.
(prove-lemma byte-write-mcode-addrp (rewrite)
     (implies (and (pc-read-memp pc mem (len lst))
                   (byte-writep x mem))
              (equal (mcode-addrp pc (byte-write value x mem) lst)
                     (mcode-addrp pc mem lst)))
     ((induct (mcode-addrp pc mem lst))
      (use (pc-byte-read-write (y pc)))
      (disable add-commutativity)))

(prove-lemma write-mem-mcode-addrp (rewrite)
     (implies (and (pc-read-memp pc mem (len lst))
                   (write-memp x mem n))
              (equal (mcode-addrp pc (write-mem value x mem n) lst)
                     (mcode-addrp pc mem lst)))
     ((induct (mcode-addrp pc mem lst))
      (use (pc-byte-read-write-mem (y pc)))
      (disable add-commutativity)))

;               DISJOINT0 AND DISJOINT

(prove-lemma disjoint0-head (rewrite)
     (and (equal (disjoint0 (head x 32) m y)
                 (disjoint0 x m y))
          (equal (disjoint0 x m (head y 32))
                 (disjoint0 x m y))))

(prove-lemma disjoint-head (rewrite)
     (and (equal (disjoint (head x 32) m y n)
                 (disjoint x m y n))
          (equal (disjoint x m (head y 32) n)
                 (disjoint x m y n))))

(prove-lemma disjoint0-la0 (rewrite)
     (implies (and (disjoint0 a m b)
                   (lessp i m))
              (not (mod32-eq (add 32 a i) b))))

(prove-lemma disjoint0-la1 (rewrite)
     (implies (and (disjoint0 a m b)
                   (lessp i m))
              (disjoint0 a i b)))
```

```
(prove-lemma disjoint0-la2 (rewrite)
     (implies (and (disjoint0 a m b)
                   (leq (plus i j) m))
              (disjoint0 (add 32 a i) j b))
     ((induct (disjoint0 a j b))
      (enable add-plus)))

(prove-lemma disjoint-la0 (rewrite)
     (implies (and (disjoint a m b n)
                   (lessp j n))
              (disjoint0 a m (add 32 b j))))

(prove-lemma disjoint-la1 (rewrite)
     (implies (and (disjoint a m b n)
                   (lessp i m)
                   (lessp j n))
              (not (mod32-eq (add 32 a i) (add 32 b j))))
     ((use (disjoint0-la0 (b (add 32 b j))))))

(prove-lemma disjoint-la2 (rewrite)
     (implies (and (disjoint a m b n)
                   (leq (plus i j) m)
                   (lessp k n))
              (disjoint0 (add 32 a i) j (add 32 b k)))
     ((induct (disjoint0 a j b))
      (enable add-plus)))

(prove-lemma disjoint-la3 (rewrite)
     (implies (and (disjoint a m b n)
                   (leq (plus i j) m)
                   (leq (plus k l) n))
              (disjoint (add 32 a i) j (add 32 b k) l))
     ((induct (disjoint x j y l))
      (enable add-plus)))

;
(prove-lemma disjoint-0 (rewrite)
     (implies (and (disjoint a m b n)
                   (leq j m)
                   (leq l n))
              (disjoint a j b l))
     ((use (disjoint-la3 (i 0) (k 0)))))

(prove-lemma disjoint-1 (rewrite)
     (implies (and (disjoint a m b n)
                   (leq j m)
                   (leq (plus k l) n))
              (disjoint a j (add 32 b k) l))
     ((use (disjoint-la3 (i 0)))))

(prove-lemma disjoint-2 (rewrite)
     (implies (and (disjoint a m b n)
                   (leq (plus i j) m)
                   (leq l n))
```

```
              (disjoint (add 32 a i) j b l))
     ((use (disjoint-la3 (k 0)))))

(prove-lemma disjoint-3 (rewrite)
     (implies (and (disjoint a m b n)
                   (leq (plus i j) m)
                   (leq (plus k l) n))
              (disjoint (add 32 a i) j (add 32 b k) l)))

;
(prove-lemma disjoint-4 (rewrite)
     (implies (and (disjoint (add 32 a i) m b n)
                   (leq (plus j (index-n 0 i)) m)
                   (leq l n))
              (disjoint a j b l))
     ((use (disjoint-2 (a (add 32 a i)) (i (index-n 0 i))))))

(prove-lemma disjoint-5 (rewrite)
     (implies (and (disjoint (add 32 a i) m b n)
                   (leq (plus j (index-n 0 i)) m)
                   (leq (plus k l) n))
              (disjoint a j (add 32 b k) l))
     ((use (disjoint-3 (a (add 32 a i)) (i (index-n 0 i))))))

(prove-lemma disjoint-6 (rewrite)
     (implies (and (disjoint (add 32 a i) m b n)
                   (leq (plus j (index-n i1 i)) m)
                   (leq l n))
              (disjoint (add 32 a i1) j b l))
     ((use (disjoint-2 (a (add 32 a i)) (i (index-n i1 i))))))

(prove-lemma disjoint-7 (rewrite)
     (implies (and (disjoint (add 32 a i) m b n)
                   (leq (plus j (index-n i1 i)) m)
                   (leq (plus k l) n))
              (disjoint (add 32 a i1) j (add 32 b k) l))
     ((use (disjoint-3 (a (add 32 a i)) (i (index-n i1 i))))))

;
(prove-lemma disjoint-8 (rewrite)
     (implies (and (disjoint a m (add 32 b k) n)
                   (leq j m)
                   (leq (plus l (index-n 0 k)) n))
              (disjoint a j b l))
     ((use (disjoint-1 (b (add 32 b k)) (k (index-n 0 k))))))

(prove-lemma disjoint-9 (rewrite)
     (implies (and (disjoint a m (add 32 b k) n)
                   (leq (plus i j) m)
                   (leq (plus l (index-n 0 k)) n))
              (disjoint (add 32 a i) j b l))
     ((use (disjoint-3 (b (add 32 b k)) (k (index-n 0 k))))))

(prove-lemma disjoint-10 (rewrite)
```

```
     (implies (and (disjoint a m (add 32 b k) n)
                   (leq j m)
                   (leq (plus l (index-n k1 k)) n))
              (disjoint a j (add 32 b k1) l))
     ((use (disjoint-1 (b (add 32 b k)) (k (index-n k1 k))))))

(prove-lemma disjoint-11 (rewrite)
     (implies (and (disjoint a m (add 32 b k) n)
                   (leq (plus i j) m)
                   (leq (plus l (index-n k1 k)) n))
              (disjoint (add 32 a i) j (add 32 b k1) l))
     ((use (disjoint-3 (b (add 32 b k)) (k (index-n k1 k))))))

;
(prove-lemma disjoint-12 (rewrite)
     (implies (and (disjoint (add 32 a i) m (add 32 b k) n)
                   (leq (plus j (index-n 0 i)) m)
                   (leq (plus l (index-n 0 k)) n))
              (disjoint a j b l))
     ((use (disjoint-9 (a (add 32 a i)) (i (index-n 0 i))))))

(prove-lemma disjoint-13 (rewrite)
     (implies (and (disjoint (add 32 a i) m (add 32 b k) n)
                   (leq (plus j (index-n i1 i)) m)
                   (leq (plus l (index-n 0 k)) n))
              (disjoint (add 32 a i1) j b l))
     ((use (disjoint-9 (a (add 32 a i)) (i (index-n i1 i))))))

(prove-lemma disjoint-14 (rewrite)
     (implies (and (disjoint (add 32 a i) m (add 32 b k) n)
                   (leq (plus j (index-n 0 i)) m)
                   (leq (plus l (index-n k1 k)) n))
              (disjoint a j (add 32 b k1) l))
     ((use (disjoint-11 (a (add 32 a i)) (i (index-n 0 i))))))

(prove-lemma disjoint-15 (rewrite)
     (implies (and (disjoint (add 32 a i) m (add 32 b k) n)
                   (leq (plus j (index-n i1 i)) m)
                   (leq (plus l (index-n k1 k)) n))
              (disjoint (add 32 a i1) j (add 32 b k1) l))
     ((use (disjoint-11 (a (add 32 a i)) (i (index-n i1 i))))))

; the commutativity of disjoint.
(prove-lemma disjoint-commutativity (rewrite)
     (equal (disjoint x m y n)
            (disjoint y n x m)))

; the dual events of the above 16 events.
(prove-lemma disjoint-0~ (rewrite)
     (implies (and (disjoint a m b n)
                   (leq j m)
                   (leq l n))
              (disjoint b l a j)))
```

```
(prove-lemma disjoint-1~ (rewrite)
     (implies (and (disjoint a m b n)
                   (leq j m)
                   (leq (plus k l) n))
              (disjoint (add 32 b k) l a j)))

(prove-lemma disjoint-2~ (rewrite)
     (implies (and (disjoint a m b n)
                   (leq (plus i j) m)
                   (leq l n))
              (disjoint b l (add 32 a i) j))
     ((use (disjoint-2))
      (disable disjoint-2)))

(prove-lemma disjoint-3~ (rewrite)
     (implies (and (disjoint a m b n)
                   (leq (plus i j) m)
                   (leq (plus k l) n))
              (disjoint (add 32 b k) l (add 32 a i) j)))

;
(prove-lemma disjoint-4~ (rewrite)
     (implies (and (disjoint (add 32 a i) m b n)
                   (leq (plus j (index-n 0 i)) m)
                   (leq l n))
              (disjoint b l a j)))

(prove-lemma disjoint-5~ (rewrite)
     (implies (and (disjoint (add 32 a i) m b n)
                   (leq (plus j (index-n 0 i)) m)
                   (leq (plus k l) n))
              (disjoint (add 32 b k) l a j)))

(prove-lemma disjoint-6~ (rewrite)
     (implies (and (disjoint (add 32 a i) m b n)
                   (leq (plus j (index-n i1 i)) m)
                   (leq l n))
              (disjoint b l (add 32 a i1) j))
     ((use (disjoint-6))
      (disable disjoint-6)))

(prove-lemma disjoint-7~ (rewrite)
     (implies (and (disjoint (add 32 a i) m b n)
                   (leq (plus j (index-n i1 i)) m)
                   (leq (plus k l) n))
              (disjoint (add 32 b k) l (add 32 a i1) j)))
;
(prove-lemma disjoint-8~ (rewrite)
     (implies (and (disjoint a m (add 32 b k) n)
                   (leq j m)
                   (leq (plus l (index-n 0 k)) n))
              (disjoint b l a j)))

(prove-lemma disjoint-9~ (rewrite)
```

```
     (implies (and (disjoint a m (add 32 b k) n)
                   (leq (plus i j) m)
                   (leq (plus l (index-n 0 k)) n))
              (disjoint b l (add 32 a i) j))
     ((use (disjoint-9))
      (disable disjoint-9)))

(prove-lemma disjoint-10~ (rewrite)
     (implies (and (disjoint a m (add 32 b k) n)
                   (leq j m)
                   (leq (plus l (index-n k1 k)) n))
              (disjoint (add 32 b k1) l a j)))

(prove-lemma disjoint-11~ (rewrite)
     (implies (and (disjoint a m (add 32 b k) n)
                   (leq (plus i j) m)
                   (leq (plus l (index-n k1 k)) n))
              (disjoint (add 32 b k1) l (add 32 a i) j)))
;
(prove-lemma disjoint-12~ (rewrite)
     (implies (and (disjoint (add 32 a i) m (add 32 b k) n)
                   (leq (plus j (index-n 0 i)) m)
                   (leq (plus l (index-n 0 k)) n))
              (disjoint b l a j)))

(prove-lemma disjoint-13~ (rewrite)
     (implies (and (disjoint (add 32 a i) m (add 32 b k) n)
                   (leq (plus j (index-n i1 i)) m)
                   (leq (plus l (index-n 0 k)) n))
              (disjoint b l (add 32 a i1) j))
     ((use (disjoint-13))
      (disable disjoint-13)))

(prove-lemma disjoint-14~ (rewrite)
     (implies (and (disjoint (add 32 a i) m (add 32 b k) n)
                   (leq (plus j (index-n 0 i)) m)
                   (leq (plus l (index-n k1 k)) n))
              (disjoint (add 32 b k1) l a j)))

(prove-lemma disjoint-15~ (rewrite)
     (implies (and (disjoint (add 32 a i) m (add 32 b k) n)
                   (leq (plus j (index-n i1 i)) m)
                   (leq (plus l (index-n k1 k)) n))
              (disjoint (add 32 b k1) l (add 32 a i1) j)))

; disjoint with asl.
(prove-lemma times-plus-lessp-cancel (rewrite)
     (implies (and (leq a k)
                   (leq b k)
                   (lessp a b))
              (equal (lessp (plus a (times i k)) (plus b (times j k)))
                     (leq i j)))
     ((induct (difference i j))))
```

```
(prove-lemma disjoint-2-asl (rewrite)
     (implies (and (disjoint a m b n)
                   (equal opsz (exp 2 cnt))
                   (lessp (nat-to-int x 32) (quotient m opsz))
                   (numberp (nat-to-int x 32))
                   (leq l n))
              (disjoint (add 32 a (asl 32 x cnt)) opsz b l))
     ((enable nat-to-int)))

(prove-lemma disjoint-2~-asl (rewrite)
     (implies (and (disjoint a m b n)
                   (equal opsz (exp 2 cnt))
                   (lessp (nat-to-int x 32) (quotient m opsz))
                   (numberp (nat-to-int x 32))
                   (leq l n))
              (disjoint b l (add 32 a (asl 32 x cnt)) opsz))
     ((enable nat-to-int)))

(prove-lemma disjoint-3-asl (rewrite)
     (implies (and (disjoint a m b n)
                   (equal opsz (exp 2 cnt))
                   (lessp (nat-to-int x 32) (quotient m opsz))
                   (numberp (nat-to-int x 32))
                   (leq (plus k l) n))
              (disjoint (add 32 a (asl 32 x cnt)) opsz (add 32 b k) l))
     ((enable nat-to-int)))

(prove-lemma disjoint-3~-asl (rewrite)
     (implies (and (disjoint a m b n)
                   (equal opsz (exp 2 cnt))
                   (lessp (nat-to-int x 32) (quotient m opsz))
                   (numberp (nat-to-int x 32))
                   (leq (plus k l) n))
              (disjoint (add 32 b k) l (add 32 a (asl 32 x cnt)) opsz))
     ((enable nat-to-int)))

(prove-lemma disjoint-9-asl (rewrite)
     (implies (and (disjoint a m (add 32 b k) n)
                   (equal opsz (exp 2 cnt))
                   (lessp (nat-to-int x 32) (quotient m opsz))
                   (numberp (nat-to-int x 32))
                   (leq (plus l (index-n 0 k)) n))
              (disjoint (add 32 a (asl 32 x cnt)) opsz b l))
     ((enable nat-to-int)))

(prove-lemma disjoint-9~-asl (rewrite)
     (implies (and (disjoint a m (add 32 b k) n)
                   (equal opsz (exp 2 cnt))
                   (lessp (nat-to-int x 32) (quotient m opsz))
                   (numberp (nat-to-int x 32))
                   (leq (plus l (index-n 0 k)) n))
              (disjoint b l (add 32 a (asl 32 x cnt)) opsz))
     ((enable nat-to-int)))
```

```
(prove-lemma disjoint-11-asl (rewrite)
     (implies (and (disjoint a m (add 32 b k) n)
                   (equal opsz (exp 2 cnt))
                   (lessp (nat-to-int x 32) (quotient m opsz))
                   (numberp (nat-to-int x 32))
                   (leq (plus l (index-n k1 k)) n))
              (disjoint (add 32 a (asl 32 x cnt)) opsz (add 32 b k1) l))
     ((enable nat-to-int)))

(prove-lemma disjoint-11~-asl (rewrite)
     (implies (and (disjoint a m (add 32 b k) n)
                   (equal opsz (exp 2 cnt))
                   (lessp (nat-to-int x 32) (quotient m opsz))
                   (numberp (nat-to-int x 32))
                   (leq (plus l (index-n k1 k)) n))
              (disjoint (add 32 b k1) l (add 32 a (asl 32 x cnt)) opsz))
     ((enable nat-to-int)))

(disable times-plus-lessp-cancel)

; a set of rewrite rules for disjoint0 and disjoint.
(prove-lemma disjoint0-x-x (rewrite)
     (equal (disjoint0 x m x)
            (zerop m)))

(prove-lemma disjoint0-deduction0 (rewrite)
     (disjoint0 x 0 y))

(prove-lemma disjoint0-deduction1 (rewrite)
     (and (equal (disjoint0 x m (add 32 x y))
                 (disjoint0 0 m y))
          (equal (disjoint0 (add 32 x y) m x)
                 (disjoint0 y m 0)))
     ((enable add)))

(prove-lemma disjoint0-deduction2 (rewrite)
     (equal (disjoint0 (add 32 x y) m (add 32 x z))
            (disjoint0 y m z))
     ((induct (disjoint0 y m z))
      (enable add)))

(prove-lemma disjoint-x-x (rewrite)
     (equal (disjoint x m x n)
            (or (zerop m) (zerop n))))

(prove-lemma disjoint-deduction0 (rewrite)
     (and (disjoint x m y 0)
          (disjoint x 0 y n)))

(prove-lemma disjoint-deduction1 (rewrite)
     (and (equal (disjoint x m (add 32 x y) n)
                 (disjoint 0 m y n))
          (equal (disjoint (add 32 x y) m x n)
                 (disjoint y m 0 n))))
```

```
(prove-lemma disjoint-deduction2 (rewrite)
     (equal (disjoint (add 32 x y) m (add 32 x z) n)
            (disjoint y m z n)))

;                     INDEX-N
(prove-lemma index-n-0 (rewrite)
     (and (equal (index-n x 0) (head x 32))
          (equal (index-n 0 x) (neg 32 x))))

(prove-lemma index-n-x-x (rewrite)
     (equal (index-n x x) 0))

(prove-lemma index-n-deduction0 (rewrite)
     (and (equal (index-n x (neg 32 y))
                 (index-n (add 32 x y) 0))
          (equal (index-n (neg 32 x) y)
                 (index-n 0 (add 32 x y)))))

(prove-lemma index-n-deduction1 (rewrite)
     (and (equal (index-n (add 32 x y) x)
                 (index-n y 0))
          (equal (index-n y (add 32 y x))
                 (index-n 0 x))))

(prove-lemma index-n-deduction2 (rewrite)
     (equal (index-n (add 32 z x) (add 32 z y))
            (index-n x y)))

(prove-lemma disjoint-deduction3 (rewrite)
     (equal (disjoint (add 32 y x) m (add 32 z x) n)
            (disjoint y m z n))
     ((enable add-commutativity)))

(disable index-n)

;           READ-MEM/WRITE-MEM WITH MEM-LST/MEM-ILST
;
; starting at address a, the contents of the memory are equal to the elements
; of lst.
(defn mem-lst (opsz a mem n lst)
  (if (zerop n)
      (nlistp lst)
    (and (equal (read-mem a mem opsz) (car lst))
         (mem-lst opsz (add (1) a opsz) mem (sub1 n) (cdr lst)))))

(defn mem-ilst (opsz a mem n lst)
  (if (zerop n)
      (nlistp lst)
    (and (equal (nat-to-int (read-mem a mem opsz) (times 8 opsz)) (car lst))
         (mem-ilst opsz (add (1) a opsz) mem (sub1 n) (cdr lst)))))

; every element in lst is in "good" range.
(prove-lemma mem-lst-nat-rangep (rewrite)
```

```
     (implies (and (mem-lst opsz a mem n lst)
                   (leq (times 8 opsz) k))
              (nat-rangep (get-nth i lst) k)))

(prove-lemma mem-ilst-int-rangep (rewrite)
     (implies (and (mem-ilst opsz a mem n lst)
                   (equal oplen (times 8 opsz)))
              (int-rangep (get-nth i lst) oplen)))

; every element in lst is a natural number.
(prove-lemma mem-lst-numberp (rewrite)
     (implies (mem-lst opsz a mem n lst)
              (numberp (get-nth i lst))))

; every element in lst is bounded by 2^(8*opsz).
(prove-lemma mem-lst-lessp (rewrite)
     (implies (and (mem-lst opsz a mem n lst)
                   (equal k (exp 2 (times 8 opsz))))
              (lessp (get-nth i lst) k))
     ((use (mem-lst-nat-rangep (k (times 8 opsz))))
      (enable nat-rangep)))

; every element in lst is an integer.
(prove-lemma mem-ilst-integerp (rewrite)
     (implies (mem-ilst opsz a mem n lst)
              (integerp (get-nth i lst))))

; a trivial, but useful, instantiation!
(prove-lemma readm-mem-lst (rewrite)
     (mem-lst opsz a mem n (readm-mem opsz a mem n)))

; head with mem-lst/mem-ilst.
(prove-lemma head-mem-lst (rewrite)
     (equal (mem-lst opsz (head x 32) mem n lst)
            (mem-lst opsz x mem n lst)))

(prove-lemma head-mem-ilst (rewrite)
     (equal (mem-ilst opsz (head x 32) mem n lst)
            (mem-ilst opsz x mem n lst)))

(prove-lemma read-mem-non-numberp (rewrite)
     (implies (not (numberp x))
              (equal (read-mem x mem k) (read-mem 0 mem k)))
     ((enable add)))

(prove-lemma mem-lst-non-numberp (rewrite)
     (implies (not (numberp x))
              (equal (mem-lst opsz x mem n lst)
                     (mem-lst opsz 0 mem n lst))))

(prove-lemma mem-lst-same (rewrite)
     (implies (and (mem-lst opsz x mem n lst)
                   (equal (nat-to-uint x) (nat-to-uint y)))
              (mem-lst opsz y mem n lst))
```

```
     ((enable nat-to-uint)))

(disable read-mem-non-numberp)
(disable mem-lst-non-numberp)
(disable mem-lst-same)

;                    CANONICAL FORMS
; Sometimes,  it is possible to have more than one representation
; for one concept.  We have to tell the prover only to use the canonical
; representations.
(prove-lemma pc-byte-read=pc-read-mem-1 (rewrite)
     (equal (pc-byte-read addr mem)
            (pc-read-mem addr mem 1))
     ((expand (pc-read-mem addr mem 1))))

(prove-lemma byte-read=read-mem-1 (rewrite)
     (equal (byte-read addr mem)
            (read-mem addr mem 1))
     ((expand (read-mem addr mem 1))))

(prove-lemma byte-write=write-mem-1 (rewrite)
     (equal (byte-write value addr mem)
            (write-mem value addr mem 1))
     ((expand (write-mem value addr mem 1))))

(prove-lemma byte-writep=write-memp-1 (rewrite)
     (equal (byte-writep x mem)
            (write-memp x mem 1))
     ((expand (write-memp x mem 1))))

;        GET-LST, PUT-LST, GET-VALS, PUT-VALS, MCAR, and MCDR
(defn get-lst (opsz m lst n)
  (if (zerop n)
      0
    (app (times 8 opsz)
         (get-nth (plus m (sub1 n)) lst)
         (get-lst opsz m lst (sub1 n)))))

(defn put-lst (opsz v n lst k)
  (if (zerop k)
      lst
    (put-lst opsz (tail v (times 8 opsz)) n
             (put-nth (head v (times 8 opsz)) (plus n (sub1 k)) lst)
             (sub1 k))))

(defn get-vals (m lst n)
  (if (zerop n)
      nil
    (append (get-vals m lst (sub1 n))
            (list (get-nth (plus m (sub1 n)) lst)))))

(defn put-vals (vals m lst n)
  (if (zerop n)
      lst
```

```
    (put-vals vals m (put-nth (get-nth (sub1 n) vals) (plus m (sub1 n)) lst)
              (sub1 n))))

(defn bv-to-lst (opsz bv n)
  (if (zerop n)
      nil
    (append (bv-to-lst opsz (tail bv (times 8 opsz)) (sub1 n))
            (list (head bv (times 8 opsz))))))

(defn lst-to-bv (opsz lst n)
  (if (zerop n)
      0
    (app (times 8 opsz)
         (get-nth (sub1 n) lst)
         (lst-to-bv opsz lst (sub1 n)))))

; mcar returns the list consisting of the first n elements of lst.
(defn mcar (n lst)
  (if (zerop n)
      nil
    (cons (car lst) (mcar (sub1 n) (cdr lst)))))

; mcdr returns the list discarding the first n elements of lst.
(defn mcdr (n lst)
  (if (zerop n)
      lst
    (mcdr (sub1 n) (cdr lst))))

; a predicate to recognize proper list.
(defn proper-lstp (lst)
  (if (nlistp lst)
      (equal lst nil)
    (proper-lstp (cdr lst))))

(prove-lemma append-len (rewrite)
     (equal (len (append x y)) (plus (len x) (len y))))

(prove-lemma get-vals-len (rewrite)
     (equal (len (get-vals m lst n)) (fix n)))

(prove-lemma bv-to-lst-len (rewrite)
     (equal (len (bv-to-lst opsz bv n)) (fix n)))

(prove-lemma get-vals-proper-lstp (rewrite)
     (proper-lstp (get-vals m lst n)))

(prove-lemma bv-to-lst-proper-lstp (rewrite)
     (proper-lstp (bv-to-lst opsz bv n)))

(prove-lemma mcdr-listp-len (rewrite)
     (equal (listp (mcdr n lst))
            (lessp n (len lst))))

(prove-lemma cdr-mcdr (rewrite)
```

```
     (equal (cdr (mcdr n lst))
            (mcdr (add1 n) lst)))

(prove-lemma mcar-mcar (rewrite)
     (implies (leq m n)
              (equal (mcar m (mcar n lst))
                     (mcar m lst))))

(prove-lemma mcdr-mcdr (rewrite)
     (equal (mcdr n (mcdr m lst))
            (mcdr (plus m n) lst)))

(prove-lemma mcar-nth (rewrite)
     (equal (get-nth n (mcar m lst))
            (if (lessp n m) (get-nth n lst) 0))
     ((enable get-nth)))

(prove-lemma mcdr-nth (rewrite)
     (equal (get-nth n (mcdr m lst))
            (get-nth (plus m n) lst)))

(prove-lemma get-lst-cdr (rewrite)
     (equal (get-lst opsz m (cdr lst) n)
            (get-lst opsz (add1 m) lst n))
     ((enable get-nth)))

(prove-lemma get-lst-mcdr (rewrite)
     (equal (get-lst opsz m (mcdr j lst) n)
            (get-lst opsz (plus m j) lst n)))

(prove-lemma get-lst-mcar (rewrite)
     (implies (leq (plus m n) j)
              (equal (get-lst opsz m (mcar j lst) n)
                     (get-lst opsz m lst n)))
     ((induct (get-lst opsz m lst n))))

(prove-lemma get-vals-cdr (rewrite)
     (equal (get-vals m (cdr lst) n)
            (get-vals (add1 m) lst n))
     ((enable get-nth)))

(prove-lemma get-vals-mcdr (rewrite)
     (equal (get-vals m (mcdr j lst) n)
            (get-vals (plus m j) lst n)))

(prove-lemma get-vals-mcar (rewrite)
     (implies (leq (plus m n) j)
              (equal (get-vals m (mcar j lst) n)
                     (get-vals m lst n)))
     ((induct (get-vals m lst n))))

(prove-lemma get-nth-append (rewrite)
     (equal (get-nth i (append x y))
            (if (lessp i (len x))
```

```
              (get-nth i x)
             (get-nth (difference i (len x)) y)))
     ((enable get-nth)))

(prove-lemma get-vals-append (rewrite)
     (implies (leq n (len x))
              (equal (get-vals 0 (append x y) n)
                     (get-vals 0 x n))))

(prove-lemma put-vals-append (rewrite)
     (implies (leq n (len x))
              (equal (put-vals (append x y) m lst n)
                     (put-vals x m lst n)))
     ((induct (put-vals x m lst n))))

; another induction hint for many theorems about memory.
(defn mem-induct2 (opsz addr i n lst j)
  (if (zerop n)
      t
    (mem-induct2 opsz
                 (add 32 addr opsz)
                 (sub1 i)
                 (sub1 n)
                 (cdr lst)
                 (difference j opsz))))

(prove-lemma read-mem-lst-la (rewrite)
     (implies (and (mem-lst opsz a mem n lst)
                   (equal (remainder j opsz) 0)
                   (lessp (quotient j opsz) n))
              (equal (read-mem (add 32 a j) mem opsz)
                     (get-nth (quotient j opsz) lst)))
     ((induct (mem-induct2 opsz a i n lst j))
      (enable add-plus)))

(prove-lemma mem-lst-get-lst0 (rewrite)
     (implies (and (mem-lst 1 a mem n lst)
                   (leq k n))
              (equal (nat-to-uint (read-mem a mem k))
                     (get-lst 1 0 lst k)))
     ((induct (read-mem a mem k))))

(prove-lemma mem-lst-get-lst (rewrite)
     (implies (and (mem-lst 1 a mem n lst)
                   (leq (plus (nat-to-uint j) k) n))
              (equal (nat-to-uint (read-mem (add 32 a j) mem k))
                     (get-lst 1 (nat-to-uint j) lst k)))
     ((induct (read-mem a mem k))
      (enable add-plus plus-add1-1)))

(prove-lemma mem-lst-get-vals0 (rewrite)
     (implies (and (mem-lst 1 a mem n lst)
                   (leq k n))
              (equal (bv-to-lst 1 (read-mem a mem k) k)
```

```
                    (get-vals 0 lst k)))
     ((induct (read-mem a mem k))))

(prove-lemma mem-lst-get-vals (rewrite)
     (implies (and (mem-lst 1 a mem n lst)
                   (leq (plus (nat-to-uint j) k) n))
              (equal (bv-to-lst 1 (read-mem (add 32 a j) mem k) k)
                     (get-vals (nat-to-uint j) lst k)))
     ((induct (read-mem a mem k))
      (enable add-plus plus-add1-1)))

; read is equivalent to get-nth.
(prove-lemma read-mem-lst0 (rewrite)
     (implies (and (mem-lst opsz a mem n lst)
                   (not (zerop n)))
              (equal (nat-to-uint (read-mem a mem opsz))
                     (get-nth 0 lst))))

(prove-lemma iread-mem-get0 (rewrite)
     (implies (and (mem-ilst opsz a mem n lst)
                   (not (zerop n))
                   (equal oplen (times 8 opsz)))
              (equal (nat-to-int (read-mem a mem opsz) oplen)
                     (get-nth 0 lst))))

(prove-lemma read-mem-lst (rewrite)
     (implies (and (mem-lst opsz a mem n lst)
                   (equal (remainder (nat-to-uint j) opsz) 0)
                   (lessp (quotient (nat-to-uint j) opsz) n))
              (equal (nat-to-uint (read-mem (add 32 a j) mem opsz))
                     (get-nth (quotient (nat-to-uint j) opsz) lst))))

(prove-lemma read-mem-lst-int (rewrite)
     (implies (and (mem-lst opsz a mem n lst)
                   (equal (remainder (nat-to-int j 32) opsz) 0)
                   (lessp (quotient (nat-to-int j 32) opsz) n)
                   (numberp (nat-to-int j 32)))
              (equal (nat-to-uint (read-mem (add 32 a j) mem opsz))
                     (get-nth (quotient (nat-to-int j 32) opsz) lst)))
     ((enable nat-to-int)))

(prove-lemma read-mem-ilst (rewrite)
     (implies (and (mem-ilst opsz a mem n lst)
                   (equal oplen (times 8 opsz))
                   (equal (remainder (nat-to-uint j) opsz) 0)
                   (lessp (quotient (nat-to-uint j) opsz) n))
              (equal (nat-to-int (read-mem (add 32 a j) mem opsz) oplen)
                     (get-nth (quotient (nat-to-uint j) opsz) lst)))
     ((induct (mem-induct2 opsz a i n lst j))
      (enable add-plus)))

(prove-lemma read-mem-ilst-int (rewrite)
     (implies (and (mem-ilst opsz a mem n lst)
                   (equal oplen (times 8 opsz))
```

```
                (equal (remainder (nat-to-uint j) opsz) 0)
                (lessp (quotient (nat-to-uint j) opsz) n)
                (numberp (nat-to-int j 32)))
           (equal (nat-to-int (read-mem (add 32 a j) mem opsz) oplen)
                  (get-nth (quotient (nat-to-uint j) opsz) lst)))
     ((induct (mem-induct2 opsz a i n lst j))
      (enable add-plus nat-to-int)))

; write to some location else does not affect mem-lst.
(prove-lemma write-else-mem-lst (rewrite)
     (implies (disjoint a (times opsz n) x m)
              (equal (mem-lst opsz a (write-mem value x mem m) n lst)
                     (mem-lst opsz a mem n lst)))
     ((enable times)))

(prove-lemma write-else-mem-ilst (rewrite)
     (implies (disjoint a (times opsz n) x m)
              (equal (mem-ilst opsz a (write-mem value x mem m) n lst)
                     (mem-ilst opsz a mem n lst)))
     ((enable times)))

; some conditions that makes disjoint true.
(prove-lemma disjoint0-leq (rewrite)
     (implies (and (leq (plus a m) 4294967296)
                   (lessp b a))
              (disjoint0 a m b))
     ((induct (disjoint0 a m b))
      (enable add nat-rangep)))

(prove-lemma disjoint-leq (rewrite)
     (implies (and (leq (plus a m) 4294967296)
                   (leq (plus b n) a))
              (disjoint a m b n))
     ((induct (disjoint a m b n))
      (enable add nat-rangep)))

(prove-lemma disjoint-leq1 (rewrite)
     (implies (and (leq (plus a m) 4294967296)
                   (leq (plus b n) a))
              (disjoint b n a m)))

(prove-lemma plus-times-lessp (rewrite)
     (implies (and (lessp (plus x (times y z)) w)
                   (lessp z1 z))
              (lessp (plus x y (times y z1)) w))
     ((induct (plus z a))))

(prove-lemma write-mem-lst-la (rewrite)
     (implies (and (mem-lst opsz a mem n lst)
                   (uint-rangep (times opsz n) 32)
                   (equal v1 (head v (times 8 opsz)))
                   (equal j (times opsz i))
                   (lessp i n)
                   (numberp v))
```

```
             (mem-lst opsz
                      a
                      (write-mem v (add 32 a j) mem opsz)
                      n
                      (put-nth v1 i lst)))
     ((induct (mem-induct2 opsz a i n lst j))
      (enable times add-plus) (disable plus)))

(defn mem-induct3 (v v1 a mem i lst k)
  (if (zerop k)
      t
    (mem-induct3 (tail v 8) (tail v1 8)
                 a (byte-write v (add 32 a (plus i (sub1 k))) mem)
                 i (put-nth (head v 8) (plus i (sub1 k)) lst)
                 (sub1 k))))

(prove-lemma mem-lst-put-lst (rewrite)
     (implies (and (mem-lst 1 a mem n lst)
                   (uint-rangep n 32)
                   (nat-rangep v (times 8 k))
                   (equal (nat-to-uint j) i)
                   (equal v1 (nat-to-uint v))
                   (leq (plus i k) n)
                   (numberp j)
                   (numberp v))
              (mem-lst 1 a (write-mem v (add 32 a j) mem k)
                       n (put-lst 1 v1 i lst k)))
     ((induct (mem-induct3 v v1 a mem i lst k))
      (enable add-plus)))

(defn mem-induct4 (v vals a mem i lst k)
  (if (zerop k)
      t
    (mem-induct4 (tail v 8) (get-vals 0 vals (sub1 k))
                 a (byte-write v (add 32 a (plus i (sub1 k))) mem)
                 i (put-nth (head v 8) (plus i (sub1 k)) lst)
                 (sub1 k))))

(prove-lemma get-vals-0 (rewrite)
     (implies (and (equal (len lst) n)
                   (proper-lstp lst))
              (equal (get-vals 0 lst n) lst)))

(prove-lemma mem-lst-put-vals (rewrite)
     (implies (and (mem-lst 1 a mem n lst)
                   (uint-rangep n 32)
                   (nat-rangep v (times 8 k))
                   (equal (nat-to-uint j) i)
                   (equal vals (bv-to-lst 1 v k))
                   (leq (plus i k) n)
                   (numberp j)
                   (numberp v))
              (mem-lst 1 a (write-mem v (add 32 a j) mem k)
                       n (put-vals vals i lst k)))
```

```
     ((induct (mem-induct4 v vals a mem i lst k))
      (enable add-plus)))

(disable get-vals-0)

(prove-lemma write-mem-lst (rewrite)
     (implies (and (mem-lst opsz a mem n lst)
                   (uint-rangep (times opsz n) 32)
                   (nat-rangep v (times 8 opsz))
                   (equal (nat-to-uint j) (times opsz i))
                   (equal v1 (nat-to-uint v))
                   (lessp i n)
                   (numberp j)
                   (numberp v))
              (mem-lst opsz a
                       (write-mem v (add 32 a j) mem opsz)
                       n (put-nth v1 i lst))))

(prove-lemma write-mem-ilst (rewrite)
     (implies (and (mem-ilst opsz a mem n lst)
                   (uint-rangep (times opsz n) 32)
                   (equal (nat-to-uint j) (times opsz i))
                   (nat-rangep v (times 8 opsz))
                   (equal v1 (nat-to-int v (times 8 opsz)))
                   (lessp i n)
                   (numberp j)
                   (numberp v))
              (mem-ilst opsz a
                        (write-mem v (add 32 a j) mem opsz)
                        n (put-nth v1 i lst)))
     ((induct (mem-induct2 opsz a i n lst j))
      (enable times add-plus) (disable plus)))

(prove-lemma write-mem-lst0 (rewrite)
     (implies (and (mem-lst opsz a mem n lst)
                   (uint-rangep (times opsz n) 32)
                   (nat-rangep v (times 8 opsz))
                   (equal v1 (nat-to-uint v))
                   (not (zerop n))
                   (numberp v))
              (mem-lst opsz a (write-mem v a mem opsz) n (put-nth v1 0 lst)))
     ((use (write-mem-lst (i 0) (j 0)))))

(prove-lemma write-mem-ilst0 (rewrite)
     (implies (and (mem-ilst opsz a mem n lst)
                   (uint-rangep (times opsz n) 32)
                   (nat-rangep v (times 8 opsz))
                   (equal v1 (nat-to-int v (times 8 opsz)))
                   (not (zerop n))
                   (numberp v))
              (mem-ilst opsz a (write-mem v a mem opsz) n (put-nth v1 0 lst)))
     ((use (write-mem-ilst (i 0) (j 0)))))

(prove-lemma write-mem-lst-int (rewrite)
```

```
     (implies (and (mem-lst opsz a mem n lst)
                   (uint-rangep (times opsz n) 32)
                   (equal (times opsz i) (nat-to-int j 32))
                   (numberp (nat-to-int j 32))
                   (nat-rangep v (times 8 opsz))
                   (equal v1 (nat-to-uint v))
                   (lessp i n)
                   (not (zerop opsz))
                   (numberp v))
              (mem-lst opsz a
                       (write-mem v (add 32 a j) mem opsz)
                       n (put-nth v1 i lst)))
     ((enable put-nth-0 nat-to-int nat-rangep)
      (disable put-nth)))

(prove-lemma write-mem-ilst-int (rewrite)
     (implies (and (mem-ilst opsz a mem n lst)
                   (uint-rangep (times opsz n) 32)
                   (equal (times opsz i) (nat-to-int j 32))
                   (numberp (nat-to-int j 32))
                   (nat-rangep v (times 8 opsz))
                   (equal v1 (nat-to-int v (times 8 opsz)))
                   (lessp i n)
                   (not (zerop opsz))
                   (numberp v))
              (mem-ilst opsz a
                        (write-mem v (add 32 a j) mem opsz)
                        n (put-nth v1 i lst)))
     ((enable put-nth-0 nat-to-int nat-rangep)
      (disable put-nth)))

;                    INTEGER VIEWS
;                    A + OPSZ * I
; to deal with the address calculation a+2*i or a+4*i, we introduce
; the following set of lemmas.
(prove-lemma read-mem-lst-asl (rewrite)
     (implies
      (and (mem-lst opsz a mem n lst)
           (equal opsz (exp 2 cnt))
           (lessp (nat-to-int i 32) n)
           (numberp (nat-to-int i 32)))
      (equal (nat-to-uint (read-mem (add 32 a (asl 32 i cnt)) mem opsz))
             (get-nth (nat-to-int i 32) lst)))
     ((use (read-mem-lst (opsz (exp 2 cnt)) (j (times (exp 2 cnt) i))))
      (enable nat-to-int)))

(prove-lemma read-mem-ilst-asl (rewrite)
     (implies
      (and (mem-ilst opsz a mem n lst)
           (equal opsz (exp 2 cnt))
           (equal oplen (times 8 opsz))
           (lessp (nat-to-int i 32) n)
           (numberp (nat-to-int i 32)))
      (equal (nat-to-int (read-mem (add 32 a (asl 32 i cnt)) mem opsz) oplen)
```

```
            (get-nth (nat-to-int i 32) lst)))
     ((use (read-mem-ilst (opsz (exp 2 cnt)) (j (times (exp 2 cnt) i))))
      (enable nat-to-int)))

(prove-lemma write-mem-lst-asl (rewrite)
     (implies (and (mem-lst opsz a mem n lst)
                   (uint-rangep (times opsz n) 32)
                   (equal opsz (exp 2 cnt))
                   (nat-rangep v (times 8 opsz))
                   (equal j (nat-to-int i 32))
                   (equal v1 (nat-to-uint v))
                   (lessp j n)
                   (numberp j)
                   (numberp v))
              (mem-lst opsz a
                       (write-mem v (add 32 a (asl 32 i cnt)) mem opsz)
                       n (put-nth v1 j lst)))
     ((use (write-mem-lst (opsz (exp 2 cnt)) (j (times (exp 2 cnt) i))))
      (enable nat-to-int)))

(prove-lemma write-mem-ilst-asl (rewrite)
     (implies (and (mem-ilst opsz a mem n lst)
                   (uint-rangep (times opsz n) 32)
                   (equal opsz (exp 2 cnt))
                   (nat-rangep v (times 8 opsz))
                   (equal v1 (nat-to-int v (times 8 opsz)))
                   (equal j (nat-to-int i 32))
                   (lessp j n)
                   (numberp v)
                   (numberp j))
              (mem-ilst opsz a
                        (write-mem v (add 32 a (asl 32 i cnt)) mem opsz)
                        n (put-nth v1 j lst)))
     ((use (write-mem-ilst (opsz (exp 2 cnt)) (j (times (exp 2 cnt) i))))
      (enable nat-to-int)))

(prove-lemma times-lessp-cancel1 (rewrite)
     (equal (lessp (times y x) (plus x (times z x)))
            (and (not (zerop x)) (leq y z)))
     ((use (times-lessp-cancel (z (add1 z))))))

(prove-lemma read-memp-ram1-asl (rewrite)
     (implies (and (ram-addrp addr mem n)
                   (equal opsz (exp 2 cnt))
                   (lessp (nat-to-int x 32) (quotient n opsz))
                   (numberp (nat-to-int x 32)))
              (read-memp (add 32 addr (asl 32 x cnt)) mem opsz))
     ((use (ram-addrp-la1 (m (times opsz (quotient n opsz))))
           (read-memp-ram1 (k (times opsz (quotient n opsz)))
                           (i (times x (exp 2 cnt)))
                           (j opsz)))
      (enable nat-to-int)
      (disable ram-addrp-la1 ram-addrp-la2 read-memp-ram1)))
```

```
(prove-lemma read-memp-rom1-asl (rewrite)
     (implies (and (rom-addrp addr mem n)
                   (equal opsz (exp 2 cnt))
                   (lessp (nat-to-int x 32) (quotient n opsz))
                   (numberp (nat-to-int x 32)))
              (read-memp (add 32 addr (asl 32 x cnt)) mem opsz))
     ((use (rom-addrp-la1 (m (times opsz (quotient n opsz))))
           (read-memp-rom1 (k (times opsz (quotient n opsz)))
                           (i (times x (exp 2 cnt)))
                           (j opsz)))
      (enable nat-to-int)
      (disable rom-addrp-la1 rom-addrp-la2 read-memp-rom1)))

(prove-lemma write-memp-ram1-asl (rewrite)
     (implies (and (ram-addrp addr mem n)
                   (equal opsz (exp 2 cnt))
                   (lessp (nat-to-int x 32) (quotient n opsz))
                   (numberp (nat-to-int x 32)))
              (write-memp (add 32 addr (asl 32 x cnt)) mem opsz))
     ((use (ram-addrp-la1 (m (times opsz (quotient n opsz))))
           (write-memp-ram1 (k (times opsz (quotient n opsz)))
                            (i (times x (exp 2 cnt)))
                            (j opsz)))
      (enable nat-to-int)
      (disable ram-addrp-la1 ram-addrp-la2 write-memp-ram1)))

;             BASIC READM-MEM/WRITEM-MEM EVENTS
;
(prove-lemma writem-mem-maintain-pc-byte-readp (rewrite)
     (equal (pc-byte-readp x (writem-mem opsz vlst addr mem))
            (pc-byte-readp x mem)))

(prove-lemma writem-mem-maintain-pc-read-memp (rewrite)
     (equal (pc-read-memp x (writem-mem opsz vlst addr mem) n)
            (pc-read-memp x mem n)))

(prove-lemma writem-mem-maintain-byte-readp (rewrite)
     (equal (byte-readp x (writem-mem opsz vlst addr mem))
            (byte-readp x mem)))

(prove-lemma writem-mem-maintain-read-memp (rewrite)
     (equal (read-memp x (writem-mem opsz vlst addr mem) n)
            (read-memp x mem n)))

(prove-lemma writem-mem-maintain-byte-writep (rewrite)
     (equal (byte-writep x (writem-mem opsz vlst addr mem))
            (byte-writep x mem)))

(prove-lemma writem-mem-maintain-write-memp (rewrite)
     (equal (write-memp x (writem-mem opsz vlst addr mem) n)
            (write-memp x mem n)))

(prove-lemma writem-mem-maintain-rom-addrp (rewrite)
     (equal (rom-addrp x (writem-mem opsz vlst addr mem) n)
```

```
           (rom-addrp x mem n)))

(prove-lemma writem-mem-maintain-ram-addrp (rewrite)
     (equal (ram-addrp x (writem-mem opsz vlst addr mem) n)
            (ram-addrp x mem n)))

(prove-lemma pc-read-mem-writem-mem (rewrite)
     (implies (and (write-memp addr mem (times opsz (len vlst)))
                   (pc-read-memp x mem n))
              (equal (pc-read-mem x (writem-mem opsz vlst addr mem) n)
                     (pc-read-mem x mem n)))
     ((induct (writem-mem opsz vlst addr mem))))

(prove-lemma writem-mem-mcode-addrp (rewrite)
     (implies (and (pc-read-memp pc mem (len lst))
                   (write-memp addr mem (times opsz (len vlst))))
              (equal (mcode-addrp pc (writem-mem opsz vlst addr mem) lst)
                     (mcode-addrp pc mem lst)))
     ((induct (writem-mem opsz vlst addr mem))))

(prove-lemma writem-else-mem-lst (rewrite)
     (implies (disjoint addr (times opsz1 (len vlst)) a (times opsz n))
              (equal (mem-lst opsz a (writem-mem opsz1 vlst addr mem) n lst)
                     (mem-lst opsz a mem n lst)))
     ((induct (writem-mem opsz1 vlst addr mem))))

(prove-lemma writem-else-mem-ilst (rewrite)
     (implies (disjoint a (times opsz n) addr (times opsz1 (len vlst)))
              (equal (mem-ilst opsz a (writem-mem opsz1 vlst addr mem) n lst)
                     (mem-ilst opsz a mem n lst)))
     ((induct (writem-mem opsz1 vlst addr mem))))

(prove-lemma read-writem-mem (rewrite)
     (implies (disjoint addr n addr1 (times opsz (len vlst)))
              (equal (read-mem addr (writem-mem opsz vlst addr1 mem) n)
                     (read-mem addr mem n)))
     ((induct (writem-mem opsz vlst addr1 mem))))

(prove-lemma readm-write-mem (rewrite)
     (implies (disjoint addr (times opsz n) addr1 k)
              (equal (readm-mem opsz addr (write-mem value addr1 mem k) n)
                     (readm-mem opsz addr mem n)))
     ((enable times)))

(defn modn-lst (n lst)
  (if (nlistp lst)
      nil
    (cons (head (car lst) n)
          (modn-lst n (cdr lst)))))

(prove-lemma modn-readm-rn (rewrite)
     (equal (modn-lst oplen (readm-rn oplen rnlst rfile))
            (readm-rn oplen rnlst rfile)))
```

```
(prove-lemma readm-writem-mem (rewrite)
     (implies (and (uint-rangep (times opsz n) 32)
                   (equal n (len vlst)))
              (equal (readm-mem opsz addr (writem-mem opsz vlst addr mem) n)
                     (modn-lst (times 8 opsz) vlst)))
     ((disable plus)))

(prove-lemma disjoint-leq-uint (rewrite)
     (implies (and (leq (plus (nat-to-uint a) m) 4294967296)
                   (leq (plus (nat-to-uint b) n) (nat-to-uint a)))
              (disjoint a m b n)))

(prove-lemma disjoint-leq1-uint (rewrite)
     (implies (and (leq (plus (nat-to-uint a) m) 4294967296)
                   (leq (plus (nat-to-uint b) n) (nat-to-uint a)))
              (disjoint b n a m)))

(disable disjoint-leq)
(disable disjoint-leq1)
(disable disjoint0-leq)
(disable disjoint-commutativity)
(disable disjoint-leq-uint)
(disable disjoint-leq1-uint)

;              MEM-LST WITH MCAR AND MCDR
(prove-lemma mem-lst-plus (rewrite)
     (equal (mem-lst opsz a mem (plus m n) lst)
            (and (mem-lst opsz a mem m (mcar m lst))
                 (mem-lst opsz (add 32 a (times opsz m)) mem n (mcdr m lst))))
     ((induct (mem-lst opsz a mem m lst))
      (enable add-plus)))

(prove-lemma mem-lst-mcar (rewrite)
     (implies (and (mem-lst opsz a mem n lst)
                   (leq k n))
              (mem-lst opsz a mem k (mcar k lst))))

(prove-lemma mem-lst-mcar-1 (rewrite)
     (implies (mem-lst opsz a mem n (mcar n lst))
              (mem-lst opsz a mem (sub1 n) (mcar (sub1 n) lst)))
     ((use (mem-lst-plus (lst (mcar n lst)) (m (sub1 n)) (n 1)))))

(prove-lemma mem-lst-mcar-2 (rewrite)
     (implies
      (and (mem-lst opsz a mem n (mcar n lst))
           (leq k n))
      (mem-lst opsz a mem (difference n k) (mcar (difference n k) lst)))
     ((use (mem-lst-plus (lst (mcar n lst)) (m (difference n k)) (n k)))))

(prove-lemma plus-difference (rewrite)
        (equal (plus m (difference n m))
               (if (lessp n m) (fix m) (fix n))))

(prove-lemma mem-lst-len (rewrite)
```

```
     (implies (mem-lst opsz a mem n lst)
              (not (lessp n (len lst)))))

(prove-lemma mem-lst-mcdr (rewrite)
     (implies (and (mem-lst opsz a mem n lst)
                   (equal i (times opsz m)))
              (mem-lst opsz (add 32 a i) mem (difference n m) (mcdr m lst)))
     ((use (mem-lst-plus (n (difference n m))))))

(prove-lemma mem-lst-mcdr-0 (rewrite)
     (implies (mem-lst opsz a mem n lst)
              (mem-lst opsz (add 32 a opsz) mem (sub1 n) (mcdr 1 lst))))

(prove-lemma mem-lst-mcdr-uint (rewrite)
     (implies (and (mem-lst opsz a mem n lst)
                   (equal (nat-to-uint i) (times opsz m))
                   (not (zerop opsz)))
              (mem-lst opsz (add 32 a i) mem (difference n m) (mcdr m lst))))

(prove-lemma mem-lst-mcdr-1 (rewrite)
     (implies (mem-lst 1 (add 32 x i) mem (difference n j) (mcdr k lst))
              (mem-lst 1
                       (add 32 x (add 32 i 1))
                       mem
                       (difference (sub1 n) j)
                       (mcdr (add1 k) lst))))

(prove-lemma mem-lst-mcdr-uint-1 (rewrite)
     (implies
      (and (mem-lst opsz (add 32 a b) mem (difference n j) (mcdr k lst))
           (equal (nat-to-uint i) (times opsz m))
           (not (zerop opsz)))
      (mem-lst opsz
               (add 32 a (add 32 b i))
               mem
               (difference n (plus m j))
               (mcdr (plus m k) lst)))
     ((use (mem-lst-mcdr-uint (a (add 32 a b))
                              (n (difference n j))
                              (lst (mcdr k lst))))))

(disable mem-lst-len)
(disable plus-difference)

;                 LSTCPY AND LSTMCPY
(defn lstcpy (i1 lst1 i2 lst2 n)
  (if (zerop n)
      lst1
    (lstcpy (add1 i1) (put-nth (get-nth i2 lst2) i1 lst1)
            (add1 i2) lst2 (sub1 n))))

(prove-lemma lstcpy-0 (rewrite)
     (equal (lstcpy i1 lst1 i2 lst2 0) lst1))
```

```
(prove-lemma lstcpy-lstcpy (rewrite)
     (implies (and (equal j1 (plus h1 i1))
                   (equal j2 (plus h1 i2)))
              (equal (lstcpy j1 (lstcpy i1 lst1 i2 lst2 h1) j2 lst2 h2)
                     (lstcpy i1 lst1 i2 lst2 (plus h1 h2))))
     ((induct (lstcpy i1 lst1 i2 lst2 h1))
      (enable put-nth-0 get-nth-0)))

(defn lstmcpy (opsz i1 lst1 i2 lst2 n)
  (if (zerop n)
      lst1
    (lstmcpy opsz (plus opsz i1) (lstcpy i1 lst1 i2 lst2 opsz)
             (plus opsz i2) lst2 (sub1 n))))

(prove-lemma lstmcpy-cpy (rewrite)
     (equal (lstmcpy h i1 lst1 i2 lst2 n)
            (lstcpy i1 lst1 i2 lst2 (times h n))))

(prove-lemma put-commutativity (rewrite)
     (implies (lessp j i)
              (equal (put-nth v1 i (put-nth v2 j lst))
                     (put-nth v2 j (put-nth v1 i lst))))
     ((enable put-nth)))

(prove-lemma lstcpy-put-nth (rewrite)
     (implies (lessp i j1)
              (equal (put-nth v i (lstcpy j1 lst1 j2 lst2 n))
                     (lstcpy j1 (put-nth v i lst1) j2 lst2 n)))
     ((induct (lstcpy j1 lst1 j2 lst2 n))
      (enable put-nth-0)))

; a variant for lstcpy.
(defn lstcpy1 (i1 lst1 i2 lst2 n)
  (if (zerop n)
      lst1
    (lstcpy1 i1
             (put-nth (get-nth (plus i2 (sub1 n)) lst2)
                      (plus i1 (sub1 n))
                      lst1)
             i2 lst2 (sub1 n))))

(prove-lemma lstcpy-add1 (rewrite)
     (equal (lstcpy i1 (put-nth (get-nth (plus i2 h) lst2) (plus i1 h) lst1)
                    i2 lst2 h)
            (lstcpy i1 lst1 i2 lst2 (add1 h)))
     ((induct (lstcpy i1 lst1 i2 lst2 h))
      (enable get-nth-0 put-nth-0)))

(prove-lemma lstcpy-cpy1 (rewrite)
     (equal (lstcpy i1 lst1 i2 lst2 n)
            (lstcpy1 i1 lst1 i2 lst2 n)))

(prove-lemma put-get-lst-is-cpy (rewrite)
     (implies (mem-lst opsz x mem n2 lst2)
```

```
             (equal (put-lst opsz (get-lst opsz i2 lst2 n) i1 lst1 n)
                    (lstcpy i1 lst1 i2 lst2 n)))
     ((induct (lstcpy1 i1 lst1 i2 lst2 n))))

(prove-lemma put-get-vals-is-cpy (rewrite)
     (equal (put-vals (get-vals i2 lst2 n) i1 lst1 n)
            (lstcpy i1 lst1 i2 lst2 n))
     ((induct (lstcpy1 i1 lst1 i2 lst2 n))))

(disable lstcpy-cpy1)

;            MMOV1-LST, MMOV-LST1, MMOVN-LST AND MMOVN-LST1
(defn mmov1-lst (i lst1 lst2 n)
  (if (zerop n)
      lst1
    (mmov1-lst (add1 i) (put-nth (get-nth i lst2) i lst1) lst2 (sub1 n))))

; the same as mov1-lst.
(defn mmov1-lst1 (i lst1 lst2 n)
  (if (zerop n)
      lst1
    (mmov1-lst1 (sub1 i) (put-nth (get-nth (sub1 i) lst2) (sub1 i) lst1)
                lst2 (sub1 n))))

(defn movn-lst (n lst1 lst2 i)
  (put-vals (get-vals i lst2 n) i lst1 n))

(defn mmovn-lst (h lst1 lst2 i nt)
  (if (zerop nt)
      lst1
    (mmovn-lst h (movn-lst h lst1 lst2 i) lst2 (plus h i) (sub1 nt))))

(defn mmovn-lst1 (h lst1 lst2 i nt)
  (if (zerop nt)
      lst1
    (mmovn-lst1 h (movn-lst h lst1 lst2 (difference i h)) lst2
                (difference i h) (sub1 nt))))

(prove-lemma mmov1-lst-0 (rewrite)
     (equal (mmov1-lst i lst1 lst2 0) lst1))

(prove-lemma mmov1-lst1-0 (rewrite)
     (equal (mmov1-lst1 i lst1 lst2 0) lst1))

(prove-lemma mmovn-lst-0 (rewrite)
     (equal (mmovn-lst n lst1 lst2 i 0) lst1))

(prove-lemma mmovn-lst1-0 (rewrite)
     (equal (mmovn-lst1 n lst1 lst2 i 0) lst1))

;                     MOD32-EQ
(prove-lemma mod32-eq-deduction0 (rewrite)
     (mod32-eq x x))
```

```
(prove-lemma mod32-eq-deduction1 (rewrite)
     (and (equal (mod32-eq 0 (neg 32 x))
                 (mod32-eq x 0))
          (equal (mod32-eq (neg 32 x) 0)
                 (mod32-eq x 0)))
     ((use (sub-cancel0 (x 0) (y (head x 32)) (n 32)))))

(prove-lemma mod32-eq-deduction2 (rewrite)
     (and (equal (mod32-eq x (add 32 x y))
                 (mod32-eq 0 y))
          (equal (mod32-eq (add 32 x y) x)
                 (mod32-eq y 0)))
     ((use (add-cancel0 (x (head x 32)) (y (head y 32)) (n 32)))))

(prove-lemma mod32-eq-deduction3 (rewrite)
     (equal (mod32-eq (add 32 x y) (add 32 x z))
            (mod32-eq y z))
     ((use (add-cancel (y (head y 32)) (z (head z 32)) (n 32)))))

(disable mod32-eq)

; generate all the cases from between-ileq.
(prove-lemma between-ileq-la (rewrite)
     (implies (and (integerp x)
                   (integerp y)
                   (integerp z))
              (equal (between-ileq x y z)
                     (if (ilessp z x)
                         f
                       (or (equal x y)
                           (between-ileq (iplus x 1) y z)))))
     ((enable iplus integerp)))

(disable between-ileq)

; readm-mem, mem-lst, and mem-ilst is not changed if read-mem is not
; changed.
(prove-lemma read-mem-plus (rewrite)
     (equal (read-mem addr mem (plus m k))
            (app (times 8 k)
                 (read-mem (add 32 addr m) mem k)
                 (read-mem addr mem m)))
     ((induct (read-mem addr mem k))
      (enable add-plus)))

(prove-lemma stepn-readm-mem (rewrite)
     (implies (equal (read-mem x (mc-mem (stepn s n)) (times opsz k))
                     (read-mem x (mc-mem s) (times opsz k)))
              (equal (readm-mem opsz x (mc-mem (stepn s n)) k)
                     (readm-mem opsz x (mc-mem s) k)))
     ((disable stepn)))

(prove-lemma stepn-mem-lst (rewrite)
     (implies (equal (readm-mem opsz x (mc-mem (stepn s n)) k)
```

```
                    (readm-mem opsz x (mc-mem s) k))
               (equal (mem-lst opsz x (mc-mem (stepn s n)) k lst)
                      (mem-lst opsz x (mc-mem s) k lst)))
     ((disable stepn)))

(prove-lemma stepn-mem-ilst (rewrite)
     (implies (equal (readm-mem opsz x (mc-mem (stepn s n)) k)
                     (readm-mem opsz x (mc-mem s) k))
              (equal (mem-ilst opsz x (mc-mem (stepn s n)) k lst)
                     (mem-ilst opsz x (mc-mem s) k lst)))
     ((disable stepn)))

(disable read-mem-plus)

;                   DISABLE EVENTS
; many events are no longer useful upon the completion of this system.
; before we enter the verification phase, we simply disable them.

;;;;;;;; disable arithmetic events.
(disable plus-add1-1)
(disable remainder-exit)
(disable quotient-exit)

;;;;;;; disable definitions in the specification.
; operations:  The semantics of these operations has been established as
; rewrite rules in the library, which will be triggered to apply when
; nat-to-uint or nat-to-int are presented.
(disable mulu)
(disable muls)
(disable quot)
(disable rem)
(disable iquot)
(disable irem)
(disable lsl)
(disable lsr)
(disable asl)
(disable asr)

; condition codes:  To avoid to open up these flag definitions is one of the
; several efforts to keep the proving space managable.  Another point is
; that one flag might have different semantics in different situations.
(disable fix-bit)
(disable add-v)
(disable add-z)
(disable add-n)
(disable addx-v)
(disable addx-z)
(disable addx-n)
(disable sub-v)
(disable sub-n)
(disable subx-v)
(disable subx-z)
(disable subx-n)
(disable and-z)
```

```
(disable and-n)
(disable mulu-v)
(disable mulu-z)
(disable mulu-n)
(disable muls-v)
(disable muls-z)
(disable muls-n)
(disable or-z)
(disable or-n)
(disable divs-z)
(disable divs-n)
(disable divu-z)
(disable divu-n)
(disable rol-c)
(disable rol-z)
(disable rol-n)
(disable ror-c)
(disable ror-z)
(disable ror-n)
(disable lsl-c)
(disable lsl-z)
(disable lsl-n)
(disable lsr-c)
(disable lsr-z)
(disable lsr-n)
(disable asl-c)
(disable asl-v)
(disable asl-z)
(disable asl-n)
(disable asr-c)
(disable asr-z)
(disable asr-n)
(disable roxl-c)
(disable roxl-z)
(disable roxl-n)
(disable roxr-c)
(disable roxr-z)
(disable roxr-n)
(disable move-z)
(disable move-n)
(disable ext-z)
(disable ext-n)
(disable swap-z)
(disable swap-n)
(disable not-z)
(disable not-n)
(disable eor-z)
(disable eor-n)

; read/write defns:  for each of them, we have had a set of rewrite rules.
; their definitions are not needed to open up.
(disable pc-read-memp)
(disable pc-read-mem)
(disable read-memp)
```

```
(disable read-mem)
(disable write-memp)
(disable write-mem)
(disable readm-rn)
(disable writem-rn)

;;;;;;;; disable intermediate lemmas for read/write events:  to prove
; some important theorems about read/write,  we proved many intermediate
; lemmas.  It is time to discard them.
(disable pc-read-memp-la0)
(disable pc-read-memp-la1)
(disable pc-read-memp-la2)
(disable pc-read-memp-la3)
(disable write-memp-la0)
(disable write-memp-la1)
(disable write-memp-la2)
(disable write-memp-la3)

;;;;;;;; disable intermediate lemmas for disjoint0/disjoint events.
(disable disjoint0)
(disable disjoint)
(disable disjoint0-la0)
(disable disjoint0-la1)
(disable disjoint0-la2)
(disable disjoint-la0)
(disable disjoint-la1)
(disable disjoint-la2)
(disable disjoint-la3)

;;;;;;;; disable intermediate lemmas for add/sub/neg events.
(disable add-non-numberp)
(disable add-commutativity)
(disable add-commutativity1)

;;;;;;;;  disable mapping functions.
(disable nat-to-uint)
(disable uint-to-nat)

;;;;;;;;  disable miscellaneous lemmas.
(disable stepn)

; addressing mode checking for each instruction.  Their arguments are
; always "concrete" values.  For all the instructions, the oplen and ins
; have to become known at the execution time.  Theoretically,  it would
; be nice to disable them.  But we do not want disabling to slow down
; the prover.  Therefore, we only disable them when we feel necessary.
(deftheory mode-guards
  (add-addr-modep1 add-addr-modep2 adda-addr-modep sub-addr-modep1
   sub-addr-modep2 suba-addr-modep and-addr-modep1 and-addr-modep2
   mul&div-addr-modep or-addr-modep1 or-addr-modep2 s&r-addr-modep
   move-addr-modep movea-addr-modep lea-addr-modep clr-addr-modep
   move-from-ccr-addr-modep negx-addr-modep neg-addr-modep
   move-to-ccr-addr-modep pea-addr-modep movem-rn-ea-addr-modep
   movem-ea-rn-addr-modep tst-addr-modep tas-addr-modep jmp-addr-modep
```

```
   jsr-addr-modep not-addr-modep scc-addr-modep addq-addr-modep
   subq-addr-modep cmp-addr-modep cmpa-addr-modep eor&eori-addr-modep
   bchg-addr-modep bclr-addr-modep bset-addr-modep btst-addr-modep
   ori-addr-modep andi-addr-modep subi-addr-modep addi-addr-modep
   cmpi-addr-modep))

; the classification of instructions, according to the opcode.
(deftheory groups
  (bit-group move-ins move-group misc-group scc-group bcc-group
   or-group sub-group cmp-group and-group add-group s&r-group))

;             INVARIANTS OF THE SPEC
(constrain h-invariant (rewrite)
   (and (equal (p x (write-mem value y mem m) k)
               (p x mem k))
        (implies (and (p x mem k) (write-memp y mem m))
                 (equal (h x (write-mem value y mem m) k)
                        (h x mem k))))
   ((p rom-addrp)
    (h rom-addrp)))

(prove-lemma addr-index2-mem (rewrite)
     (equal (mc-mem (car (addr-index2 pc addr indexwd s)))
            (mc-mem s))
     ((disable index-register bits ir-scaled)))

(prove-lemma immediate-mem (rewrite)
     (equal (mc-mem (car (immediate oplen pc s)))
            (mc-mem s)))

(prove-lemma effec-addr-mem (rewrite)
     (equal (mc-mem (car (effec-addr oplen mode rn s)))
            (mc-mem s))
     ((disable addr-index2 immediate)))

(prove-lemma mc-instate-mem (rewrite)
     (equal (mc-mem (car (mc-instate oplen ins s)))
            (mc-mem s))
     ((disable effec-addr)))

(prove-lemma mapping-h (rewrite)
     (implies (p x (mc-mem (car s&addr)) k)
              (equal (h x (mc-mem (mapping oplen v&cvznx s&addr)) k)
                     (h x (mc-mem (car s&addr)) k))))

(defn t3 (x y z) t)

; add-group.
(prove-lemma add-group-h (rewrite)
     (implies (p x (mc-mem s) k)
              (equal (h x (mc-mem (add-group opmode ins s)) k)
                     (h x (mc-mem s) k)))
     ((disable operand mapping mc-instate bits op-len pre-effect)
      (disable-theory mode-guards)))
```

```
; sub-group.
(prove-lemma sub-group-h (rewrite)
     (implies (p x (mc-mem s) k)
              (equal (h x (mc-mem (sub-group opmode ins s)) k)
                     (h x (mc-mem s) k)))
     ((disable mapping mc-instate bits op-len pre-effect)
      (disable-theory mode-guards)))

; and-group.
(prove-lemma and-group-h (rewrite)
     (implies (p x (mc-mem s) k)
              (equal (h x (mc-mem (and-group oplen ins s)) k)
                     (h x (mc-mem s) k)))
     ((disable operand mapping mc-instate)
      (disable-theory mode-guards)))

; or-group.
(prove-lemma or-group-h (rewrite)
     (implies (p x (mc-mem s) k)
              (equal (h x (mc-mem (or-group oplen ins s)) k)
                     (h x (mc-mem s) k)))
     ((disable operand mapping mc-instate)
      (disable-theory mode-guards)))

; scc-group.
(prove-lemma scc-group-h (rewrite)
     (implies (p x (mc-mem s) k)
              (equal (h x (mc-mem (scc-group ins s)) k)
                     (h x (mc-mem s) k)))
     ((disable operand mapping mc-instate branch-cc bits op-len)
      (disable-theory mode-guards)))

; cmp-group.
(prove-lemma cmp-group-h (rewrite)
     (implies (p x (mc-mem s) k)
              (equal (h x (mc-mem (cmp-group oplen ins s)) k)
                     (h x (mc-mem s) k)))
     ((disable operand mapping mc-instate bits post-effect)
      (disable-theory mode-guards)))

; bcc-group.
(prove-lemma bcc-group-h (rewrite)
     (implies (p x (mc-mem s) k)
              (equal (h x (mc-mem (bcc-group disp ins s)) k)
                     (h x (mc-mem s) k)))
     ((disable operand mc-instate mapping branch-cc bits)
      (disable-theory mode-guards)))

; bit-group.
(prove-lemma write-mem-maintain-movep-writep (rewrite)
     (equal (movep-writep addr (write-mem value x mem m) n)
            (movep-writep addr mem n))
     ((induct (movep-writep addr mem n))))
```

```
(prove-lemma movep-write-h (rewrite)
     (implies (and (p x mem k) (movep-writep addr mem n))
              (equal (h x (movep-write value addr mem n) k)
                     (h x mem k)))
     ((induct (movep-write value addr mem n))))

(prove-lemma bit-group-h (rewrite)
     (implies (p x (mc-mem s) k)
              (equal (h x (mc-mem (bit-group ins s)) k)
                     (h x (mc-mem s) k)))
     ((disable operand mc-instate mapping immediate op-len bits)
      (disable-theory mode-guards)))

; move-group.
(prove-lemma move-h (rewrite)
     (implies (p x (mc-mem s) k)
              (equal (h x (mc-mem (move-ins oplen ins s)) k)
                     (h x (mc-mem s) k)))
     ((disable operand mapping effec-addr mc-instate)
      (disable-theory mode-guards)))

(prove-lemma move-group-h (rewrite)
     (implies (p x (mc-mem s) k)
              (equal (h x (mc-mem (move-group oplen ins s)) k)
                     (h x (mc-mem s) k)))
     ((disable operand mapping mc-instate move-ins)
      (disable-theory mode-guards)))

; s&r-group.
(prove-lemma s&r-group-h (rewrite)
     (implies (p x (mc-mem s) k)
              (equal (h x (mc-mem (s&r-group ins s)) k)
                     (h x (mc-mem s) k)))
     ((disable operand mc-instate mapping op-len bits sr-cnt)
      (disable-theory mode-guards)))

; misc-group.
(prove-lemma writem-mem-h (rewrite)
     (implies (and (p x mem k)
                   (write-memp addr mem (times opsz (len vlst))))
              (equal (h x (writem-mem opsz vlst addr mem) k)
                     (h x mem k)))
     ((induct (writem-mem opsz vlst addr mem))
      (enable write-memp-la2 write-memp-la3)))

(prove-lemma movem-rnlst-len (rewrite)
     (equal (len (movem-rnlst mask i))
            (movem-len mask)))

(prove-lemma movem-pre-rnlst-len (rewrite)
     (equal (len (movem-pre-rnlst mask i lst))
            (plus (movem-len mask) (len lst))))
```

```
(prove-lemma misc-group-h (rewrite)
     (implies (p x (mc-mem s) k)
              (equal (h x (mc-mem (misc-group ins s)) k)
                     (h x (mc-mem s) k)))
     ((disable operand effec-addr mc-instate mapping op-len bits op-sz)
      (disable-theory mode-guards)))

(prove-lemma stepi-h (rewrite)
     (implies (p x (mc-mem s) k)
              (equal (h x (mc-mem (stepi s)) k)
                     (h x (mc-mem s) k)))
     ((disable-theory groups)))

(functionally-instantiate stepi-p (rewrite)
    (implies (t3 x (mc-mem s) k)
             (equal (p x (mc-mem (stepi s)) k)
                    (p x (mc-mem s) k)))
    stepi-h
    ((h p)
     (p t3)))

(prove-lemma stepn-h (rewrite)
     (implies (p x (mc-mem s) k)
              (equal (h x (mc-mem (stepn s n)) k)
                     (h x (mc-mem s) k)))
     ((disable stepi) (enable stepn)))

; the instantiations.
(functionally-instantiate stepn-pc-read-memp (rewrite)
    (implies (t3 x (mc-mem s) k)
             (equal (pc-read-memp x (mc-mem (stepn s n)) k)
                    (pc-read-memp x (mc-mem s) k)))
    stepn-h
    ((h pc-read-memp)
     (p t3)))

(functionally-instantiate stepn-read-memp (rewrite)
    (implies (t3 x (mc-mem s) k)
             (equal (read-memp x (mc-mem (stepn s n)) k)
                    (read-memp x (mc-mem s) k)))
    stepn-h
    ((h read-memp)
     (p t3)))

(functionally-instantiate stepn-write-memp (rewrite)
    (implies (t3 x (mc-mem s) k)
             (equal (write-memp x (mc-mem (stepn s n)) k)
                    (write-memp x (mc-mem s) k)))
    stepn-h
    ((h write-memp)
     (p t3)))

; after the execution of n arbitrary instructions, ROM is still ROM.
(prove-lemma stepn-rom-addrp (rewrite)
```

```
     (equal (rom-addrp x (mc-mem (stepn s n)) k)
            (rom-addrp x (mc-mem s) k))
     ((enable rom-addrp)))

; after the execution of n arbitrary instructions, RAM is still RAM.
(prove-lemma stepn-ram-addrp (rewrite)
     (equal (ram-addrp x (mc-mem (stepn s n)) k)
            (ram-addrp x (mc-mem s) k))
     ((enable ram-addrp)))

; after the execution of n arbitrary instructions, the contents of
; the memory are not modified if that portion of the memory is ROM.
(functionally-instantiate stepn-pc-read-mem (rewrite)
    (implies (rom-addrp x (mc-mem s) k)
             (equal (pc-read-mem x (mc-mem (stepn s n)) k)
                    (pc-read-mem x (mc-mem s) k)))
    stepn-h
    ((h pc-read-mem)
     (p rom-addrp)))

(prove-lemma stepn-read-mem (rewrite)
    (implies (rom-addrp x (mc-mem s) k)
             (equal (read-mem x (mc-mem (stepn s n)) k)
                    (read-mem x (mc-mem s) k)))
    ((enable read->pc-read-mem)))

; after the execution of n arbitrary instructions, the program segment
; maintains the same.  Since we always require programs in ROM.
(prove-lemma stepn-mcode-addrp (rewrite)
     (implies (rom-addrp x (mc-mem s) (len lst))
              (equal (mcode-addrp x (mc-mem (stepn s n)) lst)
                     (mcode-addrp x (mc-mem s) lst)))
     ((induct (mcode-addrp x mem lst))))

(disable splus)
(disable mcode-addrp)

; end of the proving phase. ***************

;---------------------------------------------------------------------
;                  VERIFICATION PHASE
;---------------------------------------------------------------------
; our goal is to verify programs at machine code level. Before we go to
; specific programs, we introduce some conventions of machine code program
; constructs.  This section provides some useful concepts for us to specify
; and verify machine code programs.
;
; look up as unsigned integers.
(defn uread-mem (x mem n)
  (nat-to-uint (read-mem x mem n)))

(defn uread-dn (oplen dn s)
  (nat-to-uint (read-dn oplen dn s)))
```

```
(defn uread-an (oplen an s)
  (nat-to-uint (read-an oplen an s)))

; look up as signed integers.
(defn iread-mem (x mem n)
  (nat-to-int (read-mem x mem n) (times 8 n)))

(defn iread-dn (oplen dn s)
  (nat-to-int (read-dn oplen dn s) oplen))

(defn iread-an (oplen an s)
  (nat-to-int (read-an oplen an s) oplen))

; the return address of subroutine call.
(defn rts-addr (s)
  (read-mem (read-an 32 7 s) (mc-mem s) 4))

(defn linked-rts-addr (s)
  (read-mem (add 32 (read-an 32 6 s) 4) (mc-mem s) 4))

; the saved A6 on the stack.
(defn linked-a6 (s)
  (read-mem (read-an 32 6 s) (mc-mem s) 4))

; for MOVEM instruction.  After the link, A6 points to some place
; on the stack.  s is the current machine state, opsz is the
; operation size of the MOVEM,  i is the offset relative to a6,
; and n is the number of registers moved.
(defn movem-saved (s opsz i n)
  (readm-mem opsz (sub 32 i (read-an 32 6 s)) (mc-mem s) n))

; when only one register is saved,  we use MOVE instead of MOVEM.
(defn rn-saved (s)
  (read-mem (sub 32 4 (read-an 32 6 s)) (mc-mem s) 4))

; We do not want to deal with nat-to-int and int-to-nat in the logic
; level.  In the logic level, the proof concerns purely the algorithmic
; correctness in the problem.  It is hoped to be machine independent.
(defn lst-int (n lst)
  (if (nlistp lst)
      lst
    (cons (nat-to-int (car lst) n)
          (lst-int n (cdr lst)))))

(prove-lemma get-lst-int (rewrite)
     (equal (get-nth i (lst-int n lst))
            (nat-to-int (get-nth i lst) n)))

(prove-lemma put-lst-int (rewrite)
     (equal (put-nth (nat-to-int value n) i (lst-int n lst))
            (lst-int n (put-nth value i lst))))

(defn lst-integerp (lst)
  (if (nlistp lst)
```

```
      t
    (and (integerp (car lst))
         (lst-integerp (cdr lst)))))

(prove-lemma get-lst-integerp (rewrite)
     (implies (lst-integerp lst)
              (integerp (get-nth n lst))))

(prove-lemma mem-lst-integerp (rewrite)
     (implies (and (mem-lst opsz a mem n lst)
                   (equal oplen (times 8 opsz)))
              (lst-integerp (lst-int oplen lst))))

(prove-lemma mem-ilst-lst-integerp (rewrite)
     (implies (mem-ilst opsz a mem n lst)
              (lst-integerp lst)))

(disable mem-lst)
(disable mem-ilst)

; a trick to get all the appearances of a term replaced by another
; term. For example, in subroutines, a6 is used as the addresss register
; for LINK. It is often relative to sp(a7).  We'd like to have a6
; replaced by a7.
(defn equal* (x y)
  (equal x y))

(prove-lemma equal*-reflex (rewrite)
     (equal* x x))

(prove-lemma read-rn-equal* (rewrite)
     (implies (equal* (read-rn oplen rn rfile) x)
              (equal (read-rn oplen rn rfile) x)))

(disable equal*)

; the registers d0, d1, a0, and a1 are available for any subroutines.
; There is no obligation for subroutines to restore their previous values.
; Therefore, we do not need to keep track of them.  We handle a7
; (the stack pointer) separately to make sure it is set correctly.
; a6 (the frame pointer) perhaps also deserves special treatment.
(defn d2-7a2-5p (rn)
  (and (not (zerop rn))
       (not (equal rn 1))
       (not (equal rn 8))
       (not (equal rn 9))
       (not (equal rn 14))
       (not (equal rn 15))))

; d2 will be used for some purpose.
(defn d3-7a2-5p (rn)
  (and (d2-7a2-5p rn)
       (not (equal rn 2))))
```

```
; d2 and d3 will be used for some purpose.
(defn d4-7a2-5p (rn)
  (and (d2-7a2-5p rn)
       (not (equal rn 2))
       (not (equal rn 3))))

; d2, d3, and d4 will be used for some purpose.
(defn d5-7a2-5p (rn)
  (and (d2-7a2-5p rn)
       (not (equal rn 2))
       (not (equal rn 3))
       (not (equal rn 4))))

; a2 will be used for some purpose.
(defn d2-7a3-5p (rn)
  (and (d2-7a2-5p rn)
       (not (equal rn 10))))

(defn d4-7a4-5p (rn)
  (and (d4-7a2-5p rn)
       (not (equal rn 10))
       (not (equal rn 11))))

(defn d4-7a5p (rn)
  (and (d4-7a2-5p rn)
       (not (equal rn 10))
       (not (equal rn 11))
       (not (equal rn 12))))

(defn d5-7a4-5p (rn)
  (and (d4-7a2-5p rn)
       (not (equal rn 4))
       (not (equal rn 10))
       (not (equal rn 11))))

(defn d6-7a2-5p (rn)
  (and (d4-7a2-5p rn)
       (not (equal rn 4))
       (not (equal rn 5))))

; something I don't know where to put yet.
; the mean (i+j)/2 < j iff i < j.
(prove-lemma mean-lessp-lemma (rewrite)
        (equal (lessp (plus i j) (times 2 j))
               (lessp i j))
        ((induct (difference j i))))

(prove-lemma mean-lessp (rewrite)
     (and (equal (lessp (quotient (plus i j) 2) j)
                 (lessp i j))
          (equal (lessp (quotient (plus j i) 2) j)
                 (lessp i j))))

; swap the ith and jth elements of the list.
```

```
(defn swap (i j lst)
  (put-nth (get-nth i lst) j (put-nth (get-nth j lst) i lst)))

(prove-lemma get-swap (rewrite)
     (equal (get-nth k (swap i j lst))
            (if (equal (fix k) (fix i))
                (get-nth j lst)
              (if (equal (fix k) (fix j))
                  (get-nth i lst)
                (get-nth k lst)))))

(disable get-nth)
(disable put-nth)

; C conventions.
(defn null () 0)

(defn eof () -1)

; the Nqthm counterparts of the Berkeley C string functions.
; memchr.
(defn memchr1 (i n lst ch)
  (if (equal (get-nth i lst) ch)
      (fix i)
    (if (equal (sub1 n) 0)
        f
      (memchr1 (add1 i) (sub1 n) lst ch))))

(defn memchr (n lst ch)
  (if (equal n 0) f (memchr1 0 n lst ch)))

; memcmp.
(defn memcmp1 (i n lst1 lst2)
  (if (equal (get-nth i lst1) (get-nth i lst2))
      (if (equal (sub1 n) 0)
          0
        (memcmp1 (add1 i) (sub1 n) lst1 lst2))
    (idifference (get-nth i lst1) (get-nth i lst2))))

(defn memcmp (n lst1 lst2)
  (if (equal n 0) 0 (memcmp1 0 n lst1 lst2)))

; memset.
(defn memset1 (i n lst ch)
  (if (equal (sub1 n) 0)
      (put-nth ch i lst)
    (memset1 (add1 i) (sub1 n) (put-nth ch i lst) ch)))

(defn memset (n lst ch)
  (if (equal n 0) lst (memset1 0 n lst ch)))

; strlen.
(defn strlen (i n lst)
  (if (lessp i n)
```

```
      (if (equal (get-nth i lst) (null))
          i
        (strlen (add1 i) n lst))
    i)
  ((lessp (difference n i))))

; strcpy.
(defn strcpy (i lst1 n2 lst2)
  (if (lessp i n2)
      (if (equal (get-nth i lst2) (null))
          (put-nth (null) i lst1)
        (strcpy (add1 i) (put-nth (get-nth i lst2) i lst1) n2 lst2))
    lst1)
  ((lessp (difference n2 i))))

; strcmp.
(defn strcmp (i n1 lst1 lst2)
  (if (lessp i n1)
      (if (equal (get-nth i lst1) (get-nth i lst2))
          (if (equal (get-nth i lst1) (null))
              0
            (strcmp (add1 i) n1 lst1 lst2))
        (idifference (get-nth i lst1) (get-nth i lst2)))
    0)
  ((lessp (difference n1 i))))

; strcoll.
(defn strcoll (n1 lst1 lst2)
  (strcmp 0 n1 lst1 lst2))

; strcat.
(defn strcpy1 (i lst1 j n2 lst2)
  (if (lessp j n2)
      (if (equal (get-nth j lst2) (null))
          (put-nth (null) i lst1)
        (strcpy1 (add1 i) (put-nth (get-nth j lst2) i lst1) (add1 j) n2 lst2))
    lst1)
  ((lessp (difference n2 j))))

(defn strcat (n1 lst1 n2 lst2)
  (if (equal (get-nth 0 lst1) (null))
      (strcpy1 0 lst1 0 n2 lst2)
    (strcpy1 (strlen 1 n1 lst1) lst1 0 n2 lst2)))

; strncat.
(defn strcpy2 (i lst1 j n lst2)
  (if (equal (get-nth j lst2) (null))
      (put-nth (null) i lst1)
    (if (equal (sub1 n) 0)
        (put-nth (null) (add1 i) (put-nth (get-nth j lst2) i lst1))
      (strcpy2 (add1 i) (put-nth (get-nth j lst2) i lst1) (add1 j)
               (sub1 n) lst2))))

(defn strncat (n1 lst1 n lst2)
```

```
  (if (equal n 0)
      lst1
    (if (equal (get-nth 0 lst1) (null))
        (strcpy2 0 lst1 0 n lst2)
      (strcpy2 (strlen 1 n1 lst1) lst1 0 n lst2))))

; strncmp.
(defn strncmp1 (i n lst1 lst2)
  (if (equal (get-nth i lst1) (get-nth i lst2))
      (if (equal (get-nth i lst1) 0)
          0
        (if (equal (sub1 n) 0)
            0
          (strncmp1 (add1 i) (sub1 n) lst1 lst2)))
    (idifference (get-nth i lst1) (get-nth i lst2))))

(defn strncmp (n lst1 lst2)
  (if (zerop n) 0 (strncmp1 0 n lst1 lst2)))

; strncpy.
(defn zero-list1 (i n lst)
  (if (equal (sub1 n) 0)
      lst
    (zero-list1 (add1 i) (sub1 n) (put-nth 0 i lst))))

(defn zero-list (i n lst)
  (zero-list1 (add1 i) n (put-nth 0 i lst)))

(defn strncpy1 (i n lst1 lst2)
  (if (equal (get-nth i lst2) 0)
      (zero-list i n lst1)
    (if (equal (sub1 n) 0)
        (put-nth (get-nth i lst2) i lst1)
      (strncpy1 (add1 i) (sub1 n) (put-nth (get-nth i lst2) i lst1) lst2))))

(defn strncpy (n lst1 lst2)
  (if (zerop n)
      lst1
    (strncpy1 0 n lst1 lst2)))

; strchr.
(defn strchr (i n lst ch)
  (if (lessp i n)
      (if (equal (get-nth i lst) ch)
          (fix i)
        (if (equal (get-nth i lst) 0)
            f
          (strchr (add1 i) n lst ch)))
    f)
  ((lessp (difference n i))))

; strcspn.
(defn strcspn (i1 n1 lst1 n2 lst2)
  (if (lessp i1 n1)
```

```
      (if (strchr 0 n2 lst2 (get-nth i1 lst1))
          (fix i1)
        (strcspn (add1 i1) n1 lst1 n2 lst2))
    f)
  ((lessp (difference n1 i1))))

; strrchr.
(defn strrchr (i n lst ch j)
  (if (lessp i n)
      (if (equal (get-nth i lst) ch)
          (if (equal (get-nth i lst) 0)
              (fix i)
            (strrchr (add1 i) n lst ch (fix i)))
        (if (equal (get-nth i lst) 0)
            j
          (strrchr (add1 i) n lst ch j)))
    j)
  ((lessp (difference n i))))

; strspn.
(defn strchr1 (i n lst ch)
  (if (lessp i n)
      (if (equal (get-nth i lst) 0)
          f
        (if (equal (get-nth i lst) ch)
            (fix i)
          (strchr1 (add1 i) n lst ch)))
    f)
  ((lessp (difference n i))))

(prove-lemma strchr1-bounds (rewrite)
     (and (not (lessp n (strchr1 i n lst ch)))
          (implies (strchr1 i n lst ch)
                   (not (lessp (strchr1 i n lst ch) i)))))

(prove-lemma strchr1-false-0 (rewrite)
    (not (strchr1 i n lst 0)))

(defn strspn (i1 n1 lst1 n2 lst2)
  (if (lessp i1 n1)
      (if (strchr1 0 n2 lst2 (get-nth i1 lst1))
          (strspn (add1 i1) n1 lst1 n2 lst2)
        (fix i1))
    f)
  ((lessp (difference n1 i1))))

; strpbrk.
(defn strpbrk (i1 n1 lst1 n2 lst2)
  (if (lessp i1 n1)
      (if (equal (get-nth i1 lst1) 0)
          f
        (if (strchr1 0 n2 lst2 (get-nth i1 lst1))
            (fix i1)
          (strpbrk (add1 i1) n1 lst1 n2 lst2)))
```

```
    f)
  ((lessp (difference n1 i1))))

; strstr.
(defn strstr1 (i n1 lst1 n2 lst2 len)
  (if (lessp i n1)
      (let ((j (strchr1 i n1 lst1 (get-nth 0 lst2))))
        (if (numberp j)
            (if (equal (strncmp len (mcdr (add1 j) lst1) (mcdr 1 lst2)) 0)
                j
              (strstr1 (add1 j) n1 lst1 n2 lst2 len))
          f))
    f)
  ((lessp (difference n1 i))))

(defn strstr (n1 lst1 n2 lst2)
  (if (equal (get-nth 0 lst2) 0)
      0
    (strstr1 0 n1 lst1 n2 lst2 (strlen 0 (sub1 n2) (mcdr 1 lst2)))))

; memmove.
(defn memmove-1 (lst1 lst2 i nt n)
  (if (equal (remainder n 4) 0)
      (mmovn-lst 4 lst1 lst2 i nt)
    (mmov1-lst (plus i (times 4 nt)) (mmovn-lst 4 lst1 lst2 i nt)
               lst2 (remainder n 4))))

(defn memmove-0 (lst1 lst2 i nt n)
  (if (lessp n 4)
      (mmov1-lst (plus i nt) (mmov1-lst i lst1 lst2 nt) lst2 n)
    (memmove-1 (mmov1-lst i lst1 lst2 nt) lst2 (plus i nt) (quotient n 4) n)))

(defn memmove-4 (lst1 lst2 i nt n)
  (if (equal (remainder n 4) 0)
      (mmovn-lst1 4 lst1 lst2 i nt)
    (mmov1-lst1 (difference i (times 4 nt)) (mmovn-lst1 4 lst1 lst2 i nt)
                lst2 (remainder n 4))))

(defn memmove-3 (lst1 lst2 i nt n)
  (if (lessp n 4)
      (mmov1-lst1 (difference i nt) (mmov1-lst1 i lst1 lst2 nt) lst2 n)
    (memmove-4 (mmov1-lst1 i lst1 lst2 nt) lst2 (difference i nt)
               (quotient n 4) n)))

(defn memmove (str1 str2 n lst1 lst2)
  (if (zerop n)
      lst1
    (if (equal (nat-to-uint str1) (nat-to-uint str2))
        lst2
      (let ((x1 (nat-to-uint str1))
            (x2 (nat-to-uint str2)))
        (if (lessp x1 x2)
            (if (and (equal (remainder x1 4) 0)
                     (equal (remainder x2 4) 0))
```

```
                (if (lessp n 4)
                    (mmov1-lst 0 lst1 lst2 n)
                  (memmove-1 lst1 lst2 0 (quotient n 4) n))
              (if (and (equal (remainder x1 4) (remainder x2 4))
                       (lessp 3 n))
                  (memmove-0 lst1 lst2 0 (difference 4 (remainder x1 4))
                             (difference (plus n (remainder x1 4)) 4))
                (memmove-0 lst1 lst2 0 n 0)))
            (let ((y1 (plus n x1))
                  (y2 (plus n x2)))
              (if (and (equal (remainder y1 4) 0)
                       (equal (remainder y2 4) 0))
                  (if (lessp n 4)
                      (mmov1-lst1 n lst1 lst2 n)
                    (memmove-4 lst1 lst2 n (quotient n 4) n))
                (if (and (equal (remainder y1 4) (remainder y2 4))
                         (lessp 4 n))
                    (memmove-3 lst1 lst2 n (remainder y1 4)
                               (difference n (remainder y1 4)))
                  (memmove-3 lst1 lst2 n n 0)))))))))

; dual events of string functions.  They are for internal use only and
; hardly visible for users.
(defn strlen* (i* i n lst)
  (if (lessp i n)
      (if (equal (get-nth i lst) (null))
          i*
        (strlen* (add 32 i* 1) (add1 i) n lst))
    i*)
  ((lessp (difference n i))))

(defn memchr* (i* i n lst ch)
  (if (equal (get-nth i lst) ch)
      i*
    (if (equal (sub1 n) 0)
        f
      (memchr* (add 32 i* 1) (add1 i) (sub1 n) lst ch))))

(defn strchr* (i* i n lst ch)
  (if (lessp i n)
      (if (equal (get-nth i lst) ch)
          i*
        (if (equal (get-nth i lst) 0)
            f
          (strchr* (add 32 i* 1) (add1 i) n lst ch)))
    f)
  ((lessp (difference n i))))

(defn strchr1* (i* i n lst ch)
  (if (lessp i n)
      (if (equal (get-nth i lst) 0)
          f
        (if (equal (get-nth i lst) ch)
            i*
```

```
          (strchr1* (add 32 i* 1) (add1 i) n lst ch)))
    f)
  ((lessp (difference n i))))

(defn strcspn* (i1* i1 n1 lst1 n2 lst2)
  (if (lessp i1 n1)
      (if (strchr 0 n2 lst2 (get-nth i1 lst1))
          i1*
        (strcspn* (add 32 i1* 1) (add1 i1) n1 lst1 n2 lst2))
    f)
  ((lessp (difference n1 i1))))

(defn strrchr* (i* i n lst ch j*)
  (if (lessp i n)
      (if (equal (get-nth i lst) ch)
          (if (equal (get-nth i lst) 0)
              i*
            (strrchr* (add 32 i* 1) (add1 i) n lst ch i*))
        (if (equal (get-nth i lst) 0)
            j*
          (strrchr* (add 32 i* 1) (add1 i) n lst ch j*)))
    j*)
  ((lessp (difference n i))))

(defn strpbrk* (i1* i1 n1 lst1 n2 lst2)
  (if (lessp i1 n1)
      (if (equal (get-nth i1 lst1) 0)
          f
        (if (strchr1 0 n2 lst2 (get-nth i1 lst1))
            i1*
          (strpbrk* (add 32 i1* 1) (add1 i1) n1 lst1 n2 lst2)))
    f)
  ((lessp (difference n1 i1))))

(defn strspn* (i1* i1 n1 lst1 n2 lst2)
  (if (lessp i1 n1)
      (if (strchr1 0 n2 lst2 (get-nth i1 lst1))
          (strspn* (add 32 i1* 1) (add1 i1) n1 lst1 n2 lst2)
        i1*)
    f)
  ((lessp (difference n1 i1))))

(defn strstr1* (i* i n1 lst1 n2 lst2 len)
  (if (lessp i n1)
      (let ((j* (strchr1* i* i n1 lst1 (get-nth 0 lst2)))
            (j (strchr1 i n1 lst1 (get-nth 0 lst2))))
        (if (numberp j)
            (if (equal (strncmp len (mcdr (add1 j) lst1) (mcdr 1 lst2)) 0)
                j*
              (strstr1* (add 32 j* 1) (add1 j) n1 lst1 n2 lst2 len))
          f))
    f)
  ((lessp (difference n1 i))))
```

```
(defn strstr* (n1 lst1 n2 lst2)
  (if (equal (get-nth 0 lst2) 0)
      0
    (strstr1* 0 0 n1 lst1 n2 lst2 (strlen 0 (sub1 n2) (mcdr 1 lst2)))))

; strtok.
; the location of the token.
(defn strtok-tok (str1 last n1 lst1 n2 lst2)
  (let ((i* (strspn* 0 0 n1 lst1 n2 lst2))
        (i (strspn 0 n1 lst1 n2 lst2)))
    (if (equal (nat-to-uint str1) 0)
        (if (equal (nat-to-uint last) 0)
            0
          (if (equal (get-nth i lst1) 0)
              0
            (add 32 last i*)))
      (if (equal (get-nth i lst1) 0)
          0
        (add 32 str1 i*)))))

; the new lst1.
(defn strtok-lst0 (i1 lst1 ch)
  (if (equal ch 0) lst1 (put-nth 0 i1 lst1)))

(defn strtok-lst1 (i1 n1 lst1 n2 lst2)
  (let ((i (strcspn i1 n1 lst1 n2 lst2)))
    (strtok-lst0 i lst1 (get-nth i lst1))))

(defn strtok-lst2 (i1 n1 lst1 n2 lst2)
  (if (equal (get-nth i1 lst1) 0)
      lst1
    (strtok-lst1 (add1 i1) n1 lst1 n2 lst2)))

(defn strtok-lst (n1 lst1 n2 lst2)
  (strtok-lst2 (strspn 0 n1 lst1 n2 lst2) n1 lst1 n2 lst2))

; the new value of the static variable.
(defn strtok-last0 (str1 i1* i1 n1 lst1 n2 lst2)
  (let ((i* (strcspn* i1* i1 n1 lst1 n2 lst2))
        (i (strcspn i1 n1 lst1 n2 lst2)))
    (if (equal (get-nth i lst1) 0)
        0
      (add 32 str1 (add 32 i* 1)))))

(defn strtok-last1 (str1 i1* i1 n1 lst1 n2 lst2)
  (if (equal (get-nth i1 lst1) 0)
      0
    (strtok-last0 str1 (add 32 i1* 1) (add1 i1) n1 lst1 n2 lst2)))

(defn strtok-last (str1 last n1 lst1 n2 lst2)
  (let ((i* (strspn* 0 0 n1 lst1 n2 lst2))
        (i (strspn 0 n1 lst1 n2 lst2)))
    (if (equal (nat-to-uint str1) 0)
        (if (equal (nat-to-uint last) 0)
```

```
           last
          (strtok-last1 last i* i n1 lst1 n2 lst2))
      (strtok-last1 str1 i* i n1 lst1 n2 lst2))))

; strxfrm.
(defn strxfrm1 (i lst1 lst2 n)
  (if (equal (get-nth i lst2) 0)
      (put-nth 0 i lst1)
    (if (equal (sub1 n) 0)
        (put-nth 0 i lst1)
      (strxfrm1 (add1 i) (put-nth (get-nth i lst2) i lst1) lst2 (sub1 n)))))

(defn strxfrm (lst1 lst2 n)
  (if (zerop n)
      lst1
    (strxfrm1 0 lst1 lst2 n)))

; a list of characters.
(defn lst-of-chrp (lst)
  (if (listp lst)
      (and (numberp (car lst))
           (lessp (car lst) 256)
           (lst-of-chrp (cdr lst)))
    t))

; theorems about lst-of-chrp.
(prove-lemma get-lst-of-chrp (rewrite)
     (implies (lst-of-chrp lst)
              (and (lessp (get-nth i lst) 256)
                   (numberp (get-nth i lst))))
     ((enable get-nth)))

(prove-lemma put-lst-of-chrp (rewrite)
     (implies (lst-of-chrp lst)
              (equal (lst-of-chrp (put-nth x i lst))
                     (and (numberp x) (lessp x 256))))
     ((enable put-nth)))

#|
(prove-lemma lessp-read-mem-1 (rewrite)
     (lessp (read-mem x mem 1) 256)
     ((use (byte-read-nat-rangep (n 8)))
      (enable nat-rangep)))

(prove-lemma mem-lst-of-chrp (rewrite)
     (implies (mem-lst 1 x mem n lst)
              (lst-of-chrp lst))
     ((enable mem-lst)))
|#

(disable lst-of-chrp)

; the predicate stringp.
(defn slen (i n lst)
```

```
  (if (lessp i n)
      (if (equal (get-nth i lst) (null))
          (fix i)
        (slen (add1 i) n lst))
    (fix i))
  ((lessp (difference n i))))

(defn stringp (i n lst)
  (lessp (slen i n lst) n))

; events for slen, which is part of stringp.
(prove-lemma slen-ubound (rewrite)
     (implies (leq i n)
              (not (lessp n (slen i n lst)))))

(prove-lemma slen-lbound (rewrite)
     (not (lessp (slen i n lst) i)))

(prove-lemma slen-01 (rewrite)
     (and (equal (slen i 0 lst) (fix i))
          (equal (slen i 1 lst)
                 (if (and (lessp i 1)
                          (not (equal (get-nth 0 lst) 0)))
                     1 (fix i))))
     ((enable get-nth-0)))

(prove-lemma slen-add1 (rewrite)
     (equal (slen i (add1 i) lst)
            (if (equal (get-nth i lst) 0) (fix i) (add1 i))))

(prove-lemma slen-put0 (rewrite)
     (equal (slen i n (put-nth 0 i lst))
            (fix i)))

(prove-lemma slen-put (rewrite)
     (implies (lessp j i)
              (equal (slen i n (put-nth v j lst))
                     (slen i n lst))))

(prove-lemma lessp-slen-mcdr (rewrite)
     (implies (lessp (slen (plus i k) n1 lst1) n1)
              (equal (lessp (slen i n (mcdr k lst1)) (difference n1 k)) t))
     ((enable plus-add1-1)))

(prove-lemma lessp-slen-mcdr-0 (rewrite)
     (implies (lessp (slen (add1 i) n1 lst1) n1)
              (equal (lessp (slen i n (mcdr 1 lst1)) (sub1 n1)) t))
     ((use (lessp-slen-mcdr (k 1)))))

(prove-lemma slen-rec (rewrite)
     (implies (and (not (equal (get-nth i lst) 0))
                   (lessp i n))
              (equal (slen (add1 i) n lst)
                     (slen i n lst))))
```

```
(disable slen)

; theorems about stringp.
(prove-lemma stringp-la (rewrite)
     (implies (and (stringp i n lst)
                   (int-rangep n nn)
                   (not (equal (get-nth i lst) 0)))
              (int-rangep (add1 i) nn))
     ((enable int-rangep integerp)))

; theorems about strchr.
(prove-lemma strchr-bounds (rewrite)
     (and (not (lessp n (strchr i n lst ch)))
          (implies (strchr i n lst ch)
                   (not (lessp (strchr i n lst ch) i)))))

(prove-lemma memchr-bounds (rewrite)
     (and (not (lessp (plus i n) (memchr1 i n lst ch)))
          (implies (memchr1 i n lst ch)
                   (not (lessp (memchr1 i n lst ch) i))))
     ((induct (memchr1 i n lst ch))))

(prove-lemma strpbrk-bounds (rewrite)
     (and (not (lessp n1 (strpbrk i1 n1 lst1 n2 lst2)))
          (implies (strpbrk i1 n1 lst1 n2 lst2)
                   (not (lessp (strpbrk i1 n1 lst1 n2 lst2) i1)))))

(prove-lemma strrchr-bounds (rewrite)
     (implies (leq j n)
              (not (lessp n (strrchr i n lst ch j)))))

(prove-lemma strstr-bounds (rewrite)
     (and (not (lessp n1 (strstr1 i n1 lst1 n2 lst2 len)))
          (implies (strstr1 i n1 lst1 n2 lst2 len)
                   (not (lessp (strstr1 i n1 lst1 n2 lst2 len) i)))))

(prove-lemma strspn-ubound (rewrite)
     (implies (not (zerop n1))
              (lessp (strspn i n1 lst1 n2 lst2) n1)))

(prove-lemma strspn-bounds (rewrite)
     (and (not (lessp n1 (strspn i n1 lst1 n2 lst2)))
          (implies (strspn i n1 lst1 n2 lst2)
                   (not (lessp (strspn i n1 lst1 n2 lst2) i)))))

(prove-lemma strcspn-bounds (rewrite)
     (and (not (lessp n1 (strcspn i n1 lst1 n2 lst2)))
          (implies (strcspn i n1 lst1 n2 lst2)
                   (not (lessp (strcspn i n1 lst1 n2 lst2) i)))))

; a lemma to establish nat-rangep.
(prove-lemma nat-rangep-la (rewrite)
     (implies (lessp (nat-to-uint x) (exp 2 n))
```

```
              (nat-rangep x n))
     ((enable nat-rangep nat-to-uint)))

(disable nat-rangep-la)

; some useful lemmas.  I do not know where to put them yet.
; I will put them in the right place.
(prove-lemma disjoint-1-int (rewrite)
     (implies (and (disjoint a m b n)
                   (leq j m)
                   (leq (plus (nat-to-int k 32) l) n)
                   (numberp (nat-to-int k 32)))
              (disjoint a j (add 32 b k) l))
     ((enable nat-to-int)))

(prove-lemma disjoint-2-int (rewrite)
     (implies (and (disjoint a m b n)
                   (leq (plus (nat-to-int i 32) j) m)
                   (leq l n)
                   (numberp (nat-to-int i 32)))
              (disjoint (add 32 a i) j b l))
     ((enable nat-to-int)))

(prove-lemma disjoint-2~-int (rewrite)
     (implies (and (disjoint a m b n)
                   (leq (plus (nat-to-int i 32) j) m)
                   (leq l n)
                   (numberp (nat-to-int i 32)))
              (disjoint b l (add 32 a i) j))
     ((enable nat-to-int)))

(prove-lemma disjoint-3-int (rewrite)
     (implies (and (disjoint a m b n)
                   (leq (plus (nat-to-int i 32) j) m)
                   (leq (plus (nat-to-int k 32) l) n)
                   (numberp (nat-to-int i 32))
                   (numberp (nat-to-int k 32)))
              (disjoint (add 32 a i) j (add 32 b k) l))
     ((enable nat-to-int)))

(prove-lemma read-memp-ram1-uint (rewrite)
     (implies (and (ram-addrp addr mem k)
                   (leq (plus (nat-to-uint i) j) k))
              (read-memp (add 32 addr i) mem j))
     ((enable nat-to-uint)))

(prove-lemma write-memp-ram1-uint (rewrite)
     (implies (and (ram-addrp addr mem k)
                   (leq (plus (nat-to-uint i) j) k))
              (write-memp (add 32 addr i) mem j))
     ((enable nat-to-uint)))

(prove-lemma disjoint-1-uint (rewrite)
     (implies (and (disjoint a m b n)
```

```
                (leq j m)
                (leq (plus (nat-to-uint k) l) n))
           (disjoint a j (add 32 b k) l))
     ((enable nat-to-uint)))

(prove-lemma disjoint-2-uint (rewrite)
     (implies (and (disjoint a m b n)
                (leq (plus (nat-to-uint i) j) m)
                (leq l n))
           (disjoint (add 32 a i) j b l))
     ((enable nat-to-uint)))

(prove-lemma disjoint-2~-uint (rewrite)
     (implies (and (disjoint a m b n)
                (leq (plus (nat-to-uint i) j) m)
                (leq l n))
           (disjoint b l (add 32 a i) j))
     ((enable nat-to-uint)))

(prove-lemma disjoint-3-uint (rewrite)
     (implies (and (disjoint a m b n)
                (leq (plus (nat-to-uint i) j) m)
                (leq (plus (nat-to-uint k) l) n))
           (disjoint (add 32 a i) j (add 32 b k) l))
     ((enable nat-to-uint)))

(prove-lemma disjoint-5-uint (rewrite)
     (implies (and (disjoint (add 32 a i) m b n)
                (leq (plus j (index-n 0 i)) m)
                (leq (plus (nat-to-uint k) l) n))
           (disjoint a j (add 32 b k) l))
     ((enable nat-to-uint)))

(prove-lemma disjoint-6-uint (rewrite)
     (implies (and (disjoint (add 32 a i) m b n)
                (leq (plus j (index-n i1 i)) m)
                (leq l n))
           (disjoint (add 32 a i1) j b l))
     ((enable nat-to-uint)))

(prove-lemma disjoint-7-uint (rewrite)
     (implies (and (disjoint (add 32 a i) m b n)
                (leq (plus j (index-n i1 i)) m)
                (leq (plus (nat-to-uint k) l) n))
           (disjoint (add 32 a i1) j (add 32 b k) l))
     ((enable nat-to-uint)))

(prove-lemma read-rn-int-8_32 (rewrite)
     (equal (nat-to-int (read-rn 8 rn rfile) 32)
            (nat-to-uint (read-rn 8 rn rfile)))
     ((enable read-rn nat-to-int nat-to-uint)))

(prove-lemma read-rn-int-16_32 (rewrite)
     (equal (nat-to-int (read-rn 16 rn rfile) 32)
```

```
            (nat-to-uint (read-rn 16 rn rfile)))
     ((enable read-rn nat-to-int nat-to-uint)))

(prove-lemma read-rn-int-8_16 (rewrite)
     (equal (nat-to-int (read-rn 8 rn rfile) 16)
            (nat-to-uint (read-rn 8 rn rfile)))
     ((enable read-rn nat-to-int nat-to-uint)))

(prove-lemma read-mem-int-8_32 (rewrite)
     (equal (nat-to-int (read-mem x mem 1) 32)
            (nat-to-uint (read-mem x mem 1)))
     ((use (read-mem-nat-rangep (n 31) (k 1)))
      (enable nat-to-int nat-to-uint nat-rangep)))

(prove-lemma read-mem-int-16_32 (rewrite)
     (equal (nat-to-int (read-mem x mem 2) 32)
            (nat-to-uint (read-mem x mem 2)))
     ((use (read-mem-nat-rangep (n 31) (k 2)))
      (enable nat-to-int nat-to-uint nat-rangep)))

(prove-lemma read-mem-int-8_16 (rewrite)
     (equal (nat-to-int (read-mem x mem 1) 16)
            (nat-to-uint (read-mem x mem 1)))
     ((use (read-mem-nat-rangep (n 15) (k 1)))
      (enable nat-to-int nat-to-uint nat-rangep)))

(prove-lemma idifference-int-rangep (rewrite)
     (implies (and (numberp x)
                   (numberp y)
                   (int-rangep x n)
                   (int-rangep y n))
              (int-rangep (idifference x y) n))
     ((enable int-rangep iplus idifference)))

(disable idifference)

; some more arithmetic.
(prove-lemma difference-cancel-1 (rewrite)
     (implies (leq i j)
              (and (equal (difference (plus i j) (times 2 i))
                          (difference j i))
                   (equal (difference (times 2 j) (plus i j))
                          (difference j i))))
     ((enable times)))

(prove-lemma difference-is-1 (rewrite)
     (equal (equal (difference x y) 1)
            (equal x (add1 y))))

(prove-lemma mean-difference-1 (rewrite)
     (implies (leq i j)
              (equal (difference (quotient (plus i j) 2) i)
                     (quotient (difference j i) 2)))
     ((use (quotient-difference (x (plus i j)) (y (times 2 i)) (z 2)))))
```

```
(prove-lemma mean-difference-2 (rewrite)
     (implies (leq i j)
              (not (lessp (quotient (difference j i) 2)
                          (difference (sub1 j) (quotient (plus i j) 2)))))
     ((use (quotient-diff (x (times 2 j)) (y (plus i j)) (z 2)))
      (disable quotient-times-lessp quotient remainder)))

(prove-lemma plus-0 (rewrite)
     (equal (plus 0 x) (fix x)))

(prove-lemma plus-times-sub1 (rewrite)
     (equal (plus x (times x (sub1 y)))
            (if (zerop y) (fix x) (times x y))))

(disable plus-times-sub1)

(prove-lemma plus2-times-sub1 (rewrite)
     (equal (plus x y (times x (sub1 z)))
            (if (zerop z)
                (plus x y)
              (plus y (times x z)))))

(prove-lemma plus3-times-sub1 (rewrite)
     (equal (plus x y z (times x (sub1 z1)))
            (if (zerop z1)
                (plus x y z)
              (plus y z (times x z1)))))

(prove-lemma lessp-cancel-4294967295 (rewrite)
     (equal (lessp (plus 4294967295 x) 4294967296)
            (zerop x))
     ((use (plus-lessp-cancel-add1 (y 4294967295)))))

(prove-lemma difference-cancel-4294967295 (rewrite)
     (equal (difference (plus 4294967295 x) 4294967296)
            (sub1 x))
     ((use (difference-plus-cancel-add1 (y 4294967295)))))

(prove-lemma lessp-cancel-4294967292 (rewrite)
     (equal (lessp (plus 4294967292 x) 4294967296)
            (lessp x 4))
     ((use (plus-lessp-cancel-1 (x 4294967292) (y x) (z 4)))
      (disable plus-lessp-cancel-1)))

(prove-lemma difference-cancel-4294967292 (rewrite)
     (equal (difference (plus 4294967292 x) 4294967296)
            (difference x 4)))

(prove-lemma difference-lessp-cancel-1 (rewrite)
     (equal (lessp (difference a c) (times c (sub1 b)))
            (if (leq c a) (lessp a (times c b)) (lessp 1 b)))
     ((use (difference-lessp-cancel (b (times c b))))
      (disable difference-lessp-cancel)))
```

```
(prove-lemma difference-lessp-cancel-2 (rewrite)
     (implies (leq c a)
              (equal (lessp (difference a c) (plus d (times c (sub1 b))))
                     (if (zerop b)
                         (lessp (difference a c) d)
                       (lessp a (plus d (times c b))))))
     ((use (difference-lessp-cancel (b (plus d (times c b)))))
      (disable difference-lessp-cancel)))

; two funny lemmas.  But seems more useful than add-uint!
(prove-lemma add-uintxx (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (lessp (plus (nat-to-uint x) (nat-to-uint y))
                          (exp 2 n)))
              (equal (nat-to-uint (add n x y))
                     (plus (nat-to-uint x) (nat-to-uint y)))))

(prove-lemma add-uintxxx (rewrite)
     (implies (and (nat-rangep x n)
                   (nat-rangep y n)
                   (not (lessp (plus (nat-to-uint x) (nat-to-uint y))
                               (exp 2 n))))
              (equal (nat-to-uint (add n x y))
                     (difference (plus (nat-to-uint x) (nat-to-uint y))
                                 (exp 2 n)))))

(disable add-uintxxx)

(prove-lemma ram-addrp-3 (rewrite)
     (implies (and (ram-addrp (add 32 addr i) mem k)
                   (leq (plus (index-n h i) j) k))
              (ram-addrp (add 32 addr h) mem j)))

(prove-lemma ram-addrp-4 (rewrite)
     (implies (and (ram-addrp addr mem k)
                   (leq (plus (nat-to-uint h) j) k))
              (ram-addrp (add 32 addr h) mem j))
     ((enable nat-to-uint)))

(prove-lemma and-z-commutativity (rewrite)
     (equal (and-z n x y) (and-z n y x))
     ((enable and-z)))

; finally, establish the database.
(make-lib "mc20-2")
```

Appendix C

The Nqthm Script of Program Proofs

Appendix C provides a complete documentation for all the proofs presented in this dissertation.

C.1 Greatest Common Divisor

```
;            Proof of the Correctness of a GCD Program
#|

The following C program computes the greatest common divisor of two
integers a and b. We, here, investigate the machine code of this
program generated by a widely used C compiler, and verify the
correctness of the code.

/* computes the greatest common divisor by Euclid's algorithm */
gcd(int a, int b)
{
  while (a != 0){
    if (b == 0) return (a);
    if (a > b)
      a = a % b;
    else b = b % a;
  };
  return (b);
}

Here is the MC68020 assembly code of the above GCD program.  The code is
generated by "gcc -O".

0x22c6 <gcd>:           linkw a6,#0
0x22ca <gcd+4>:         moveml d2-d3,sp@-
0x22ce <gcd+8>:         movel a6@(8),d2
0x22d2 <gcd+12>:        movel a6@(12),d3
0x22d6 <gcd+16>:        tstl d2
0x22d8 <gcd+18>:        beq 0x22f6 <gcd+48>
0x22da <gcd+20>:        tstl d3
0x22dc <gcd+22>:        bne 0x22e2 <gcd+28>
0x22de <gcd+24>:        movel d2,d0
0x22e0 <gcd+26>:        bra 0x22f8 <gcd+50>
0x22e2 <gcd+28>:        cmpl d2,d3
0x22e4 <gcd+30>:        bge 0x22ee <gcd+40>
```

```
0x22e6 <gcd+32>:        divsll d3,d0,d2
0x22ea <gcd+36>:        movel d0,d2
0x22ec <gcd+38>:        bra 0x22d6 <gcd+16>
0x22ee <gcd+40>:        divsll d2,d0,d3
0x22f2 <gcd+44>:        movel d0,d3
0x22f4 <gcd+46>:        bra 0x22d6 <gcd+16>
0x22f6 <gcd+48>:        movel d3,d0
0x22f8 <gcd+50>:        moveml a6@(-8),d2-d3
0x22fe <gcd+56>:        unlk a6
0x2300 <gcd+58>:        rts

The machine code of the above program is:

<gcd>:       0x4e56  0x0000  0x48e7  0x3000  0x242e  0x0008  0x262e  0x000c
<gcd+16>:    0x4a82  0x671c  0x4a83  0x6604  0x2002  0x6016  0xb682  0x6c08
<gcd+32>:    0x4c43  0x2800  0x2400  0x60e8  0x4c42  0x3800  0x2600  0x60e0
<gcd+48>:    0x2003  0x4cee  0x000c  0xfff8  0x4e5e  0x4e75

'(78      86      0       0       72      231     48      0
  36      46      0       8       38      46      0       12
  74      130     103     28      74      131     102     4
  32      2       96      22      182     130     108     8
  76      67      40      0       36      0       96      232
  76      66      56      0       38      0       96      224
  32      3       76      238     0       12      255     248
  78      94      78      117)
|#

; now we start the correctness proof of this GCD program, defined by
; (gcd-code).
(defn gcd-code ()
  '(78      86      0       0       72      231     48      0
    36      46      0       8       38      46      0       12
    74      130     103     28      74      131     102     4
    32      2       96      22      182     130     108     8
    76      67      40      0       36      0       96      232
    76      66      56      0       38      0       96      224
    32      3       76      238     0       12      255     248
    78      94      78      117))

(constrain gcd-load (rewrite)
   (equal (gcd-loadp s)
          (and (evenp (gcd-addr))
               (numberp (gcd-addr))
               (nat-rangep (gcd-addr) 32)
               (rom-addrp (gcd-addr) (mc-mem s) 60)
               (mcode-addrp (gcd-addr) (mc-mem s) (gcd-code))))
    ((gcd-loadp (lambda (s) f))
     (gcd-addr (lambda () 1))))

(prove-lemma stepn-gcd-loadp (rewrite)
     (equal (gcd-loadp (stepn s n))
            (gcd-loadp s)))
```

```
; the functional description of the program (gcd-code).
(defn gcd (a b)
  (if (zerop a)
      (fix b)
    (if (zerop b)
        a
      (if (lessp b a)
          (gcd (remainder a b) b)
        (gcd a (remainder b a)))))
  ((lessp (plus a b))))

; the clock function.
(defn gcd-t1 (a b)
  (if (zerop a)
      6
      (if (zerop b)
          9
        (if (lessp b a)
            (splus 9 (gcd-t1 (remainder a b) b))
          (splus 9 (gcd-t1 a (remainder b a))))))
  ((lessp (plus a b))))

(defn gcd-t (a b)
  (splus 4 (gcd-t1 a b)))

; an induction hint.
(defn gcd-induct (s a b)
  (if (or (zerop a) (zerop b))
      t
    (if (lessp b a)
        (gcd-induct (stepn s 9) (remainder a b) b)
      (gcd-induct (stepn s 9) a (remainder b a))))
  ((lessp (plus a b))))

; the initial state.
(defn gcd-statep (s a b)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (mc-pc s) (mc-mem s) 60)
       (mcode-addrp (mc-pc s) (mc-mem s) (gcd-code))
       (ram-addrp (sub 32 12 (read-sp s)) (mc-mem s) 24)
       (equal a (iread-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal b (iread-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
       (numberp a)
       (numberp b)))

; an intermediate state.
(defn gcd-s0p (s a b)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 16 (mc-pc s)) (mc-mem s) 60)
       (mcode-addrp (sub 32 16 (mc-pc s)) (mc-mem s) (gcd-code))
       (ram-addrp (sub 32 8 (read-an 32 6 s)) (mc-mem s) 24)
       (equal a (iread-dn 32 2 s))
```

```
       (equal b (iread-dn 32 3 s))
       (numberp a)
       (numberp b)))

; s --> s0.
(prove-lemma gcd-s-s0 (rewrite)
     (implies (gcd-statep s a b)
              (and (gcd-s0p (stepn s 4) a b)
                   (equal (linked-rts-addr (stepn s 4)) (rts-addr s))
                   (equal (linked-a6 (stepn s 4)) (read-an 32 6 s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 4)))
                          (sub 32 4 (read-sp s)))
                   (equal (movem-saved (stepn s 4) 4 8 2)
                          (readm-rn 32 '(2 3) (mc-rfile s))))))

(prove-lemma gcd-s-s0-rfile (rewrite)
     (implies (and (gcd-statep s a b)
                   (d4-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 4)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma gcd-s-s0-mem (rewrite)
     (implies (and (gcd-statep s a b)
                   (disjoint x l (sub 32 12 (read-sp s)) 24))
              (equal (read-mem x (mc-mem (stepn s 4)) l)
                     (read-mem x (mc-mem s) l))))

; s0 --> exit.
; base case: s0 --> exit.
(prove-lemma gcd-s0-sn-base-1 (rewrite)
     (implies (and (gcd-s0p s a b)
                   (zerop a))
              (and (equal (mc-status (stepn s 6)) 'running)
                   (equal (mc-pc (stepn s 6)) (linked-rts-addr s))
                   (equal (iread-dn 32 0 (stepn s 6)) (fix b))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 6))) (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 6)))
                          (add 32 (read-an 32 6 s) 8))
                   (equal (read-mem x (mc-mem (stepn s 6)) l)
                          (read-mem x (mc-mem s) l)))))

(prove-lemma gcd-s0-sn-base-rfile-1 (rewrite)
     (implies (and (gcd-s0p s a b)
                   (zerop a)
                   (d2-7a2-5p rn)
                   (leq oplen 32))
              (equal (read-rn oplen rn (mc-rfile (stepn s 6)))
                     (if (d4-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (get-vlst oplen 0 rn '(2 3) (movem-saved s 4 8 2))))))

(prove-lemma gcd-s0-sn-base-2 (rewrite)
     (implies (and (gcd-s0p s a b)
                   (not (zerop a))
```

```
                (zerop b))
           (and (equal (mc-status (stepn s 9)) 'running)
                (equal (mc-pc (stepn s 9)) (linked-rts-addr s))
                (equal (iread-dn 32 0 (stepn s 9)) (fix a))
                (equal (read-rn 32 14 (mc-rfile (stepn s 9))) (linked-a6 s))
                (equal (read-rn 32 15 (mc-rfile (stepn s 9)))
                       (add 32 (read-an 32 6 s) 8))
                (equal (read-mem x (mc-mem (stepn s 9)) l)
                       (read-mem x (mc-mem s) l)))))

(prove-lemma gcd-s0-sn-base-rfile-2 (rewrite)
     (implies (and (gcd-s0p s a b)
                (not (zerop a))
                (zerop b)
                (d2-7a2-5p rn)
                (leq oplen 32))
           (equal (read-rn oplen rn (mc-rfile (stepn s 9)))
                  (if (d4-7a2-5p rn)
                      (read-rn oplen rn (mc-rfile s))
                    (get-vlst oplen 0 rn '(2 3) (movem-saved s 4 8 2))))))

; induction case:  s0 --> s0.
(enable integerp)
(enable iremainder)

(prove-lemma gcd-s0-s0-1 (rewrite)
     (implies (and (gcd-s0p s a b)
                (not (zerop a))
                (not (zerop b))
                (lessp b a))
           (and (gcd-s0p (stepn s 9) (remainder a b) b)
                (equal (read-rn oplen 14 (mc-rfile (stepn s 9)))
                       (read-rn oplen 14 (mc-rfile s)))
                (equal (linked-a6 (stepn s 9)) (linked-a6 s))
                (equal (linked-rts-addr (stepn s 9))
                       (linked-rts-addr s))
                (equal (movem-saved (stepn s 9) 4 8 2)
                       (movem-saved s 4 8 2))
                (equal (read-mem x (mc-mem (stepn s 9)) l)
                       (read-mem x (mc-mem s) l)))))

(prove-lemma gcd-s0-s0-2 (rewrite)
     (implies (and (gcd-s0p s a b)
                (not (zerop a))
                (not (zerop b))
                (not (lessp b a)))
           (and (gcd-s0p (stepn s 9) a (remainder b a))
                (equal (read-rn oplen 14 (mc-rfile (stepn s 9)))
                       (read-rn oplen 14 (mc-rfile s)))
                (equal (linked-a6 (stepn s 9)) (linked-a6 s))
                (equal (linked-rts-addr (stepn s 9))
                       (linked-rts-addr s))
                (equal (movem-saved (stepn s 9) 4 8 2)
                       (movem-saved s 4 8 2))
```

```
                  (equal (read-mem x (mc-mem (stepn s 9)) l)
                         (read-mem x (mc-mem s) l)))))

(prove-lemma gcd-s0-s0-rfile-1 (rewrite)
     (implies (and (gcd-s0p s a b)
                   (not (zerop a))
                   (not (zerop b))
                   (lessp b a)
                   (d4-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 9)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma gcd-s0-s0-rfile-2 (rewrite)
     (implies (and (gcd-s0p s a b)
                   (not (zerop a))
                   (not (zerop b))
                   (not (lessp b a))
                   (d4-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 9)))
                     (read-rn oplen rn (mc-rfile s)))))

; put together.
(prove-lemma gcd-s0-sn (rewrite)
     (let ((sn (stepn s (gcd-t1 a b))))
       (implies (gcd-s0p s a b)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (iread-dn 32 0 sn) (gcd a b))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))
                     (equal (read-mem x (mc-mem sn) k)
                            (read-mem x (mc-mem s) k))))))
     ((induct (gcd-induct s a b))
      (disable gcd-s0p iread-dn linked-rts-addr linked-a6)))

(prove-lemma gcd-s0-sn-rfile (rewrite)
     (implies (and (gcd-s0p s a b)
                   (d2-7a2-5p rn)
                   (leq oplen 32))
              (equal (read-rn oplen rn (mc-rfile (stepn s (gcd-t1 a b))))
                     (if (d4-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (get-vlst oplen 0 rn '(2 3) (movem-saved s 4 8 2))))))
     ((induct (gcd-induct s a b))
      (disable gcd-s0p)))

; the correctness of gcd.
(prove-lemma gcd-correctness (rewrite)
     (let ((sn (stepn s (gcd-t a b))))
       (implies (gcd-statep s a b)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
```

```
                    (equal (read-rn 32 14 (mc-rfile sn))
                           (read-rn 32 14 (mc-rfile s)))
                    (equal (read-rn 32 15 (mc-rfile sn))
                           (add 32 (read-an 32 7 s) 4))
                    (implies (and (d2-7a2-5p rn)
                                  (leq oplen 32))
                             (equal (read-rn oplen rn (mc-rfile sn))
                                    (read-rn oplen rn (mc-rfile s))))
                    (implies (disjoint x k (sub 32 12 (read-sp s)) 24)
                             (equal (read-mem x (mc-mem sn) k)
                                    (read-mem x (mc-mem s) k)))
                    (equal (iread-dn 32 0 sn) (gcd a b)))))
     ((disable gcd-statep gcd-s0p linked-rts-addr linked-a6)))

(disable gcd-t)

; to prove that (gcd a b) does compute the greatest common divisor, we need
; to prove the following two theorems:
;     1. (gcd a b) divides a and (gcd a b) divides b.
;        i.e. (gcd a b) is a common divisor of a and b.
;     2. any divisor of a and b is no greater than (gcd a b).
(prove-lemma remainder-remainder (rewrite)
     (implies (equal (remainder b c) 0)
              (equal (remainder (remainder a b) c)
                     (remainder a c))))

(prove-lemma gcd-is-cd (rewrite)
     (and (equal (remainder a (gcd a b)) 0)
          (equal (remainder b (gcd a b)) 0)))

(prove-lemma gcd-the-greatest (rewrite)
     (implies (and (not (zerop a))
                   (not (zerop b))
                   (equal (remainder a x) 0)
                   (equal (remainder b x) 0))
              (not (lessp (gcd a b) x)))
     ((induct (gcd a b))))

; a simple timing analysis.
(prove-lemma lessp-times-lessp (rewrite)
     (implies (and (lessp a b)
                   (not (zerop c)))
              (lessp a (times b c))))

(prove-lemma remainder-shrink-half (rewrite)
     (implies (and (leq b a)
                   (not (zerop b)))
              (not (lessp (quotient a 2) (remainder a b))))
     ((expand (times 2 x))
      (enable quotient-generalize)))

(prove-lemma gcd-t-shrink-1 (rewrite)
     (implies (and (leq b a)
                   (not (zerop b)))
```

```
            (not (lessp (sub1 (log 2 a)) (log 2 (remainder a b)))))
     ((use (log-leq (b 2) (x (remainder a b)) (y (quotient a 2))))
      (disable quotient-times-lessp)))

(prove-lemma gcd-t-shrink-2 (rewrite)
     (implies (and (leq b a)
                   (not (zerop b))
                   (not (equal a 1)))
              (not (lessp (times 9 (plus x (log 2 a)))
                          (plus 9 (times 9 (plus x (log 2 (remainder a b))))))))
     ((use (log-leq (b 2) (x (remainder a b)) (y (quotient a 2))))
      (disable quotient-times-lessp)))

(defn gcd-t2 (a b)
  (if (zerop a)
      6
    (if (zerop b)
        9
      (if (lessp b a)
          (plus 9 (gcd-t2 (remainder a b) b))
        (if (lessp a b)
            (plus 9 (gcd-t2 a (remainder b a)))
          18))))
  ((lessp (plus a b))))

(prove-lemma gcd-t2-ub ()
     (leq (gcd-t2 a b) (plus 18 (times 9 (plus (log 2 a) (log 2 b))))))

(prove-lemma gcd-t1-t2 (rewrite)
     (equal (gcd-t1 a b) (gcd-t2 a b))
     ((enable splus)))

(prove-lemma gcd-t-ub ()
     (leq (gcd-t a b)
          (plus 22 (times 9 (plus (log 2 a) (log 2 b)))))
     ((use (gcd-t2-ub))
      (enable gcd-t splus)))

(prove-lemma gcd-t-ubound ()
     (implies (and (lessp a (exp 2 31))
                   (lessp b (exp 2 31)))
              (leq (gcd-t a b) 580))
     ((use (ta-lemma-2 (x 22) (y 9) (a1 (exp 2 31)) (b1 (exp 2 31)))
           (gcd-t-ub))))
```

C.2 Integer Square Root

```
;       Proof of the Correctness of an Integer Square Root Program
;
#|
This is a revisit to our ISQRT proof.  Dr. Don Good talked with
```

Dr. Steve Zeigler, vice president for Ada products at Verdix, and they agreed to let me publicize the machine codes generated by their present Ada compiler.

The following Ada function ISQRT computes the integer square root of a given nonnegative integer i. This is our third proof about ISQRT.

```
function isqrt (i:integer) return integer is
  j : integer := (i / 2);
 begin
  while ((i / j) < j) loop
    j := (j + (i / j)) / 2;
  end loop;
  return j;
 end isqrt;
```

Here is the MC68020 assembly code of the above ISQRT program. The code is from Dr. Steve Zeigler and generated by their present Ada compiler.

```
   1  function isqrt (i:integer) return integer is
      00000: link.w      a6, #-04
   2     j : integer := (i / 2);
      00004: move.l      d2, d1
      00006: bge.b       06    -> 0e
      00008: addi.l      #01, d1
      0000e: asr.l       #01, d1
   3  begin
   4     while not ((i / j) >= j) loop
      00010: move.l      d2, d0
      00012: divsl.l     d1, d0:d0
      00016: trapv
      00018: cmp.l       d0, d1
      0001a: ble.b       01c    -> 038
   5        j := (j + (i / j)) / 2;
      0001c: add.l       d1, d0
      0001e: trapv
      00020: move.l      d0, d1
      00022: bge.b       06    -> 02a
      00024: addi.l      #01, d1
      0002a: asr.l       #01, d1
   4     while not ((i / j) >= j) loop
      0002c: move.l      d2, d0
      0002e: divsl.l     d1, d0:d0
      00032: trapv
   6  end loop;
      00034: cmp.l       d0, d1
      00036: bgt.b       -01c    -> 01c
   7  return j;
      00038: move.l      d1, d0
      0003a: unlk        a6
      0003c: rts
   8 end isqrt;
```

```
0x4e56  0xfffc  0x2202  0x6c06  0x0681  0x0000  0x0001  0xe281
```

```
0x2002  0x4c41  0x0800  0x4e76  0xb280  0x6f1c  0xd081  0x4e76
0x2200  0x6c06  0x0681  0x0000  0x0001  0xe281  0x2002  0x4c41
0x0800  0x4e76  0xb280  0x6ee4  0x2001  0x4e5e  0x4e75

'(78      86      255     252     34      2       108     6
  6       129     0       0       0       1       226     129
  32      2       76      65      8       0       78      118
  178     128     111     28      208     129     78      118
  34      0       108     6       6       129     0       0
  0       1       226     129     32      2       76      65
  8       0       78      118     178     128     110     228
  32      1       78      94      78      117)
|#

; in the logic, the above program is defined by (isqrt-code).
(defn isqrt-code ()
  '(78      86      255     252     34      2       108     6
    6       129     0       0       0       1       226     129
    32      2       76      65      8       0       78      118
    178     128     111     28      208     129     78      118
    34      0       108     6       6       129     0       0
    0       1       226     129     32      2       76      65
    8       0       78      118     178     128     110     228
    32      1       78      94      78      117))

(defn sq (x)
  (times x x))

; isqrt1 is a function in the Logic simulating the loop of the above code.
(defn isqrt1 (i j)
  (if (zerop j)
      (fix i)
    (if (lessp (quotient i j) j)
        (isqrt1 i (quotient (plus j (quotient i j)) 2))
      (fix j))))

; isqrt specifies the semantics of ISQRT in the Logic.  To see why
; the function isqrt computes the square root for any nonnegative
; integer input, please refer to the proof in file isqrt.events.
(defn isqrt (i)
  (let ((j1 (quotient (plus (quotient i 2) (quotient i (quotient i 2))) 2)))
    (if (lessp i (sq (quotient i 2)))
        (isqrt1 i j1)
      (quotient i 2))))

; the computation time of this program.
(defn isqrt1-t (i j)
  (if (zerop j)
      0
    (if (lessp (quotient i j) j)
        (splus 10 (isqrt1-t i (quotient (plus j (quotient i j)) 2)))
      8)))

(defn isqrt-t (i)
```

```
  (let ((j1 (quotient (plus (quotient i 2) (quotient i (quotient i 2))) 2)))
    (if (lessp i (sq (quotient i 2)))
        (splus 14 (isqrt1-t i j1))
      12)))

; enable a few functions.
(enable iplus)
(enable integerp)
(enable iquotient)
(enable ilessp)

; disable a few functions.
(disable remainder)
(disable quotient)

(prove-lemma isqrt-no-overflow (rewrite)
             (implies (and (int-rangep (times 2 j) n)
                           (lessp (quotient i j) j))
                      (int-rangep (plus j (quotient i j)) n)))

(prove-lemma j-nonzerop (rewrite)
            (implies (and (lessp 1 i)
                          (lessp 0 j))
                     (not (equal (quotient (plus j (quotient i j)) 2) 0))))

(prove-lemma j-int-rangep (rewrite)
     (implies (and (int-rangep (times 2 j) n)
                   (lessp (quotient i j) j))
              (int-rangep (times 2 (quotient (plus j (quotient i j)) 2))
                          n)))

; an induction hint.
(defn isqrt-induct (s i j)
  (if (zerop j)
      t
    (if (lessp (quotient i j) j)
        (isqrt-induct (stepn s 10) i (quotient (plus j (quotient i j)) 2))
      t)))

; the properties of the initial state.
(defn isqrt-statep (s i)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (mc-pc s) (mc-mem s) 70)
       (mcode-addrp (mc-pc s) (mc-mem s) (isqrt-code))
       (ram-addrp (sub 32 8 (read-sp s)) (mc-mem s) 12)
       (equal i (iread-dn 32 2 s))
       (ilessp 1 i)))

; an intermediate state s0.
(defn isqrt-s0p (s i j)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 44 (mc-pc s)) (mc-mem s) 70)
```

```
       (mcode-addrp (sub 32 44 (mc-pc s)) (mc-mem s) (isqrt-code))
       (ram-addrp (sub 32 4 (read-an 32 6 s)) (mc-mem s) 12)
       (equal i (iread-dn 32 2 s))
       (equal j (iread-dn 32 1 s))
       (int-rangep (times 2 j) 32)
       (ilessp 1 i)
       (ilessp 0 j)))

(prove-lemma initial-j-int-rangep (rewrite)
     (implies (int-rangep i n)
              (int-rangep (times 2 (quotient i 2)) n))
     ((enable int-rangep)))

; from the initial state to exit.
(prove-lemma isqrt-s-exit (rewrite)
     (implies (and (isqrt-statep s i)
                   (not (lessp i (times (quotient i 2) (quotient i 2)))))
              (and (equal (mc-status (stepn s 12)) 'running)
                   (equal (mc-pc (stepn s 12)) (rts-addr s))
                   (equal (iread-dn 32 0 (stepn s 12)) (quotient i 2))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 12)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 12)))
                          (add 32 (read-sp s) 4)))))

(prove-lemma isqrt-s-exit-rfile (rewrite)
     (implies (and (isqrt-statep s i)
                   (not (lessp i (times (quotient i 2) (quotient i 2))))
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 12)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma isqrt-s-exit-mem (rewrite)
     (implies (and (isqrt-statep s i)
                   (not (lessp i (times (quotient i 2) (quotient i 2))))
                   (disjoint x k (sub 32 8 (read-sp s)) 12))
              (equal (read-mem x (mc-mem (stepn s 12)) k)
                     (read-mem x (mc-mem s) k))))

; from the initial state to s0.
(prove-lemma isqrt-s-s0 (rewrite)
     (let ((j1 (quotient (plus (quotient i 2) (quotient i (quotient i 2))) 2)))
       (implies (and (isqrt-statep s i)
                     (lessp i (times (quotient i 2) (quotient i 2))))
                (and (isqrt-s0p (stepn s 14) i j1)
                     (equal (linked-rts-addr (stepn s 14)) (rts-addr s))
                     (equal (linked-a6 (stepn s 14)) (read-an 32 6 s))
                     (equal (read-rn 32 14 (mc-rfile (stepn s 14)))
                            (sub 32 4 (read-sp s)))))))
     ((disable quotient-equal-0)))

(prove-lemma isqrt-s-s0-rfile (rewrite)
     (implies (and (isqrt-statep s i)
                   (lessp i (times (quotient i 2) (quotient i 2)))
```

```
                (d2-7a2-5p rn))
           (equal (read-rn oplen rn (mc-rfile (stepn s 14)))
                  (read-rn oplen rn (mc-rfile s))))
     ((disable quotient-equal-0)))

(prove-lemma isqrt-s-s0-mem (rewrite)
     (implies (and (isqrt-statep s i)
                   (lessp i (times (quotient i 2) (quotient i 2)))
                   (disjoint x k (sub 32 8 (read-sp s)) 12))
              (equal (read-mem x (mc-mem (stepn s 14)) k)
                     (read-mem x (mc-mem s) k)))
     ((disable quotient-equal-0)))

; from s0 to exit (base case), or from s0 to s0 (induction case).
; base case: s0 --> exit.
(prove-lemma isqrt-s0-exit-base (rewrite)
     (implies (and (isqrt-s0p s i j)
                   (not (lessp (quotient i j) j)))
              (and (equal (mc-status (stepn s 8)) 'running)
                   (equal (mc-pc (stepn s 8)) (linked-rts-addr s))
                   (equal (iread-dn 32 0 (stepn s 8)) j)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 8)))
                          (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 8)))
                          (add 32 (read-an 32 6 s) 8)))))

(prove-lemma isqrt1-s0-exit-rfile-base (rewrite)
     (implies (and (isqrt-s0p s i j)
                   (not (lessp (quotient i j) j))
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 8)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma isqrt1-s0-exit-mem-base (rewrite)
     (implies (and (isqrt-s0p s i j)
                   (not (lessp (quotient i j) j)))
              (equal (read-mem x (mc-mem (stepn s 8)) k)
                     (read-mem x (mc-mem s) k))))

; induction case: s0 --> s0.
(prove-lemma isqrt-s0-s0 (rewrite)
     (implies (and (isqrt-s0p s i j)
                   (lessp (quotient i j) j))
              (and (isqrt-s0p (stepn s 10)
                              i
                              (quotient (plus j (quotient i j)) 2))
                   (equal (read-rn oplen 14 (mc-rfile (stepn s 10)))
                          (read-rn oplen 14 (mc-rfile s)))
                   (equal (linked-a6 (stepn s 10)) (linked-a6 s))
                   (equal (linked-rts-addr (stepn s 10))
                          (linked-rts-addr s))))
     ((disable quotient-equal-0 quotient-times-lessp)))

(prove-lemma isqrt-s0-s0-rfile (rewrite)
```

```
     (implies (and (isqrt-s0p s i j)
                   (lessp (quotient i j) j)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 10)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma isqrt-s0-s0-mem (rewrite)
     (implies (and (isqrt-s0p s i j)
                   (lessp (quotient i j) j)
                   (disjoint x k (sub 32 8 (read-an 32 6 s)) 20))
              (equal (read-mem x (mc-mem (stepn s 10)) k)
                     (read-mem x (mc-mem s) k))))

(prove-lemma isqrt-s0p-j_nonzerop (rewrite)
             (implies (isqrt-s0p s i j)
                      (and (not (equal j 0))
                           (numberp j))))

(disable isqrt-statep)
(disable isqrt-s0p)

; put together:  s0 --> exit.
(prove-lemma isqrt-s0-exit (rewrite)
     (let ((sn (stepn s (isqrt1-t i j))))
       (implies (isqrt-s0p s i j)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (iread-dn 32 0 sn) (isqrt1 i j))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))))))
    ((induct (isqrt-induct s i j))
     (disable quotient-times-lessp linked-rts-addr linked-a6 iread-dn)))

(prove-lemma isqrt-s0-exit-rfile (rewrite)
     (implies (and (isqrt-s0p s i j)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s (isqrt1-t i j))))
                     (read-rn oplen rn (mc-rfile s))))
     ((induct (isqrt-induct s i j))))

(prove-lemma isqrt-s0-exit-mem (rewrite)
     (implies (and (isqrt-s0p s i j)
                   (disjoint x k (sub 32 8 (read-an 32 6 s)) 20))
              (equal (read-mem x (mc-mem (stepn s (isqrt1-t i j))) k)
                     (read-mem x (mc-mem s) k)))
     ((induct (isqrt-induct s i j))))

(disable isqrt-s0p-j_nonzerop)

; ISQRT program is correct.
; after an execution of this program,  the machine state satisfies:
;  0. normal exit.
```

```
;  1. the pc is returned to the next instruction of the caller.
;  2. the result -- ISQRT(i), is stored in D0.
;  3. a6, used by LINK, is restored to its original content.
;  4. the stack pointer sp(a7) is updated correctly to pop off one frame.
; the registers not touched by this program still have their original values.
; the memory is correctly changed -- the local effect on memory.
(prove-lemma isqrt-correctness (rewrite)
     (let ((sn (stepn s (isqrt-t i))))
       (implies (isqrt-statep s i)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (read-an 32 6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 7 s) 4))
                     (implies (d2-7a2-5p rn)
                              (equal (read-rn oplen rn (mc-rfile sn))
                                     (read-rn oplen rn (mc-rfile s))))
                     (implies (disjoint x k (sub 32 12 (read-sp s)) 20)
                              (equal (read-mem x (mc-mem sn) k)
                                     (read-mem x (mc-mem s) k)))
                     (equal (iread-dn 32 0 sn) (isqrt i))))))
     ((disable rts-addr linked-rts-addr linked-a6 iread-dn
               quotient-times-lessp)))

(disable isqrt-t)

; start to prove isqrt correct.
(prove-lemma isqrt1-lower-bound (rewrite)
     (implies (not (zerop j))
              (not (lessp i (sq (isqrt1 i j))))))

(prove-lemma quotient-by-2 (rewrite)
     (not (lessp (plus (quotient x 2) (quotient x 2))
                 (sub1 x))))

(prove-lemma main-trick (rewrite)
     (not (lessp (sq (add1 (quotient (plus j k) 2)))
                 (plus (times j k) j)))
     ((induct (difference j k))))

(prove-lemma sq-add1-non-zero (rewrite)
     (not (equal (sq (add1 x)) 0)))

(prove-lemma main (rewrite)
     (implies (not (zerop j))
              (lessp i
                     (sq (add1 (quotient (plus j (quotient i j)) 2))))))
     ((disable sq)))

(prove-lemma isqrt1-upper-bound (rewrite)
     (implies (lessp i (sq (add1 j)))
              (lessp i (sq (add1 (isqrt1 i j))))))
     ((disable sq)))
```

```
(prove-lemma isqrt->isqrt1 (rewrite)
     (implies (lessp 1 i)
              (lessp i (sq (add1 (quotient i 2))))))

; (isqrt i) is the square root of i:  (isqrt i)^2 <= i < [(isqrt i)+1]^2.
(prove-lemma isqrt-logic-correctness ()
     (implies (lessp 1 i)
              (and (lessp i (sq (add1 (isqrt i))))
                   (not (lessp i (sq (isqrt i)))))
     ((disable sq isqrt1)))

; a simple time analysis.
(prove-lemma quotient-mono-1 (rewrite)
     (implies (and (leq y z)
                   (not (zerop y))
                   (not (zerop z)))
              (not (lessp (quotient x y) (quotient x z)))))

(prove-lemma mean-lessp-1 (rewrite)
     (implies (not (lessp x y))
              (not (lessp x (quotient (plus x y) 2))))
     ((induct (difference x y))))

(prove-lemma isqrt1-t-la-0 ()
     (let ((j1 (quotient (plus j (quotient i j)) 2)))
       (implies (and (leq (quotient i j) j)
                     (lessp 1 i)
                     (lessp 0 j))
                (not (lessp (difference x (quotient i j))
                            (difference x (quotient i j1)))))))
     ((use (quotient-mono-1 (x i)
                            (y (quotient (plus j (quotient i j)) 2))
                            (z j)))
      (disable quotient-times-lessp)))

(prove-lemma mean-difference-1-1 (rewrite)
     (implies (leq i j)
              (equal (difference (quotient (plus j i) 2) i)
                     (quotient (difference j i) 2))))

(prove-lemma isqrt1-t-la-1 (rewrite)
     (let ((j1 (quotient (plus j (quotient i j)) 2)))
       (implies (and (lessp (quotient i j) j)
                     (lessp 1 i)
                     (lessp 0 j))
                (not (lessp (quotient (difference j (quotient i j)) 2)
                            (difference j1 (quotient i j1)))))))
     ((use (isqrt1-t-la-0 (x (quotient (plus j (quotient i j)) 2))))
      (disable quotient-times-lessp)))

(prove-lemma isqrt1-t-la-2 (rewrite)
     (let ((j1 (quotient (plus j (quotient i j)) 2)))
       (implies (and (lessp (quotient i j) j)
```

```
                    (lessp 1 i)
                    (lessp 0 j))
               (not (lessp (log 2 (quotient (difference j (quotient i j)) 2))
                           (log 2 (difference j1 (quotient i j1)))))))
     ((use (isqrt1-t-la-1))
      (disable quotient-times-lessp log)))

(prove-lemma isqrt-t1-ubound ()
     (implies
      (and (lessp 1 i)
           (lessp 0 j))
      (not (lessp (plus 18 (times 10 (log 2 (difference j (quotient i j)))))
                  (isqrt1-t i j))))
     ((enable splus)
      (disable  quotient-times-lessp log)
      (expand (log 2 (difference j (quotient i j))))))

(prove-lemma isqrt-t->isqrt1-t ()
     (implies (lessp 1 i)
              (not (lessp (plus 4 (isqrt1-t i (quotient i 2))) (isqrt-t i))))
     ((enable splus isqrt-t)))

(prove-lemma log-mono (rewrite)
     (not (lessp (log b y) (log b (difference y x))))
     ((use (log-leq (x (difference y x))))))

(prove-lemma isqrt-t-ubound-la (rewrite)
     (implies (lessp 1 i)
              (not (lessp (plus 12 (times 10 (log 2 i)))
                          (isqrt-t i))))
     ((use (isqrt-t->isqrt1-t)
           (isqrt-t1-ubound (j (quotient i 2))))))

(prove-lemma isqrt-t-ubound ()
     (implies (and (lessp i (exp 2 31))
                   (lessp 1 i))
              (leq (isqrt-t i) 322))
     ((use (ta-lemma-1 (a i) (a1 (exp 2 31)) (x 12) (y 10)))))
```

C.3 Binary Search

```
;         Proof of the Correctness of a Binary Search Program
;
#|
The following C function BSEARCH determines if a given value x occurs
in the sorted array a.

/* from K&R */
/* bsearch:  find x in a[0] <= a[1] <= ... <= a[n-1] */
int bsearch (int x, int a[], int n)
{
  int low, high, mid;
```

```
  low = 0;
  high = n;
  while (low < high) {
    mid = (low + high) / 2;
    if (x < a[mid])
      high = mid;
    else if (x > a[mid])
      low = mid + 1;
    else return mid;
  }
  return -1;
}
```

Here is the MC68020 assembly code of the above BSEARCH program. The code is generated by "gcc -O".

```
0x22f0 <bsearch>:       linkw a6,#0
0x22f4 <bsearch+4>:     moveml d2-d3,sp@-
0x22f8 <bsearch+8>:     movel a6@(8),d3
0x22fc <bsearch+12>:    moveal a6@(12),a0
0x2300 <bsearch+16>:    clrl d1
0x2302 <bsearch+18>:    movel a6@(16),d2
0x2306 <bsearch+22>:    cmpl d1,d2
0x2308 <bsearch+24>:    ble 0x232a <bsearch+58>
0x230a <bsearch+26>:    movel d1,d0
0x230c <bsearch+28>:    addl d2,d0
0x230e <bsearch+30>:    bpl 0x2312 <bsearch+34>
0x2310 <bsearch+32>:    addql #1,d0
0x2312 <bsearch+34>:    asrl #1,d0
0x2314 <bsearch+36>:    cmpl 0(a0)[d0.l*4],d3
0x2318 <bsearch+40>:    bge 0x231e <bsearch+46>
0x231a <bsearch+42>:    movel d0,d2
0x231c <bsearch+44>:    bra 0x2306 <bsearch+22>
0x231e <bsearch+46>:    cmpl 0(a0)[d0.l*4],d3
0x2322 <bsearch+50>:    ble 0x232c <bsearch+60>
0x2324 <bsearch+52>:    movel d0,d1
0x2326 <bsearch+54>:    addql #1,d1
0x2328 <bsearch+56>:    bra 0x2306 <bsearch+22>
0x232a <bsearch+58>:    movel #-1,d0
0x232c <bsearch+60>:    moveml a6@(-8),d2-d3
0x2332 <bsearch+66>:    unlk a6
0x2334 <bsearch+68>:    rts
```

The machine code of the above program is:

```
<bsearch>:      0x4e56  0x0000  0x48e7  0x3000  0x262e  0x0008  0x206e  0x000c
<bsearch+16>:   0x4281  0x242e  0x0010  0xb481  0x6f20  0x2001  0xd082  0x6a02
<bsearch+32>:   0x5280  0xe280  0xb6b0  0x0c00  0x6c04  0x2400  0x60e8  0xb6b0
<bsearch+48>:   0x0c00  0x6f08  0x2200  0x5281  0x60dc  0x70ff  0x4cee  0x000c
<bsearch+64>:   0xfff8  0x4e5e  0x4e75
```

In the Logic, this is:

```
'(78      86      0       0       72      231     48      0
  38      46      0       8       32      110     0       12
  66      129     36      46      0       16      180     129
  111     32      32      1       208     130     106     2
  82      128     226     128     182     176     12      0
  108     4       36      0       96      232     182     176
  12      0       111     8       34      0       82      129
  96      220     112     255     76      238     0       12
  255     248     78      94      78      117)
|#

; in the logic, the above program is defined by (bsearch-code).
(defn bsearch-code ()
  '(78      86      0       0       72      231     48      0
    38      46      0       8       32      110     0       12
    66      129     36      46      0       16      180     129
    111     32      32      1       208     130     106     2
    82      128     226     128     182     176     12      0
    108     4       36      0       96      232     182     176
    12      0       111     8       34      0       82      129
    96      220     112     255     76      238     0       12
    255     248     78      94      78      117))

(prove-lemma mean-bounds (rewrite)
     (implies (lessp i j)
              (and (lessp (quotient (plus i j) 2) j)
                   (not (lessp (quotient (plus i j) 2) i)))))

(prove-lemma ilessp-lessp (rewrite)
     (implies (and (numberp x)
                   (numberp y))
              (equal (ilessp x y) (lessp x y))))

(disable ilessp)

; bsearch1 is a function in the logic to simulate the loop of the
; above code.
(defn bsearch1 (x lst i j)
  (let ((k (quotient (plus i j) 2)))
    (if (lessp i j)
        (if (ilessp x (get-nth k lst))
            (bsearch1 x lst i k)
          (if (ilessp (get-nth k lst) x)
              (bsearch1 x lst (add1 k) j)
            k))
      -1))
  ((lessp (difference j i))))

; bsearch is a function in the logic to simulate the above code.
(defn bsearch (x n lst)
  (bsearch1 x lst 0 n))

; the computation time of the loop.
(defn bsearch1-t (x lst i j)
```

```
  (let ((k (quotient (plus i j) 2)))
    (if (lessp i j)
        (if (ilessp x (get-nth k lst))
            (splus 10 (bsearch1-t x lst i k))
          (if (ilessp (get-nth k lst) x)
              (splus 13 (bsearch1-t x lst (add1 k) j))
            13))
      6))
  ((lessp (difference j i))))

; the computation time of the code.
(defn bsearch-t (x n lst)
  (splus 6 (bsearch1-t x lst 0 n)))

; an induction hint.
(defn bsearch-induct (s x lst i j)
  (let ((k (quotient (plus i j) 2)))
    (if (lessp i j)
        (if (ilessp x (get-nth k lst))
            (bsearch-induct (stepn s 10) x lst i k)
          (if (ilessp (get-nth k lst) x)
              (bsearch-induct (stepn s 13) x lst (add1 k) j)
            t))
      t))
  ((lessp (difference j i))))

; the preconditions of the initial state.
(defn bsearch-statep (s x a n lst)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (mc-pc s) (mc-mem s) 70)
       (mcode-addrp (mc-pc s) (mc-mem s) (bsearch-code))
       (ram-addrp (sub 32 12 (read-sp s)) (mc-mem s) 28)
       (ram-addrp a (mc-mem s) (times 4 n))
       (mem-ilst 4 a (mc-mem s) n lst)
       (disjoint (sub 32 12 (read-sp s)) 28 a (times 4 n))
       (equal a (read-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
       (equal n (iread-mem (add 32 (read-sp s) 12) (mc-mem s) 4))
       (equal x (iread-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (int-rangep (times 2 n) 32)
       (numberp n)))

; the conditions of an intermediate state.
(defn bsearch-s0p (s x a n lst i j)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 22 (mc-pc s)) (mc-mem s) 70)
       (mcode-addrp (sub 32 22 (mc-pc s)) (mc-mem s) (bsearch-code))
       (ram-addrp (sub 32 8 (read-an 32 6 s)) (mc-mem s) 28)
       (ram-addrp a (mc-mem s) (times 4 n))
       (mem-ilst 4 a (mc-mem s) n lst)
       (disjoint (sub 32 8 (read-an 32 6 s)) 28 a (times 4 n))
       (equal a (read-an 32 0 s))
       (equal i (nat-to-int (read-dn 32 1 s) 32))
```

```
       (equal j (nat-to-int (read-dn 32 2 s) 32))
       (equal x (nat-to-int (read-dn 32 3 s) 32))
       (int-rangep (times 2 j) 32)
       (numberp i)
       (numberp j)
       (numberp n)
       (leq i n)
       (leq j n)))

; the initial segment.  From the initial state to s0.
(prove-lemma bsearch-s-s0p ()
     (implies (bsearch-statep s x a n lst)
              (bsearch-s0p (stepn s 6) x a n lst 0 n)))

(prove-lemma bsearch-s-s0 (rewrite)
     (implies (bsearch-statep s x a n lst)
              (and (equal (linked-rts-addr (stepn s 6))
                          (rts-addr s))
                   (equal (linked-a6 (stepn s 6))
                          (read-an 32 6 s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 6)))
                          (sub 32 4 (read-sp s)))
                   (equal (movem-saved (stepn s 6) 4 8 2)
                          (readm-rn 32 '(2 3) (mc-rfile s))))))

(prove-lemma bsearch-s-s0-rfile (rewrite)
     (implies (and (bsearch-statep s x a n lst)
                   (d4-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 6)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma bsearch-s-s0-mem (rewrite)
     (implies (and (bsearch-statep s x a n lst)
                   (disjoint x k (sub 32 12 (read-sp s)) 28))
              (equal (read-mem x (mc-mem (stepn s 6)) k)
                     (read-mem x (mc-mem s) k))))

; from s0 to s0 (induction case), from s0 to exit (base case).
; base case: s0 --> sn.
(prove-lemma bsearch-s0-sn-base1 (rewrite)
     (implies (and (bsearch-s0p s x a n lst i j)
                   (not (lessp i j)))
              (and (equal (mc-status (stepn s 6)) 'running)
                   (equal (mc-pc (stepn s 6)) (linked-rts-addr s))
                   (equal (iread-dn 32 0 (stepn s 6)) -1)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 6)))
                          (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 6)))
                          (add 32 (read-an 32 6 s) 8))
                   (equal (read-mem x (mc-mem (stepn s 6)) 1)
                          (read-mem x (mc-mem s) 1)))))

(prove-lemma bsearch-s0-sn-rfile-base1 (rewrite)
     (implies (and (bsearch-s0p s x a n lst i j)
```

```
                (not (lessp i j))
                (d2-7a2-5p rn)
                (leq oplen 32))
           (equal (read-rn oplen rn (mc-rfile (stepn s 6)))
                  (if (d4-7a2-5p rn)
                      (read-rn oplen rn (mc-rfile s))
                    (get-vlst oplen 0 rn '(2 3) (movem-saved s 4 8 2))))))

(enable iplus)
(enable iquotient)
(disable quotient)
(disable remainder)
(disable times)

(prove-lemma bsearch-crock(rewrite)
     (implies (and (int-rangep (times 2 j) n)
                   (lessp i j))
              (int-rangep (plus i j) n)))

(prove-lemma bsearch-s0-sn-base2 (rewrite)
     (let ((k (quotient (plus i j) 2)))
       (implies (and (bsearch-s0p s x a n lst i j)
                     (lessp i j)
                     (not (ilessp x (get-nth k lst)))
                     (not (ilessp (get-nth k lst) x)))
                (and (equal (mc-status (stepn s 13)) 'running)
                     (equal (mc-pc (stepn s 13)) (linked-rts-addr s))
                     (equal (iread-dn 32 0 (stepn s 13))
                            (quotient (plus i j) 2))
                     (equal (read-rn 32 14 (mc-rfile (stepn s 13)))
                            (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile (stepn s 13)))
                            (add 32 (read-an 32 6 s) 8))
                     (equal (read-mem x (mc-mem (stepn s 13)) l)
                            (read-mem x (mc-mem s) l))))))

(prove-lemma bsearch1-s0-sn-rfile-base2 (rewrite)
    (let ((k (quotient (plus i j) 2)))
      (implies (and (bsearch-s0p s x a n lst i j)
                    (lessp i j)
                    (not (ilessp x (get-nth k lst)))
                    (not (ilessp (get-nth k lst) x))
                    (d2-7a2-5p rn)
                    (leq oplen 32))
               (equal (read-rn oplen rn (mc-rfile (stepn s 13)))
                      (if (d4-7a2-5p rn)
                          (read-rn oplen rn (mc-rfile s))
                        (get-vlst oplen 0 rn '(2 3) (movem-saved s 4 8 2)))))))

; from s0 to s0 (induction case).
(prove-lemma bsearch-s0-s0-1 (rewrite)
     (let ((k (quotient (plus i j) 2)))
       (implies (and (bsearch-s0p s x a n lst i j)
                     (lessp i j)
```

```
                    (ilessp x (get-nth k lst)))
               (and (bsearch-s0p (stepn s 10) x a n lst i k)
                    (equal (read-rn oplen 14 (mc-rfile (stepn s 10)))
                           (read-rn oplen 14 (mc-rfile s)))
                    (equal (linked-a6 (stepn s 10)) (linked-a6 s))
                    (equal (linked-rts-addr (stepn s 10))
                           (linked-rts-addr s))
                    (equal (movem-saved (stepn s 10) 4 8 2)
                           (movem-saved s 4 8 2))
                    (equal (read-mem x (mc-mem (stepn s 10)) l)
                           (read-mem x (mc-mem s) l))))))

(prove-lemma bsearch-s0-s0-rfile1 (rewrite)
     (let ((k (quotient (plus i j) 2)))
       (implies (and (bsearch-s0p s x a n lst i j)
                     (lessp i j)
                     (ilessp x (get-nth k lst))
                     (d4-7a2-5p rn))
                (equal (read-rn oplen rn (mc-rfile (stepn s 10)))
                       (read-rn oplen rn (mc-rfile s))))))

(prove-lemma bsearch-s0-s0-2 (rewrite)
     (let ((k (quotient (plus i j) 2)))
       (implies (and (bsearch-s0p s x a n lst i j)
                     (lessp i j)
                     (not (ilessp x (get-nth k lst)))
                     (ilessp (get-nth k lst) x))
                (and (bsearch-s0p (stepn s 13) x a n lst (add1 k) j)
                     (equal (read-rn oplen 14 (mc-rfile (stepn s 13)))
                            (read-rn oplen 14 (mc-rfile s)))
                     (equal (linked-a6 (stepn s 13)) (linked-a6 s))
                     (equal (linked-rts-addr (stepn s 13)) (linked-rts-addr s))
                     (equal (movem-saved (stepn s 13) 4 8 2)
                            (movem-saved s 4 8 2))
                     (equal (read-mem x (mc-mem (stepn s 13)) l)
                            (read-mem x (mc-mem s) l))))))

(prove-lemma bsearch1-s0-s0-rfile2 (rewrite)
     (let ((k (quotient (plus i j) 2)))
       (implies (and (bsearch-s0p s x a n lst i j)
                     (lessp i j)
                     (not (ilessp x (get-nth k lst)))
                     (ilessp (get-nth k lst) x)
                     (d4-7a2-5p rn))
                (equal (read-rn oplen rn (mc-rfile (stepn s 13)))
                       (read-rn oplen rn (mc-rfile s))))))

(disable bsearch-statep)
(disable bsearch-s0p)

(prove-lemma bsearch-s0-sn (rewrite)
     (let ((sn (stepn s (bsearch1-t x lst i j))))
       (implies (bsearch-s0p s x a n lst i j)
                (and (equal (mc-status sn) 'running)
```

```
                    (equal (mc-pc sn) (linked-rts-addr s))
                    (equal (iread-dn 32 0 sn) (bsearch1 x lst i j))
                    (equal (read-rn 32 14 (mc-rfile sn))
                           (linked-a6 s))
                    (equal (read-rn 32 15 (mc-rfile sn))
                           (add 32 (read-an 32 6 s) 8))
                    (equal (read-mem x (mc-mem sn) k)
                           (read-mem x (mc-mem s) k)))))
     ((induct (bsearch-induct s x lst i j))
      (disable linked-rts-addr linked-a6)))

(prove-lemma bsearch-s0-sn-rfile (rewrite)
     (implies
      (and (bsearch-s0p s x a n lst i j)
           (d2-7a2-5p rn)
           (leq oplen 32))
      (equal (read-rn oplen rn (mc-rfile (stepn s (bsearch1-t x lst i j))))
             (if (d4-7a2-5p rn)
                 (read-rn oplen rn (mc-rfile s))
               (get-vlst oplen 0 rn '(2 3) (movem-saved s 4 8 2)))))
     ((induct (bsearch-induct s x lst i j))))

; the correctness statement of this BSEARCH program.
; after an execution of this program,  the machine state satisfies:
;  0. normal exit.
;  1. the program counter is returned to the next instruction of the caller.
;  2. the result -- (bsearch x n lst), is stored in the register D0.
;  3. a6, used by LINK, is restored to its original value.
;  4. a7, the stack pointer, is updated correctly to pop off one frame.
;  5. the registers d2-d7 and a2-a5 maintain their original values.
;  6. the memory is only locally changed wrt this program.
(prove-lemma bsearch-correctness (rewrite)
     (let ((sn (stepn s (bsearch-t x n lst))))
       (implies (bsearch-statep s x a n lst)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-sp s) 4))
                     (implies (and (d2-7a2-5p rn)
                                   (leq oplen 32))
                              (equal (read-rn oplen rn (mc-rfile sn))
                                     (read-rn oplen rn (mc-rfile s))))
                     (implies (disjoint x k (sub 32 12 (read-sp s)) 28)
                              (equal (read-mem x (mc-mem sn) k)
                                     (read-mem x (mc-mem s) k)))
                     (equal (iread-dn 32 0 sn) (bsearch x n lst))))))
     ((use (bsearch-s-s0p))
      (disable rts-addr linked-rts-addr linked-a6)))

(disable bsearch-t)

; in the logic, bsearch has these properties:
```

```
;      1. if it returns a nonnegative integer i, then lst[i] = x.
;      2. if it returns -1, then x is not in lst.
(defn member1 (x lst i j)
  (if (lessp i j)
      (if (equal x (get-nth i lst))
          t
        (member1 x lst (add1 i) j))
    f)
  ((lessp (difference j i))))

(defn orderedp (lst)
  (if (nlistp lst)
      t
    (if (nlistp (cdr lst))
        t
      (and (ileq (car lst) (cadr lst))
           (orderedp (cdr lst))))))

(prove-lemma leq-trans (rewrite)
     (implies (and (not (ilessp x y))
                   (not (ilessp y z)))
              (not (ilessp x z)))
     ((enable ilessp)))

(prove-lemma int-equal (rewrite)
     (implies (and (integerp x)
                   (integerp y)
                   (not (ilessp x y))
                   (not (ilessp y x)))
              (equal (equal x y) t))
     ((enable ilessp integerp)))

(prove-lemma orderedp-ordered (rewrite)
     (implies (and (orderedp lst)
                   (leq i j)
                   (lessp j (len lst)))
              (equal (ilessp (get-nth j lst) (get-nth i lst)) f))
     ((enable get-nth)))

(prove-lemma bsearch1-found (rewrite)
     (implies (and (not (equal (bsearch1 x lst i j) -1))
                   (lst-integerp lst)
                   (integerp x))
              (equal (get-nth (bsearch1 x lst i j) lst)
                     x))
     ((disable quotient remainder)))

(prove-lemma bsearch1-not-found-1 (rewrite)
     (implies (and (orderedp lst)
                   (ilessp (get-nth k lst) x)
                   (leq i k)
                   (lessp k j)
                   (leq j (len lst)))
              (equal (member1 x lst i j)
```

```
                  (member1 x lst (add1 k) j))))

(prove-lemma bsearch1-not-found-2-lemma (rewrite)
     (implies (and (orderedp lst)
                   (ilessp x (get-nth k lst))
                   (leq k i)
                   (lessp j (len lst)))
              (not (member1 x lst i j))))

(prove-lemma bsearch1-not-found-2 (rewrite)
     (implies (and (orderedp lst)
                   (ilessp x (get-nth k lst))
                   (lessp k j)
                   (leq j (len lst)))
              (equal (member1 x lst i j)
                     (member1 x lst i k))))

(disable bsearch1-not-found-2-lemma)

(prove-lemma bsearch1-not-found (rewrite)
     (implies (and (equal (bsearch1 x lst i j) -1)
                   (orderedp lst)
                   (lst-integerp lst)
                   (integerp x)
                   (leq j (len lst)))
              (not (member1 x lst i j)))
     ((induct (bsearch1 x lst i j))))

(defn member2 (x lst i j)
  (if (lessp i j)
      (if (equal x (get-nth i lst))
          t
        (member2 x (cdr lst) i (sub1 j)))
    f)
  ((lessp (difference j i))))

(prove-lemma member2-member ()
     (equal (member2 x lst 0 (len lst))
            (member x lst))
     ((enable get-nth)))

(prove-lemma member2-lemma (rewrite)
     (implies (not (equal x (get-nth i lst)))
              (equal (member2 x lst (add1 i) j)
                     (member2 x lst i j)))
     ((enable get-nth)))

(prove-lemma member1-member2 ()
     (equal (member1 x lst i j)
            (member2 x lst i j)))

(prove-lemma bsearch-found (rewrite)
     (implies (and (not (equal (bsearch x n lst) -1))
                   (lst-integerp lst)
```

```
                (integerp x))
           (equal (get-nth (bsearch x n lst) lst)
                  x)))

(prove-lemma bsearch-not-found (rewrite)
     (implies (and (equal (bsearch x (len lst) lst) -1)
                   (orderedp lst)
                   (lst-integerp lst)
                   (integerp x))
              (not (member x lst)))
     ((use (member1-member2 (i 0) (j (len lst)))
           (member2-member))))

; an upper bound.
(prove-lemma bsearch1-t-0 (rewrite)
     (equal (bsearch1-t x lst i i) 6)
     ((expand (bsearch1-t x lst i i))))

(prove-lemma bsearch1-t-crock (rewrite)
     (implies (and (lessp i j)
                   (not
                    (lessp
                     (plus 26
                           (times 13
                                  (log 2
                                       (difference (sub1 j)
                                                   (quotient (plus i j) 2)))))
                     (bsearch1-t x lst (add1 (quotient (plus i j) 2)) j))))
              (equal
               (lessp
                (plus 26 (times 13 (log 2 (difference j i))))
                (plus 13 (bsearch1-t x lst (add1 (quotient (plus i j) 2)) j)))
               f))
     ((use (log-leq (b 2)
                    (x (difference (sub1 j) (quotient (plus i j) 2)))
                    (y (quotient (difference j i) 2)))
           (mean-difference-2))))

(prove-lemma bsearch1-t-ubound (rewrite)
     (not (lessp (plus 26 (times 13 (log 2 (difference j i))))
                 (bsearch1-t x lst i j)))
     ((enable splus)))

(disable bsearch1-t-crock)

(prove-lemma bsearch-t-ubound-la ()
     (leq (bsearch-t x n lst)
          (plus 32 (times 13 (log 2 n))))
     ((enable splus bsearch-t)))

(prove-lemma bsearch-t-ubound ()
     (implies (lessp n (exp 2 31))
              (leq (bsearch-t x n lst) 435))
     ((use (ta-lemma-1 (x 32) (y 13) (a n) (a1 (exp 2 31)))
```

```
          (bsearch-t-ubound-la))))
```

C.4 Quick Sort

```
;           Proof of the Correctness of a Quicksort Program
;
#|
The following C function QSORT sorts a[left], ..., a[right] into increasing
order.  The program is a slightly modified version of K&R.

/* qsort: sort a[left]...a[right] into increasing order.  We use the middle */
/* element of each subarray for partitioning.   */
void qsort (int a[], int left, int right)
{
  int i, last, temp;

  if (left >= right)
    return;
  last = (left + right) / 2;
  temp = a[left];
  a[left] = a[last];
  a[last] = temp;
  last = left;
  for (i = left + 1; i<= right; i++)
    if (a[i] < a[left]){
      temp = a[++last];
      a[last] = a[i];
      a[i] = temp;
    };
  temp = a[left];
  a[left] = a[last];
  a[last] = temp;
  qsort(a, left, last-1);
  qsort(a, last+1, right);
}

Here is the MC68020 assembly code of the above QSORT program.  The code is
generated by "gcc -O".

0x22b8 <qsort>:         linkw fp,#0
0x22bc <qsort+4>:       moveml d2-d4/a2-a3,sp@-
0x22c0 <qsort+8>:       moveal fp@(8),a3
0x22c4 <qsort+12>:      movel fp@(12),d3
0x22c8 <qsort+16>:      movel fp@(16),d4
0x22cc <qsort+20>:      cmpl d3,d4
0x22ce <qsort+22>:      ble 0x2338 <qsort+128>
0x22d0 <qsort+24>:      movel d3,d2
0x22d2 <qsort+26>:      addl d4,d2
0x22d4 <qsort+28>:      bpl 0x22d8 <qsort+32>
0x22d6 <qsort+30>:      addql #1,d2
0x22d8 <qsort+32>:      asrl #1,d2
0x22da <qsort+34>:      movel 0(a3)[d3.l*4],d1
```

```
0x22de <qsort+38>:      movel 0(a3)[d2.l*4],0(a3)[d3.l*4]
0x22e4 <qsort+44>:      movel d1,0(a3)[d2.l*4]
0x22e8 <qsort+48>:      movel d3,d2
0x22ea <qsort+50>:      movel d2,d0
0x22ec <qsort+52>:      bra 0x2308 <qsort+80>
0x22ee <qsort+54>:      moveal 0(a3)[d0.l*4],a0
0x22f2 <qsort+58>:      cmpal 0(a3)[d3.l*4],a0
0x22f6 <qsort+62>:      bge 0x2308 <qsort+80>
0x22f8 <qsort+64>:      addql #1,d2
0x22fa <qsort+66>:      movel 0(a3)[d2.l*4],d1
0x22fe <qsort+70>:      movel 0(a3)[d0.l*4],0(a3)[d2.l*4]
0x2304 <qsort+76>:      movel d1,0(a3)[d0.l*4]
0x2308 <qsort+80>:      addql #1,d0
0x230a <qsort+82>:      cmpl d0,d4
0x230c <qsort+84>:      bge 0x22ee <qsort+54>
0x230e <qsort+86>:      movel 0(a3)[d3.l*4],d1
0x2312 <qsort+90>:      movel 0(a3)[d2.l*4],0(a3)[d3.l*4]
0x2318 <qsort+96>:      movel d1,0(a3)[d2.l*4]
0x231c <qsort+100>:     moveal d2,a0
0x231e <qsort+102>:     pea a0@(-1)
0x2322 <qsort+106>:     movel d3,sp@-
0x2324 <qsort+108>:     movel a3,sp@-
0x2326 <qsort+110>:     lea 0x22b8 <qsort>,a2
0x232a <qsort+114>:     jsr a2@
0x232c <qsort+116>:     movel d4,sp@-
0x232e <qsort+118>:     moveal d2,a0
0x2330 <qsort+120>:     pea a0@(1)
0x2334 <qsort+124>:     movel a3,sp@-
0x2336 <qsort+126>:     jsr a2@
0x2338 <qsort+128>:     moveml fp@(-20),d2-d4/a2-a3
0x233e <qsort+134>:     unlk fp
0x2340 <qsort+136>:     rts
```

The machine code of the above program is:

```
<qsort>:      0x4e56  0x0000  0x48e7  0x3830  0x266e  0x0008  0x262e  0x000c
<qsort+16>:   0x282e  0x0010  0xb883  0x6f68  0x2403  0xd484  0x6a02  0x5282
<qsort+32>:   0xe282  0x2233  0x3c00  0x27b3  0x2c00  0x3c00  0x2781  0x2c00
<qsort+48>:   0x2403  0x2002  0x601a  0x2073  0x0c00  0xb1f3  0x3c00  0x6c10
<qsort+64>:   0x5282  0x2233  0x2c00  0x27b3  0x0c00  0x2c00  0x2781  0x0c00
<qsort+80>:   0x5280  0xb880  0x6ce0  0x2233  0x3c00  0x27b3  0x2c00  0x3c00
<qsort+96>:   0x2781  0x2c00  0x2042  0x4868  0xffff  0x2f03  0x2f0b  0x45fa
<qsort+112>:  0xff90  0x4e92  0x2f04  0x2042  0x4868  0x0001  0x2f0b  0x4e92
<qsort+128>:  0x4cee  0x0c1c  0xffec  0x4e5e  0x4e75
```

In the Logic, it looks like:

```
'(78      86      0       0       72      231     56      48
  38      110     0       8       38      46      0       12
  40      46      0       16      184     131     111     104
  36      3       212     132     106     2       82      130
  226     130     34      51      60      0       39      179
  44      0       60      0       39      129     44      0
  36      3       32      2       96      26      32      115
```

```
  12      0       177     243     60      0       108     16
  82      130     34      51      44      0       39      179
  12      0       44      0       39      129     12      0
  82      128     184     128     108     224     34      51
  60      0       39      179     44      0       60      0
  39      129     44      0       32      66      72      104
  255     255     47      3       47      11      69      250
  255     144     78      146     47      4       32      66
  72      104     0       1       47      11      78      146
  76      238     12      28      255     236     78      94
  78      117)
|#

; in the Logic, the above program is defined by (qsort-code).
(defn qsort-code ()
  '(78      86      0       0       72      231     56      48
    38      110     0       8       38      46      0       12
    40      46      0       16      184     131     111     104
    36      3       212     132     106     2       82      130
    226     130     34      51      60      0       39      179
    44      0       60      0       39      129     44      0
    36      3       32      2       96      26      32      115
    12      0       177     243     60      0       108     16
    82      130     34      51      44      0       39      179
    12      0       44      0       39      129     12      0
    82      128     184     128     108     224     34      51
    60      0       39      179     44      0       60      0
    39      129     44      0       32      66      72      104
    255     255     47      3       47      11      69      250
    255     144     78      146     47      4       32      66
    72      104     0       1       47      11      78      146
    76      238     12      28      255     236     78      94
    78      117))

(prove-lemma ilessp-lessp (rewrite)
     (implies (numberp x)
              (equal (ilessp x y) (lessp x y))))

(disable ilessp)
(enable iplus)
(enable idifference)
(enable iquotient)

; qsort is a function in Nqthm that characterizes the functional semantics of
; the program (qsort-code).
(defn qpart-aux (l r lst last i)
  (if (lessp r i)
      (swap l last lst)
    (if (ilessp (get-nth i lst) (get-nth l lst))
        (qpart-aux l r (swap (add1 last) i lst) (add1 last) (add1 i))
      (qpart-aux l r lst last (add1 i))))
  ((lessp (difference (add1 r) i))))

(defn qpart (l r lst)
```

```
  (qpart-aux l r (swap l (quotient (plus l r) 2) lst) l (add1 l)))

(defn qlast-aux (l r lst last i)
  (if (lessp r i)
      (fix last)
    (if (ilessp (get-nth i lst) (get-nth l lst))
        (qlast-aux l r (swap (add1 last) i lst) (add1 last) (add1 i))
      (qlast-aux l r lst last (add1 i))))
  ((lessp (difference (add1 r) i))))

(defn qlast (l r lst)
  (qlast-aux l r (swap l (quotient (plus l r) 2) lst) l (add1 l)))

(prove-lemma qlast-aux-lb (rewrite)
     (implies (leq left last)
              (not (lessp (qlast-aux left right lst last i) left))))

(prove-lemma qlast-lb (rewrite)
     (not (lessp (qlast left right lst) left)))

(prove-lemma qlast-aux-ub (rewrite)
     (implies (and (lessp last i)
                   (leq i right))
              (not (lessp right (qlast-aux left right lst last i))))
     ((expand (qlast-aux left i lst last i)
              (qlast-aux left i lst last (add1 i))
              (qlast-aux left i lst (add1 last) (add1 i)))))

(prove-lemma qlast-ub (rewrite)
     (implies (lessp left right)
              (not (lessp right (qlast left right lst)))))

(disable qlast)
(disable qpart)

(defn qsort (l r lst)
  (if (lessp l r)
      (qsort (add1 (qlast l r lst))
             r
             (qsort l (sub1 (qlast l r lst)) (qpart l r lst)))
    lst)
  ((lessp (difference r l))))

; the computation time of the program.
(defn qpart-aux-t (a l r n lst last i)
  (if (lessp r i)
      11
    (if (ilessp (get-nth i lst) (get-nth l lst))
        (splus 10
               (qpart-aux-t a l r n (swap (add1 last) i lst)
                            (add1 last) (add1 i)))
      (splus 6 (qpart-aux-t a l r n lst last (add1 i)))))
  ((lessp (difference (add1 r) i))))
```

```
(defn qpart-t (a l r n lst)
  (let ((lst1 (swap l (quotient (plus l r) 2) lst)))
    (splus 18 (qpart-aux-t a l r n lst1 l (add1 l)))))

(defn qsort-10 (a l r n lst) 10)

(defn qsort-5 (a l r n lst) 5)

(defn qsort-3 (a l r n lst) 3)

(defn qsort-t (a l r n lst)
  (let ((last (qlast l r lst))
        (qlst (qpart l r lst)))
    (if (lessp l r)
        (splus (qpart-t a l r n lst)
               (splus (qsort-t a l (sub1 last) n qlst)
                      (splus (qsort-5 a l r n lst)
                             (splus (qsort-t a (add1 last) r n
                                             (qsort l (sub1 last) qlst))
                                    (qsort-3 a l r n lst)))))
      (qsort-10 a l r n lst)))
  ((lessp (difference r l))))

; an induction hint.
(defn qsort-induct (s a l r n lst)
  (let ((last (qlast l r lst))
        (qlst (qpart l r lst)))
    (if (lessp l r)
        (and (qsort-induct (stepn s (qpart-t a l r n lst))
                           a l (idifference last 1) n qlst)
             (qsort-induct
              (stepn s (splus (qpart-t a l r n lst)
                              (splus (qsort-t a l (sub1 last) n qlst)
                                     (qsort-5 a l r n lst))))
              a (add1 last) r n (qsort l (sub1 last) qlst)))
      t))
  ((lessp (difference r l))))

; the preconditions of the initial state.
(defn qstack (l r lst)
  (let ((last (qlast l r lst))
        (lst1 (qpart l r lst)))
    (if (lessp l r)
        (max (plus 40 (qstack l (sub1 last) lst1))
             (plus 52 (qstack (add1 last) r (qsort l (sub1 last) lst1))))
      68))
  ((lessp (difference r l))))

(prove-lemma qstack-la0 (rewrite)
     (not (lessp (qstack l r lst) 68)))

(prove-lemma qstack-la1 (rewrite)
     (let ((last (qlast l r lst))
           (lst1 (qpart l r lst)))
```

```
       (implies (lessp l r)
                (not (lessp (qstack l r lst)
                            (plus 40 (qstack l (sub1 last) lst1)))))))

(prove-lemma qstack-la2 (rewrite)
     (let ((last (qlast l r lst))
           (lst1 (qpart l r lst)))
       (implies (lessp l r)
                (not (lessp (qstack l r lst)
                            (plus 52 (qstack (add1 last)
                                             r
                                             (qsort l (sub1 last) lst1))))))))

; an upper bound of stack space.
(prove-lemma qstack-ubound-la-1 (rewrite)
     (implies
      (lessp l r)
      (not (lessp
            (times 52 (difference r l))
            (plus 52 (times 52 (difference (sub1 (qlast l r lst)) l)))))))

(prove-lemma qstack-ubound-la-2 (rewrite)
     (implies
      (lessp l r)
      (not (lessp
            (times 52 (difference r l))
            (plus 52 (times 52 (difference r (add1 (qlast l r lst)))))))))

(prove-lemma qstack-ubound ()
     (leq (qstack l r lst)
          (plus 68 (times 52 (difference r l))))
     ((disable difference)))

(disable qstack)

; the initial state.
(defn qsort-statep (s a l r n lst)
  (let ((sp (sub 32 (difference (qstack l r lst) 16) (read-sp s))))
    (and (equal (mc-status s) 'running)
         (evenp (mc-pc s))
         (rom-addrp (mc-pc s) (mc-mem s) 138)
         (mcode-addrp (mc-pc s) (mc-mem s) (qsort-code))
         (ram-addrp a (mc-mem s) (times 4 n))
         (mem-ilst 4 a (mc-mem s) n lst)
         (ram-addrp sp (mc-mem s) (qstack l r lst))
         (disjoint a (times 4 n) sp (qstack l r lst))
         (equal a (read-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
         (equal l (iread-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
         (equal r (iread-mem (add 32 (read-sp s) 12) (mc-mem s) 4))
         (lessp (qstack l r lst) (exp 2 32))
         (numberp l)
         (lessp r n)
         (uint-rangep (times 4 n) 32))))
```

```
(defn qsort-sp (s a l r n lst)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (mc-pc s) (mc-mem s) 138)
       (mcode-addrp (mc-pc s) (mc-mem s) (qsort-code))
       (ram-addrp a (mc-mem s) (times 4 n))
       (mem-ilst 4 a (mc-mem s) n lst)
       (equal a (read-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal l (iread-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
       (equal r (iread-mem (add 32 (read-sp s) 12) (mc-mem s) 4))
       (numberp l)
       (numberp n)
       (lessp r n)
       (int-rangep n 32)
       (int-rangep (times 2 r) 32)
       (uint-rangep (times 4 n) 32)
       (ram-addrp (sub 32 52 (read-sp s)) (mc-mem s) 68)
       (disjoint a (times 4 n) (sub 32 52 (read-sp s)) 68)))

(prove-lemma qsort-statep-sp (rewrite)
     (implies (qsort-statep s a l r n lst)
              (qsort-sp s a l r n lst))
     ((enable int-rangep)))

; the conditions of the intermediate state s0.  s0 is the state of the
; machine right before the for statement in the corresponding C program.
(defn qsort-s0p (s s0 a l r n lst last i)
  (and (equal (linked-rts-addr s0) (rts-addr s))
       (equal (linked-a6 s0) (read-an 32 6 s))
       (equal* (read-rn 32 14 (mc-rfile s0)) (sub 32 4 (read-sp s)))
       (equal* (read-rn 32 15 (mc-rfile s0)) (sub 32 24 (read-sp s)))
       (equal (movem-saved s0 4 20 5)
              (readm-rn 32 '(2 3 4 10 11) (mc-rfile s)))
       (equal (mc-status s0) 'running)
       (evenp (mc-pc s0))
       (rom-addrp (sub 32 82 (mc-pc s0)) (mc-mem s0) 138)
       (mcode-addrp (sub 32 82 (mc-pc s0)) (mc-mem s0) (qsort-code))
       (ram-addrp (sub 32 52 (read-sp s)) (mc-mem s0) 68)
       (ram-addrp a (mc-mem s0) (times 4 n))
       (mem-ilst 4 a (mc-mem s0) n lst)
       (disjoint a (times 4 n) (sub 32 52 (read-sp s)) 68)
       (equal a (read-an 32 3 s0))
       (equal l (nat-to-int (read-rn 32 3 (mc-rfile s0)) 32))
       (equal r (nat-to-int (read-rn 32 4 (mc-rfile s0)) 32))
       (equal last (nat-to-int (read-rn 32 2 (mc-rfile s0)) 32))
       (equal i (nat-to-int (read-rn 32 0 (mc-rfile s0)) 32))
       (numberp l)
       (numberp r)
       (numberp n)
       (numberp last)
       (numberp i)
       (lessp l r)
       (lessp last i)
       (leq i (add1 r))
```

```
       (lessp r n)
       (int-rangep n 32)
       (uint-rangep (times 4 n) 32)))

; the conditions of the intermediate state s1.  s1 is the machine
; state after the execution of the first JSR instruction, but before
; the recursive execution of QSORT.
(defn qsort-s1p (s s1 a l r n lst last)
  (and (equal (linked-rts-addr s1) (rts-addr s))
       (equal (linked-a6 s1) (read-an 32 6 s))
       (equal* (read-rn 32 14 (mc-rfile s1)) (sub 32 4 (read-sp s)))
       (equal* (read-rn 32 15 (mc-rfile s1)) (sub 32 40 (read-sp s)))
       (equal (movem-saved s1 4 20 5)
              (readm-rn 32 '(2 3 4 10 11) (mc-rfile s)))
       (equal (mc-status s1) 'running)
       (equal (mc-pc s1) (read-an 32 2 s1))
       (evenp (read-an 32 2 s1))
       (evenp (rts-addr s1))
       (rom-addrp (read-an 32 2 s1) (mc-mem s1) 138)
       (mcode-addrp (read-an 32 2 s1) (mc-mem s1) (qsort-code))
       (rom-addrp (sub 32 116 (rts-addr s1)) (mc-mem s1) 138)
       (mcode-addrp (sub 32 116 (rts-addr s1)) (mc-mem s1) (qsort-code))
       (ram-addrp (sub 32 52 (read-sp s)) (mc-mem s1) 68)
       (ram-addrp a (mc-mem s1) (times 4 n))
       (mem-ilst 4 a (mc-mem s1) n lst)
       (disjoint a (times 4 n) (sub 32 52 (read-sp s)) 68)
       (equal a (read-an 32 3 s1))
       (equal l (nat-to-int (read-rn 32 3 (mc-rfile s1)) 32))
       (equal r (nat-to-int (read-rn 32 4 (mc-rfile s1)) 32))
       (equal last (nat-to-int (read-rn 32 2 (mc-rfile s1)) 32))
       (equal a (read-mem (add 32 (read-sp s1) 4) (mc-mem s1) 4))
       (equal l (iread-mem (add 32 (read-sp s1) 8) (mc-mem s1) 4))
       (equal (iread-mem (add 32 (read-sp s1) 12) (mc-mem s1) 4)
              (idifference last 1))
       (numberp l)
       (numberp r)
       (numberp last)
       (lessp l r)
       (leq last r)
       (lessp r n)
       (int-rangep n 32)
       (uint-rangep (times 4 n) 32)))

; the conditions of the intermediate state s2.  s2 is the machine
; state right after the first recursive call to QSORT.
(defn qsort-s2p (s s2 a l r n lst last)
  (and (equal (linked-rts-addr s2) (rts-addr s))
       (equal (linked-a6 s2) (read-an 32 6 s))
       (equal* (read-rn 32 14 (mc-rfile s2)) (sub 32 4 (read-sp s)))
       (equal* (read-rn 32 15 (mc-rfile s2)) (sub 32 36 (read-sp s)))
       (equal (movem-saved s2 4 20 5)
              (readm-rn 32 '(2 3 4 10 11) (mc-rfile s)))
       (equal (mc-status s2) 'running)
       (evenp (mc-pc s2))
```

```
       (evenp (read-an 32 2 s2))
       (rom-addrp (read-an 32 2 s2) (mc-mem s2) 138)
       (mcode-addrp (read-an 32 2 s2) (mc-mem s2) (qsort-code))
       (rom-addrp (sub 32 116 (mc-pc s2)) (mc-mem s2) 138)
       (mcode-addrp (sub 32 116 (mc-pc s2)) (mc-mem s2) (qsort-code))
       (ram-addrp (sub 32 16 (read-sp s2)) (mc-mem s2) 68)
       (ram-addrp a (mc-mem s2) (times 4 n))
       (mem-ilst 4 a (mc-mem s2) n lst)
       (disjoint a (times 4 n) (sub 32 16 (read-sp s2)) 68)
       (equal a (read-an 32 3 s2))
       (equal l (nat-to-int (read-rn 32 3 (mc-rfile s2)) 32))
       (equal r (nat-to-int (read-rn 32 4 (mc-rfile s2)) 32))
       (equal last (nat-to-int (read-rn 32 2 (mc-rfile s2)) 32))
       (numberp l)
       (numberp r)
       (numberp last)
       (lessp l r)
       (leq last r)
       (lessp r n)
       (int-rangep n 32)
       (uint-rangep (times 4 n) 32)))

; the conditions of the intermediate state s3.  s3 is the machine
; state right after the execution of the second JSR instruction, but
; before the execution of QSORT.
(defn qsort-s3p (s s3 a l r n lst last)
  (and (equal (linked-rts-addr s3) (rts-addr s))
       (equal (linked-a6 s3) (read-an 32 6 s))
       (equal* (read-rn 32 14 (mc-rfile s3)) (sub 32 4 (read-sp s)))
       (equal* (read-rn 32 15 (mc-rfile s3)) (sub 32 52 (read-sp s)))
       (equal (movem-saved s3 4 20 5)
              (readm-rn 32 '(2 3 4 10 11) (mc-rfile s)))
       (equal (mc-status s3) 'running)
       (equal (mc-pc s3) (read-an 32 2 s3))
       (evenp (read-an 32 2 s3))
       (evenp (rts-addr s3))
       (rom-addrp (read-an 32 2 s3) (mc-mem s3) 138)
       (mcode-addrp (read-an 32 2 s3) (mc-mem s3) (qsort-code))
       (rom-addrp (sub 32 128 (rts-addr s3)) (mc-mem s3) 138)
       (mcode-addrp (sub 32 128 (rts-addr s3)) (mc-mem s3) (qsort-code))
       (ram-addrp (read-sp s3) (mc-mem s3) 68)
       (ram-addrp a (mc-mem s3) (times 4 n))
       (mem-ilst 4 a (mc-mem s3) n lst)
       (disjoint a (times 4 n) (read-sp s3) 68)
       (equal a (read-an 32 3 s3))
       (equal l (nat-to-int (read-rn 32 3 (mc-rfile s3)) 32))
       (equal r (nat-to-int (read-rn 32 4 (mc-rfile s3)) 32))
       (equal last (nat-to-int (read-rn 32 2 (mc-rfile s3)) 32))
       (equal a (read-mem (add 32 (read-sp s3) 4) (mc-mem s3) 4))
       (equal (iread-mem (add 32 (read-sp s3) 8) (mc-mem s3) 4) (add1 last))
       (equal r (iread-mem (add 32 (read-sp s3) 12) (mc-mem s3) 4))
       (numberp l)
       (numberp r)
       (numberp last)
```

```
       (lessp l r)
       (leq last r)
       (lessp r n)
       (int-rangep n 32)
       (uint-rangep (times 4 n) 32)))

; the conditions of the intermediate state s4.  s4 is the machine
; state right after the second recursive call to QSORT.
(defn qsort-s4p (s s4 a l r n lst)
  (and (equal (linked-rts-addr s4) (rts-addr s))
       (equal (linked-a6 s4) (read-an 32 6 s))
       (equal* (read-rn 32 14 (mc-rfile s4)) (sub 32 4 (read-sp s)))
       (equal* (read-rn 32 15 (mc-rfile s4)) (sub 32 48 (read-sp s)))
       (equal (movem-saved s4 4 20 5)
              (readm-rn 32 '(2 3 4 10 11) (mc-rfile s)))
       (equal (mc-status s4) 'running)
       (evenp (mc-pc s4))
       (rom-addrp (sub 32 128 (mc-pc s4)) (mc-mem s4) 138)
       (mcode-addrp (sub 32 128 (mc-pc s4)) (mc-mem s4) (qsort-code))
       (ram-addrp (sub 32 48 (read-an 32 6 s4)) (mc-mem s4) 68)
       (ram-addrp a (mc-mem s4) (times 4 n))
       (mem-ilst 4 a (mc-mem s4) n lst)
       (disjoint a (times 4 n) (sub 32 48 (read-an 32 6 s4)) 68)))

; the conditions of the final state.  s5 is the machine state after
; the execution of this QSORT program.
(defn qsort-s5p (s s5 a l r n lst)
  (and (equal (mc-status s5) 'running)
       (equal (mc-pc s5) (rts-addr s))
       (equal (read-rn 32 14 (mc-rfile s5)) (read-rn 32 14 (mc-rfile s)))
       (equal (read-rn 32 15 (mc-rfile s5)) (add 32 (read-sp s) 4))
       (mem-ilst 4 a (mc-mem s5) n lst)
       (equal (read-dn 32 2 s5) (read-dn 32 2 s))
       (equal (read-dn 32 3 s5) (read-dn 32 3 s))
       (equal (read-dn 32 4 s5) (read-dn 32 4 s))
       (equal (read-an 32 2 s5) (read-an 32 2 s))
       (equal (read-an 32 3 s5) (read-an 32 3 s))))

(defn-sk qsort-sk (s s5 a l r n lst)
  (forall
   (x k)
   (let ((sp (sub 32 (difference (qstack l r lst) 16) (read-sp s))))
     (implies
      (and (disjoint sp (qstack l r lst) x k)
           (disjoint a (times 4 n) x k))
      (equal (read-mem x (mc-mem s5) k)
             (read-mem x (mc-mem s) k))))))

(disable qsort-sk)

(prove-lemma qsort-sk-1 (rewrite)
  (let ((sp (sub 32 (difference (qstack l r lst) 16) (read-sp s))))
    (implies (and (qsort-sk s s5 a l r n lst)
                  (disjoint sp (qstack l r lst) x k)
```

```
                (disjoint a (times 4 n) x k))
             (equal (read-mem x (mc-mem s5) k)
                    (read-mem x (mc-mem s) k))))
  ((use (qsort-sk))))

(prove-lemma qsort-sk-2 (rewrite)
  (let ((sp (sub 32 (difference (qstack l r lst) 16) (read-sp s))))
    (implies (and (qsort-sk s s5 a l r n lst)
                  (disjoint sp (qstack l r lst) x (times opsz k))
                  (disjoint a (times 4 n) x (times opsz k)))
             (equal (readm-mem opsz x (mc-mem s5) k)
                    (readm-mem opsz x (mc-mem s) k))))
  ((induct (readm-mem opsz x mem k))))

(disable qsort-sk-1)
(disable qsort-sk-2)

; base case: from the initial state to exit (s --> exit).
(prove-lemma qsort-base (rewrite)
     (implies (and (qsort-sp s a l r n lst)
                   (not (lessp l r)))
              (qsort-s5p s (stepn s (qsort-10 a l r n lst)) a l r n lst)))

(prove-lemma qsort-base-rfile (rewrite)
     (implies
      (and (qsort-sp s a l r n lst)
           (not (lessp l r))
           (d2-7a2-5p rn)
           (leq oplen 32))
      (equal (read-rn oplen rn (mc-rfile (stepn s (qsort-10 a l r n lst))))
             (read-rn oplen rn (mc-rfile s)))))

(prove-lemma qsort-base-mem (rewrite)
     (implies (and (qsort-sp s a l r n lst)
                   (not (lessp l r))
                   (disjoint (sub 32 24 (read-sp s)) 40 x k))
              (equal (read-mem x (mc-mem (stepn s 10)) k)
                     (read-mem x (mc-mem s) k))))

(prove-lemma qsort-base-mem-sk (rewrite)
     (implies (and (qsort-sp s a l r n lst)
                   (not (lessp l r)))
              (qsort-sk s (stepn s (qsort-10 a l r n lst)) a l r n lst))
     ((use (qsort-sk (s5 (stepn s 10))))
      (disable qsort-sp)))

; induction case: s --> s0 --> s1 --> s2 --> s3 --> s4 --> exit.
; s --> s0:
(prove-lemma add1-int-rangep (rewrite)
             (implies (lessp x (nat-to-int y n))
                      (int-rangep (add1 x) n))
             ((enable int-rangep nat-to-int)))

(prove-lemma mean-bounds (rewrite)
```

```
     (implies (lessp i j)
              (and (lessp (quotient (plus i j) 2) j)
                   (not (lessp (quotient (plus i j) 2) i)))))

(prove-lemma int-rangep-plus-1 (rewrite)
     (implies (and (int-rangep (times 2 j) n)
                   (lessp i j))
              (int-rangep (plus i j) n)))

(prove-lemma qsort-s-s0 (rewrite)
     (let ((lst1 (swap l (quotient (plus l r) 2) lst)))
       (implies (and (qsort-sp s a l r n lst)
                     (lessp l r))
                (qsort-s0p s (stepn s 18) a l r n lst1 l (add1 l))))
     ((disable times quotient)))

(prove-lemma qsort-s-s0-rfile (rewrite)
     (implies (and (qsort-sp s a l r n lst)
                   (lessp l r)
                   (d5-7a4-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 18)))
                     (read-rn oplen rn (mc-rfile s))))
     ((disable times quotient)))

(prove-lemma qsort-s-s0-mem (rewrite)
     (implies (and (qsort-sp s a l r n lst)
                   (lessp l r)
                   (disjoint (sub 32 52 (read-sp s)) 68 x k)
                   (disjoint a (times 4 n) x k))
              (equal (read-mem x (mc-mem (stepn s 18)) k)
                     (read-mem x (mc-mem s) k)))
     ((disable times quotient)))

; s0 --> s1:
; base case (s0 --> s1):
(prove-lemma qsort-s0-s1 (rewrite)
     (implies (and (qsort-s0p s s0 a l r n lst last i)
                   (lessp r i)
                   (equal last1 (fix last)))
              (qsort-s1p s (stepn s0 11) a l r n (swap l last lst) last1))
     ((disable times lessp)))

(prove-lemma qsort-s0-s1-rfile (rewrite)
     (implies (and (qsort-s0p s s0 a l r n lst last i)
                   (lessp r i)
                   (d5-7a4-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s0 11)))
                     (read-rn oplen rn (mc-rfile s0))))
     ((disable times lessp)))

(prove-lemma qsort-s0-s1-mem (rewrite)
     (implies (and (qsort-s0p s s0 a l r n lst last i)
                   (lessp r i)
                   (disjoint (sub 32 52 (read-sp s)) 68 x k)
```

```
                (disjoint a (times 4 n) x k))
           (equal (read-mem x (mc-mem (stepn s0 11)) k)
                  (read-mem x (mc-mem s0) k)))
     ((disable times lessp)))

; induction case (s0 --> s0):
(prove-lemma add1-int-rangepxx (rewrite)
             (implies (and (leq i r)
                           (lessp r n)
                           (int-rangep n 32))
                      (int-rangep (add1 i) 32))
             ((enable int-rangep nat-to-int)))

(prove-lemma qsort-s0-s0-1 (rewrite)
     (implies (and (qsort-s0p s s0 a l r n lst last i)
                   (not (lessp r i))
                   (not (ilessp (get-nth i lst) (get-nth l lst))))
              (qsort-s0p s (stepn s0 6) a l r n lst last (add1 i)))
     ((disable times lessp)))

(prove-lemma qsort-s0-s0-2 (rewrite)
     (implies (and (qsort-s0p s s0 a l r n lst last i)
                   (not (lessp r i))
                   (ilessp (get-nth i lst) (get-nth l lst)))
              (qsort-s0p s (stepn s0 10) a l r n (swap (add1 last) i lst)
                         (add1 last) (add1 i)))
     ((disable times lessp)))

(prove-lemma qsort-s0-s0-rfile-1 (rewrite)
     (implies (and (qsort-s0p s s0 a l r n lst last i)
                   (not (lessp r i))
                   (not (ilessp (get-nth i lst) (get-nth l lst)))
                   (d5-7a4-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s0 6)))
                     (read-rn oplen rn (mc-rfile s0))))
     ((disable times lessp)))

(prove-lemma qsort-s0-s0-rfile-2 (rewrite)
     (implies (and (qsort-s0p s s0 a l r n lst last i)
                   (not (lessp r i))
                   (ilessp (get-nth i lst) (get-nth l lst))
                   (d5-7a4-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s0 10)))
                     (read-rn oplen rn (mc-rfile s0))))
     ((disable times lessp)))

(prove-lemma qsort-s0-s0-mem-1 (rewrite)
     (implies (and (qsort-s0p s s0 a l r n lst last i)
                   (not (lessp r i))
                   (not (ilessp (get-nth i lst) (get-nth l lst))))
              (equal (read-mem x (mc-mem (stepn s0 6)) k)
                     (read-mem x (mc-mem s0) k)))
     ((disable times lessp)))
```

```
(prove-lemma qsort-s0-s0-mem-2 (rewrite)
     (implies (and (qsort-s0p s s0 a l r n lst last i)
                   (not (lessp r i))
                   (ilessp (get-nth i lst) (get-nth l lst))
                   (disjoint a (times 4 n) x k))
              (equal (read-mem x (mc-mem (stepn s0 10)) k)
                     (read-mem x (mc-mem s0) k)))
     ((disable times lessp)))

; induction hint for the partition.
(defn qpart-induct (s l r lst last i)
  (if (lessp r i)
      t
    (if (ilessp (get-nth i lst) (get-nth l lst))
        (qpart-induct (stepn s 10) l r (swap (add1 last) i lst)
                      (add1 last) (add1 i))
      (qpart-induct (stepn s 6) l r lst last (add1 i))))
  ((lessp (difference (add1 r) i))))

(prove-lemma qpart-aux-ct (rewrite)
     (implies (qsort-s0p s s0 a l r n lst last i)
              (qsort-s1p s (stepn s0 (qpart-aux-t a l r n lst last i))
                         a l r n (qpart-aux l r lst last i)
                         (qlast-aux l r lst last i)))
     ((induct (qpart-induct s0 l r lst last i))
      (disable qsort-s0p qsort-s1p swap)))

(prove-lemma qsort-s-s1 (rewrite)
     (implies (and (qsort-sp s a l r n lst)
                   (lessp l r))
              (qsort-s1p s (stepn s (qpart-t a l r n lst)) a l r n
                         (qpart l r lst) (qlast l r lst)))
     ((enable qpart qlast)
      (disable swap qsort-sp qsort-s0p qsort-s1p)))

(prove-lemma qpart-aux-rfile (rewrite)
     (let ((s1 (stepn s0 (qpart-aux-t a l r n lst last i))))
       (implies (and (qsort-s0p s s0 a l r n lst last i)
                     (d5-7a4-5p rn))
                (equal (read-rn oplen rn (mc-rfile s1))
                       (read-rn oplen rn (mc-rfile s0))))))
     ((induct (qpart-induct s0 l r lst last i))
      (disable swap qsort-s0p)))

(prove-lemma qsort-s-s1-rfile (rewrite)
     (implies
      (and (qsort-sp s a l r n lst)
           (lessp l r)
           (d5-7a4-5p rn))
      (equal (read-rn oplen rn (mc-rfile (stepn s (qpart-t a l r n lst))))
             (read-rn oplen rn (mc-rfile s))))
     ((use (qsort-s-s0))
      (disable qsort-sp qsort-s0p)))
```

```
(prove-lemma qpart-aux-mem (rewrite)
     (let ((s1 (stepn s0 (qpart-aux-t a l r n lst last i))))
       (implies (and (qsort-s0p s s0 a l r n lst last i)
                     (disjoint (sub 32 52 (read-sp s)) 68 x k)
                     (disjoint a (times 4 n) x k))
                (equal (read-mem x (mc-mem s1) k)
                       (read-mem x (mc-mem s0) k))))
     ((induct (qpart-induct s0 l r lst last i))
      (disable swap qsort-s0p)))

(prove-lemma qsort-sk-s-s1 (rewrite)
     (implies (and (qsort-sp s a l r n lst)
                   (lessp l r))
              (qsort-sk s (stepn s (qpart-t a l r n lst)) a l r n lst))
     ((use (qsort-sk (s5 (stepn s (qpart-t a l r n lst))))
           (qsort-s-s0))
      (disable swap qsort-sp qsort-s0p qsort-s-s0)))

(disable qpart-t)

; s2 --> s3:
(prove-lemma qsort-s2-s3 (rewrite)
     (let ((qlst (qpart l r lst))
           (last (qlast l r lst)))
       (implies (qsort-s2p s s2 a l r n (qsort l (sub1 last) qlst) last)
                (qsort-s3p s (stepn s2 (qsort-5 a l r n lst))
                           a l r n (qsort l (sub1 last) qlst) last))))

(prove-lemma qsort-s2-s3-rfile-la (rewrite)
     (implies (and (qsort-s2p s s2 a l r n lst1 last)
                   (d5-7a4-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s2 5)))
                     (read-rn oplen rn (mc-rfile s2)))))

(prove-lemma qsort-s2-s3-rfile (rewrite)
     (let ((qlst (qpart l r lst))
           (last (qlast l r lst))
           (s2 (stepn (stepn s k1) k2)))
       (implies (and (qsort-s2p s s2 a l r n (qsort l (sub1 last) qlst) last)
                     (d5-7a4-5p rn))
                (equal (read-rn oplen rn
                                (mc-rfile (stepn s2 (qsort-5 a l r n lst))))
                       (read-rn oplen rn (mc-rfile s2)))))
     ((disable qsort-s2p stepn-rewriter)))

(prove-lemma qsort-s2-s3-mem (rewrite)
     (implies (and (qsort-s2p s s2 a l r n lst last)
                   (disjoint (sub 32 52 (read-sp s)) 68 x k))
              (equal (read-mem x (mc-mem (stepn s2 5)) k)
                     (read-mem x (mc-mem s2) k))))

(prove-lemma qsort-sk-s-s3 (rewrite)
     (let ((lst1 (qpart l r lst))
           (last (qlast l r lst)))
```

```
       (implies (and (qsort-s2p s s2 a l r n (qsort l (sub1 last) lst1) last)
                     (qsort-sk s s2 a l r n lst))
                (qsort-sk s (stepn s2 (qsort-5 a l r n lst)) a l r n lst)))
     ((use (qsort-sk (s5 (stepn s2 5))))
      (enable qsort-sk-1) (disable qsort-s2p)))

; s4 --> exit:
(prove-lemma qsort-s4-s5 (rewrite)
     (let ((qlst (qpart l r lst))
           (last (qlast l r lst)))
       (implies (qsort-s4p s s4 a l r n
                           (qsort (add1 last) r (qsort l (sub1 last) qlst)))
                (qsort-s5p s (stepn s4 (qsort-3 a l r n lst)) a l r n
                           (qsort (add1 last) r (qsort l (sub1 last) qlst))))))

(prove-lemma qsort-s4-s5-rfile-la (rewrite)
     (implies (and (qsort-s4p s s4 a l r n (qsort l r lst))
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s4 3)))
                     (if (d5-7a4-5p rn)
                         (read-rn oplen rn (mc-rfile s4))
                       (read-rn oplen rn (mc-rfile s))))))

(prove-lemma qsort-s4-s5-rfile (rewrite)
     (let ((s4 (stepn (stepn (stepn (stepn s k1) k2) k3) k4)))
       (implies
        (and (qsort-s4p s s4 a l r n (qsort l r lst))
             (leq oplen 32)
             (d2-7a2-5p rn))
        (equal (read-rn oplen rn (mc-rfile (stepn s4 (qsort-3 a l r n lst))))
               (if (d5-7a4-5p rn)
                 (read-rn oplen rn (mc-rfile s4))
               (read-rn oplen rn (mc-rfile s)))))))
     ((disable qsort-s4p)))

(prove-lemma qsort-s4-s5-mem (rewrite)
     (implies (and (qsort-s4p s s4 a l r n lst)
                   (disjoint (sub 32 52 (read-sp s)) 68 x k))
              (equal (read-mem x (mc-mem (stepn s4 3)) k)
                     (read-mem x (mc-mem s4) k))))

(prove-lemma qsort-sk-s-s5 (rewrite)
     (implies (and (qsort-s4p s s4 a l r n (qsort l r lst))
                   (qsort-sk s s4 a l r n lst))
              (qsort-sk s (stepn s4 (qsort-3 a l r n lst)) a l r n lst))
     ((use (qsort-sk (s5 (stepn s4 3))))
      (enable qsort-sk-1) (disable qsort-s4p)))

; some auxiliary lemmas.
(prove-lemma qlast-aux-0 (rewrite)
     (implies (lessp r i)
              (equal (qlast-aux l r lst last i) (fix last))))
```

```
(prove-lemma qlast-0 (rewrite)
     (implies (leq r l)
              (equal (qlast l r lst) (fix l)))
     ((enable qlast)))

(prove-lemma qstack-la3 (rewrite)
     (implies (not (lessp l r))
              (equal (qstack l r lst) 68))
     ((enable qstack)))

(prove-lemma qstack-0 (rewrite)
     (and (equal (qstack l -1 lst) 68)
          (equal (qstack l 0 lst) 68)
          (equal (qstack l 1 lst) 68))
     ((enable qstack)))

(prove-lemma qstack-la4 (rewrite)
     (let ((last (qlast l r lst))
           (lst1 (qpart l r lst)))
       (not (lessp (qstack l r lst)
                   (qstack l (sub1 last) lst1))))
     ((expand (qstack l r lst))))

; s1 --> s2:  the first recursive call.
(prove-lemma qsort-s1-crock (rewrite)
     (implies (and (leq (plus q1 40) q)
                   (leq 68 q1)
                   (lessp q (exp 2 32)))
              (not (lessp q (plus q1 (add 32 4294967256
                                          (add 32
                                               (neg 32 (difference q1 16))
                                               (difference q 16)))))))
     ((enable neg nat-rangep add-nat-la sub-nat-la)
      (disable sub-neg)))

(prove-lemma qstack-la1~ (rewrite)
     (implies (and (equal last (qlast l r lst))
                   (equal lst1 (qpart l r lst))
                   (lessp l r))
              (not (lessp (qstack l r lst)
                          (plus 40 (qstack l (sub1 last) lst1))))))

(prove-lemma qsort-s1-s (rewrite)
     (let ((lst1 (qpart l r lst))
           (last (qlast l r lst)))
       (implies (and (qsort-s1p s (stepn s k) a l r n lst1 last)
                     (qsort-statep s a l r n lst))
                (qsort-statep (stepn s k) a l (idifference last 1) n lst1)))
     ((disable add-commutativity1 qstack-la3)))

(prove-lemma qsort-s1-crock1 (rewrite)
   (implies (and (leq 68 q1)
                 (lessp (plus 40 q1) 4294967296)
                 (leq (plus x k) 4294967296)
```

```
               (leq 4294967272 x))
           (disjoint
            (add 32 4294967256 (neg 32 (difference q1 16))) q1 x k))
   ((enable disjoint-leq disjoint-leq1)))

(prove-lemma qsort-s1-crock2-crock (rewrite)
     (implies (and (leq 68 q1)
                   (lessp q1 4294967256))
              (not (lessp (add 32 4294967256
                                  (neg 32 (difference q1 16)))
                          16)))
     ((enable neg nat-rangep add-nat-la sub-nat-la)
      (disable sub-neg)))

(prove-lemma qsort-s1-crock2 (rewrite)
   (implies (and (leq 68 q1)
                 (lessp (plus 40 q1) 4294967296)
                 (leq (plus x k) 16))
            (disjoint
             (add 32 4294967256 (neg 32 (difference q1 16))) q1 x k))
   ((enable disjoint-leq disjoint-leq1)))

(disable qsort-s1-crock2-crock)

(prove-lemma qsort-s1-s2 (rewrite)
     (let ((lst1 (qpart l r lst))
           (last (qlast l r lst)))
       (implies
        (and (qsort-s1p s s1 a l r n lst1 last)
             (qsort-statep s a l r n lst)
             (qsort-s5p s1 (stepn s1 k) a l (idifference last 1) n
                        (qsort l (sub1 last) lst1))
             (qsort-sk s1 (stepn s1 k) a l (idifference last 1) n lst1))
         (qsort-s2p s (stepn s1 k) a l r n (qsort l (sub1 last) lst1) last)))
     ((enable qsort-sk-1 qsort-sk-2)
      (disable add-commutativity1)))

(disable qsort-s1-crock1)
(disable qsort-s1-crock2)

(prove-lemma qsort-s1-s2-mem (rewrite)
     (let ((lst1 (qpart l r lst))
           (last (qlast l r lst)))
       (implies (and (qsort-s1p s s1 a l r n lst1 last)
                     (qsort-statep s a l r n lst)
                     (qsort-sk s s1 a l r n lst)
                     (qsort-sk s1 (stepn s1 k) a l (idifference last 1) n lst1)
                     (disjoint a (times 4 n) x k1)
                     (disjoint (sub 32 (difference (qstack l r lst) 16)
                                    (read-sp s))
                               (qstack l r lst) x k1))
                (equal (read-mem x (mc-mem (stepn s1 k)) k1)
                       (read-mem x (mc-mem s) k1))))
     ((enable qsort-sk-1)
```

```
      (disable add-commutativity1 qstack-la3)))

(disable qstack-la1~)

(prove-lemma qsort-sk-s-s2 (rewrite)
     (let ((lst1 (qpart l r lst))
           (last (qlast l r lst)))
       (implies
        (and (qsort-s1p s s1 a l r n lst1 last)
             (qsort-statep s a l r n lst)
             (qsort-sk s s1 a l r n lst)
             (qsort-sk s1 (stepn s1 k) a l (idifference last 1) n lst1))
         (qsort-sk s (stepn s1 k) a l r n lst)))
     ((use (qsort-sk (s5 (stepn s1 k))))
      (disable qsort-s1p qsort-statep idifference)))

(disable qsort-s1-s2-mem)

; s3 --> s4:  the second recursive call.
(prove-lemma qsort-s3-crock (rewrite)
     (implies (and (leq (plus q1 52) q)
                   (leq 68 q1)
                   (lessp q (exp 2 32)))
              (not (lessp q (plus q1 (add 32 4294967244
                                          (add 32
                                               (neg 32 (difference q1 16))
                                               (difference q 16)))))))
     ((enable neg nat-rangep add-nat-la sub-nat-la)
      (disable sub-neg)))

(prove-lemma qstack-la2~ (rewrite)
     (implies (and (equal last (qlast l r lst))
                   (equal lst1 (qpart l r lst))
                   (lessp l r))
              (not (lessp (qstack l r lst)
                          (plus 52 (qstack (add1 last)
                                           r
                                           (qsort l (sub1 last) lst1)))))))

(prove-lemma qsort-s3-s-la (rewrite)
     (let ((lst1 (qsort l (sub1 (qlast l r lst)) (qpart l r lst)))
           (last (qlast l r lst)))
       (implies (and (qsort-s3p s (stepn s k) a l r n lst1 last)
                     (qsort-statep s a l r n lst))
                (qsort-statep (stepn s k) a (add1 last) r n lst1)))
     ((disable add-commutativity1 qstack-la3)))

(prove-lemma qsort-s3-s (rewrite)
     (let ((lst1 (qsort l (sub1 (qlast l r lst)) (qpart l r lst)))
           (last (qlast l r lst)))
       (implies (and (qsort-s3p s (stepn (stepn (stepn s k1) k2) k3)
                                a l r n lst1 last)
                     (qsort-statep s a l r n lst))
                (qsort-statep (stepn (stepn (stepn s k1) k2) k3)
```

```
                            a (add1 last) r n lst1)))
     ((use (qsort-s3-s-la (k (splus k1 (splus k2 k3)))))
      (disable qsort-s3p qsort-statep)))

(prove-lemma qsort-s3-crock1 (rewrite)
   (implies (and (leq 68 q1)
                 (lessp (plus 52 q1) 4294967296)
                 (leq (plus x k) 4294967296)
                 (leq 4294967260 x))
            (disjoint
             (add 32 4294967244 (neg 32 (difference q1 16))) q1 x k))
   ((enable disjoint-leq disjoint-leq1)))

(prove-lemma qsort-s3-crock2-crock (rewrite)
     (implies (and (leq 68 q1)
                   (lessp q1 4294967244))
              (not (lessp (add 32 4294967244
                                  (neg 32 (difference q1 16)))
                          16)))
     ((enable neg nat-rangep add-nat-la sub-nat-la)
      (disable sub-neg)))

(prove-lemma qsort-s3-crock2 (rewrite)
   (implies (and (leq 68 q1)
                 (lessp (plus 52 q1) 4294967296)
                 (leq (plus x k) 16))
            (disjoint
             (add 32 4294967244 (neg 32 (difference q1 16))) q1 x k))
   ((enable disjoint-leq disjoint-leq1)))

(disable qsort-s3-crock2-crock)

(prove-lemma qsort-s3-s4 (rewrite)
    (let ((lst1 (qsort l (sub1 (qlast l r lst)) (qpart l r lst)))
          (last (qlast l r lst)))
      (implies (and (qsort-s3p s s3 a l r n lst1 last)
                    (qsort-statep s a l r n lst)
                    (qsort-s5p s3 (stepn s3 k) a (add1 last) r n
                               (qsort (add1 last) r lst1))
                    (qsort-sk s3 (stepn s3 k) a (add1 last) r n lst1))
               (qsort-s4p s (stepn s3 k) a l r n (qsort (add1 last) r lst1))))
    ((enable qsort-sk-1 qsort-sk-2)
     (disable add-commutativity1)))

(disable qsort-s3-crock1)
(disable qsort-s3-crock2)

(prove-lemma qsort-s3-s4-mem (rewrite)
     (let ((lst1 (qsort l (sub1 (qlast l r lst)) (qpart l r lst)))
           (last (qlast l r lst)))
       (implies (and (qsort-s3p s s3 a l r n lst1 last)
                     (qsort-statep s a l r n lst)
                     (qsort-sk s s3 a l r n lst)
                     (qsort-sk s3 (stepn s3 k) a (add1 last) r n lst1)
```

```
                    (disjoint a (times 4 n) x k1)
                    (disjoint (sub 32 (difference (qstack l r lst) 16)
                                  (read-sp s))
                              (qstack l r lst) x k1))
               (equal (read-mem x (mc-mem (stepn s3 k)) k1)
                      (read-mem x (mc-mem s) k1))))
     ((enable qsort-sk-1)
      (disable add-commutativity1 qstack-la3)))

(disable qstack-la2~)

(prove-lemma qsort-sk-s-s4 (rewrite)
     (let ((lst1 (qsort l (sub1 (qlast l r lst)) (qpart l r lst)))
           (last (qlast l r lst)))
       (implies (and (qsort-s3p s s3 a l r n lst1 last)
                     (qsort-statep s a l r n lst)
                     (qsort-sk s s3 a l r n lst)
                     (qsort-sk s3 (stepn s3 k) a (add1 last) r n lst1))
                (qsort-sk s (stepn s3 k) a l r n lst)))
     ((use (qsort-sk (s5 (stepn s3 k))))
      (disable qsort-s3p qsort-statep)))

(disable qsort-s3-s4-mem)

; the correctness of the QSORT program.
(prove-lemma qsort-t--1 (rewrite)
     (equal (qsort-t a l (idifference r 1) n lst)
            (qsort-t a l (sub1 r) n lst))
     ((enable qsort-t)))

(prove-lemma qsort--1 (rewrite)
     (equal (qsort l (idifference r 1) lst)
            (qsort l (sub1 r) lst))
     ((enable qsort)))

(disable qsort-10)
(disable qsort-5)
(disable qsort-3)

(prove-lemma qsort-correctness-la (rewrite)
     (implies
      (and (qsort-statep s a l r n lst)
           (leq oplen 32)
           (d2-7a2-5p rn))
      (and (qsort-s5p s (stepn s (qsort-t a l r n lst)) a l r n
                      (qsort l r lst))
           (qsort-sk s (stepn s (qsort-t a l r n lst)) a l r n lst)
           (equal (read-rn oplen rn (mc-rfile (stepn s (qsort-t a l r n lst))))
                  (read-rn oplen rn (mc-rfile s)))))
     ((induct (qsort-induct s a l r n lst))
      (disable qsort-statep qsort-sp qsort-s0p qsort-s1p qsort-s2p
               qsort-s3p qsort-s4p qsort-s5p idifference qlast-0
               qlast-ub qlast-lb)))
```

```
(prove-lemma qsort-correctness (rewrite)
     (let ((sn (stepn s (qsort-t a l r n lst)))
           (sp (sub 32 (difference (qstack l r lst) 16) (read-sp s))))
       (implies (qsort-statep s a l r n lst)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-rn 32 15 (mc-rfile s)) 4))
                     (implies (and (leq oplen 32)
                                   (d2-7a2-5p rn))
                              (equal (read-rn oplen rn (mc-rfile sn))
                                     (read-rn oplen rn (mc-rfile s))))
                     (implies (and (disjoint sp (qstack l r lst) x k)
                                   (disjoint a (times 4 n) x k))
                              (equal (read-mem x (mc-mem sn) k)
                                     (read-mem x (mc-mem s) k)))
                     (mem-ilst 4 a (mc-mem sn) n (qsort l r lst))))))
    ((use (qsort-correctness-la (oplen 32) (rn 2)))
     (enable qsort-sk-1)
     (disable qsort-statep)))

; in the logic, qsort is correct:
;       1. if left <= i <= j <= right,  lst'[i] <= lst'[j].
;       2. for all x,  (count-lst x l r lst) = (count-lst x l r lst').
(prove-lemma put-commute (rewrite)
     (implies (not (equal (fix i) (fix j)))
              (equal (put-nth v1 i (put-nth v2 j lst))
                     (put-nth v2 j (put-nth v1 i lst)))))
     ((enable put-nth)))

(prove-lemma swap-put-commute (rewrite)
     (implies (and (lessp i l)
                   (lessp i r))
              (equal (swap l r (put-nth v i lst))
                     (put-nth v i (swap l r lst)))))

(prove-lemma swap-commute (rewrite)
     (implies (and (lessp i l)
                   (lessp i r)
                   (lessp j l)
                   (lessp j r))
              (equal (swap l r (swap i j lst))
                     (swap i j (swap l r lst)))))

(defn sublst-ileq (x l r lst)
  (if (lessp r l)
      t
    (and (ileq x (get-nth l lst))
         (sublst-ileq x (add1 l) r lst)))
  ((lessp (difference (add1 r) l))))

(defn sublsts-ileq (l r lst l1 r1 lst1)
```

```
  (if (lessp r l)
      t
    (and (sublst-ileq (get-nth l lst) l1 r1 lst1)
         (sublsts-ileq (add1 l) r lst l1 r1 lst1)))
  ((lessp (difference (add1 r) l))))

(prove-lemma sublst-ileq-lemma (rewrite)
     (implies (and (sublst-ileq x l r lst)
                   (leq l j)
                   (leq j r))
              (not (ilessp (get-nth j lst) x)))
     ((enable get-nth-0)))

(prove-lemma sublsts-ileq-la1 (rewrite)
     (implies (and (sublsts-ileq l r lst l1 r1 lst1)
                   (leq l i)
                   (leq i r))
              (sublst-ileq (get-nth i lst) l1 r1 lst1))
     ((enable get-nth-0)))

(prove-lemma sublst-ileq-put (rewrite)
     (implies (and (ileq x y)
                   (ileq x (get-nth j lst)))
              (equal (sublst-ileq x l r (put-nth y j lst))
                     (sublst-ileq x l r lst)))
     ((enable get-nth-0)))

(prove-lemma sublsts-ileq-put (rewrite)
     (implies (and (sublst-ileq y l1 r1 lst1)
                   (sublst-ileq (get-nth j lst) l1 r1 lst1))
              (equal (sublsts-ileq l r (put-nth y j lst) l1 r1 lst1)
                     (sublsts-ileq l r lst l1 r1 lst1)))
     ((enable get-nth-0)))

(prove-lemma sublst-ileq-swap-la ()
     (implies (and (sublst-ileq x l r lst)
                   (leq l i)
                   (leq i r)
                   (leq l j)
                   (leq j r))
              (sublst-ileq x l r (swap i j lst))))

(prove-lemma sublsts-ileq-swap-la ()
     (implies (and (sublsts-ileq l r lst l1 r1 lst1)
                   (leq l i)
                   (leq i r)
                   (leq l j)
                   (leq j r))
              (sublsts-ileq l r (swap i j lst) l1 r1 lst1)))

(disable swap)

(prove-lemma sublst-ileq-swap-swap (rewrite)
     (equal (sublst-ileq x l r (swap i j (swap i j lst)))
```

```
            (sublst-ileq x l r lst))
     ((enable get-nth-0)))

(prove-lemma sublst-ileq-swap (rewrite)
     (implies (and (leq l i)
                   (leq i r)
                   (leq l j)
                   (leq j r))
              (equal (sublst-ileq x l r (swap i j lst))
                     (sublst-ileq x l r lst)))
     ((use (sublst-ileq-swap-la (lst (swap i j lst)))
           (sublst-ileq-swap-la))))

(prove-lemma sublsts-ileq-swap-swap (rewrite)
     (equal (sublsts-ileq l r (swap i j (swap i j lst)) l1 r1 lst1)
            (sublsts-ileq l r lst l1 r1 lst1))
     ((enable get-nth-0)))

(prove-lemma sublsts-ileq-swap (rewrite)
     (implies (and (leq l i)
                   (leq i r)
                   (leq l j)
                   (leq j r))
              (equal (sublsts-ileq l r (swap i j lst) l1 r1 lst1)
                     (sublsts-ileq l r lst l1 r1 lst1)))
     ((use (sublsts-ileq-swap-la (lst (swap i j lst)))
           (sublsts-ileq-swap-la))))

(prove-lemma sublst-ileq-qpart-aux (rewrite)
     (implies (and (leq l last)
                   (lessp last i)
                   (leq last r)
                   (leq l l1)
                   (leq l1 r)
                   (leq r1 r))
              (equal (sublst-ileq x l r (qpart-aux l1 r1 lst last i))
                     (sublst-ileq x l r lst)))
     ((induct (qpart-aux l1 r1 lst last i))))

(prove-lemma sublsts-ileq-qpart-aux (rewrite)
     (implies (and (leq l last)
                   (lessp last i)
                   (leq last r2)
                   (leq l l2)
                   (leq l2 r)
                   (leq r2 r))
              (equal (sublsts-ileq l r (qpart-aux l2 r2 lst last i) l1 r1 lst1)
                     (sublsts-ileq l r lst l1 r1 lst1)))
     ((induct (qpart-aux l2 r2 lst last i))))

(prove-lemma sublst-ileq-qsort (rewrite)
     (implies (and (leq l l1)
                   (leq r1 r))
              (equal (sublst-ileq x l r (qsort l1 r1 lst))
```

```
                    (sublst-ileq x l r lst)))
     ((enable qpart)
      (disable sublst-ileq)))

(prove-lemma sublsts-ileq-qsort1 (rewrite)
     (implies (and (leq l l2)
                   (leq r2 r))
              (equal (sublsts-ileq l r (qsort l2 r2 lst) l1 r1 lst1)
                     (sublsts-ileq l r lst l1 r1 lst1)))
     ((enable qpart)
      (disable sublsts-ileq)))

(prove-lemma sublsts-ileq-qsort2 (rewrite)
     (implies (and (leq l1 l2)
                   (leq r2 r1))
              (equal (sublsts-ileq l r lst l1 r1 (qsort l2 r2 lst1))
                     (sublsts-ileq l r lst l1 r1 lst1)))
     ((disable qsort)))

(defn qpartx (l r lst last i)
  (if (lessp r i)
      lst
    (if (ilessp (get-nth i lst) (get-nth l lst))
        (qpartx l r (swap (add1 last) i lst) (add1 last) (add1 i))
      (qpartx l r lst last (add1 i))))
  ((lessp (difference (add1 r) i))))

(prove-lemma qpart-aux-qpartx (rewrite)
     (equal (qpart-aux l r lst last i)
            (swap l (qlast-aux l r lst last i) (qpartx l r lst last i)))
     ((enable swap get-nth put-nth)))

(prove-lemma qpartx-get-1 (rewrite)
     (implies (and (lessp last i)
                   (lessp right j))
              (equal (get-nth j (qpartx left right lst last i))
                     (get-nth j lst)))
     ((induct (qpartx left right lst last i))))

(prove-lemma qpartx-get-2 (rewrite)
     (implies (and (lessp last i)
                   (leq j last))
              (equal (get-nth j (qpartx left right lst last i))
                     (get-nth j lst)))
     ((induct (qpartx left right lst last i))))

(prove-lemma qpartx-ilessp-1 (rewrite)
     (implies (and (leq l last)
                   (lessp last i)
                   (lessp last j)
                   (leq j (qlast-aux l r lst last i)))
              (ilessp (get-nth j (qpartx l r lst last i))
                      (get-nth l lst)))
     ((induct (qpartx l r lst last i))))
```

```
(defn open-sublst-ileq (x l r lst)
  (if (lessp (add1 l) r)
      (and (ileq x (get-nth (add1 l) lst))
           (open-sublst-ileq x (add1 l) r lst))
    t)
  ((lessp (difference r l))))

(prove-lemma open-sublst-ileq-la0 (rewrite)
     (implies (not (lessp (add1 l) r))
              (open-sublst-ileq x l r lst)))

(prove-lemma open-sublst-ileq-la1 (rewrite)
     (implies (and (open-sublst-ileq x last i lst)
                   (lessp last j)
                   (lessp j i))
              (not (ilessp (get-nth j lst) x))))

(prove-lemma open-sublst-ileq-la2 (rewrite)
     (equal (open-sublst-ileq x last (add1 i) lst)
            (if (ileq x (get-nth i lst))
                (open-sublst-ileq x last i lst)
              (not (lessp (add1 last) (add1 i))))))

(prove-lemma open-sublst-ileq-la3 (rewrite)
     (implies (leq j l)
              (equal (open-sublst-ileq x l r (put-nth v j lst))
                     (open-sublst-ileq x l r lst))))

(prove-lemma open-sublst-ileq-la4 (rewrite)
     (implies (leq r j)
              (equal (open-sublst-ileq x l r (put-nth v j lst))
                     (open-sublst-ileq x l r lst))))

(prove-lemma open-sublst-ileq-la5 (rewrite)
     (equal (open-sublst-ileq x (add1 last) (add1 i) (swap (add1 last) i lst))
            (open-sublst-ileq x last i lst))
     ((enable swap)))

(prove-lemma qpartx-ilessp-2 (rewrite)
     (implies (and (open-sublst-ileq (get-nth l lst) last i lst)
                   (leq l last)
                   (lessp last i)
                   (lessp (qlast-aux l r lst last i) j)
                   (leq j r))
              (not (ilessp (get-nth j (qpartx l r lst last i))
                           (get-nth l lst))))
     ((induct (qpartx l r lst last i))))

(prove-lemma open-sublst-ileq-la6 (rewrite)
     (open-sublst-ileq (get-nth l lst) l (add1 l) lst))

(prove-lemma qpart-ilessp-la1-la ()
     (implies (and (leq l j)
```

```
                (lessp j (qlast-aux l r lst l (add1 l))))
           (not (ilessp (get-nth l lst)
                        (get-nth j (qpart-aux l r lst l (add1 l)))))))

(prove-lemma qpart-ilessp-la1 (rewrite)
     (implies (and (leq l j)
                   (leq j (sub1 (qlast-aux l r lst l (add1 l)))))
              (not (ilessp (get-nth l lst)
                           (get-nth j (qpart-aux l r lst l (add1 l))))))
     ((use (qpart-ilessp-la1-la))
      (enable get-nth-0)))

(prove-lemma qpart-ilessp-la2 (rewrite)
     (implies (and (leq (qlast-aux l r lst l (add1 l)) j)
                   (leq j r))
              (not (ilessp (get-nth j (qpart-aux l r lst l (add1 l)))
                           (get-nth l lst))))
     ((use (qpartx-ilessp-2 (last l) (i (add1 l))))
      (enable get-nth-0)))

(prove-lemma qpart-equal (rewrite)
     (implies (and (leq l last)
                   (lessp last i))
              (equal (get-nth (qlast-aux l r lst last i)
                              (qpart-aux l r lst last i))
                     (get-nth l lst)))
     ((enable get-nth-0)))

(prove-lemma qsort-get-1 (rewrite)
     (implies (lessp j left)
              (equal (get-nth j (qsort left right lst))
                     (get-nth j lst)))
     ((enable qpart qlast)))

(prove-lemma get-swap-1 (rewrite)
     (implies (and (lessp j i)
                   (lessp k i))
              (equal (get-nth i (swap j k lst))
                     (get-nth i lst))))

(prove-lemma qlast-aux-swap (rewrite)
     (implies (and (leq l last)
                   (lessp last i)
                   (lessp a l)
                   (lessp b l))
              (equal (qlast-aux l r (swap a b lst) last i)
                     (qlast-aux l r lst last i)))
     ((induct (qlast-aux l r lst last i))))

(prove-lemma qlast-swap (rewrite)
     (implies (and (lessp l r)
                   (lessp a l)
                   (lessp b l))
              (equal (qlast l r (swap a b lst))
```

```
                  (qlast l r lst)))
     ((enable qlast) (disable qlast-aux)))

(prove-lemma qpart-aux-swap (rewrite)
     (implies (and (leq l last)
                   (lessp last i)
                   (lessp a l)
                   (lessp b l))
              (equal (qpart-aux l r (swap a b lst) last i)
                     (swap a b (qpart-aux l r lst last i))))
     ((induct (qpart-aux l r lst last i))))

(prove-lemma qpart-swap (rewrite)
     (implies (and (lessp l r)
                   (lessp a l)
                   (lessp b l))
              (equal (qpart l r (swap a b lst))
                     (swap a b (qpart l r lst))))
     ((enable qpart) (disable qpart-aux)))

(prove-lemma qsort-swap (rewrite)
     (implies (and (lessp i l)
                   (lessp j l))
              (equal (qsort l r (swap i j lst))
                     (swap i j (qsort l r lst))))
     ((induct (qsort l r lst))))

(prove-lemma qsort-qpartx (rewrite)
     (implies (and (leq l r)
                   (leq l last)
                   (lessp last i)
                   (leq last r)
                   (lessp r l1))
              (equal (qsort l1 r1 (qpartx l r lst last i))
                     (qpartx l r (qsort l1 r1 lst) last i)))
     ((induct (qpartx l r lst last i))))

(prove-lemma qsort-get-2 (rewrite)
     (implies (lessp r l1)
              (equal (get-nth j (qsort l1 r1 (qsort l r lst)))
                     (if (lessp r j)
                         (get-nth j (qsort l1 r1 lst))
                       (get-nth j (qsort l r lst))))))
     ((enable qpart)
      (disable quotient plus qlast-aux qpart-aux qpartx)))

(disable qpart-aux-qpartx)

(prove-lemma ilessp-trans (rewrite)
     (implies (and (ilessp a b) (ileq b c))
              (ilessp a c))
     ((enable ilessp)))

(prove-lemma qpart-ilessp (rewrite)
```

```
     (implies (and (leq l i)
                   (leq i (qlast l r lst))
                   (leq (qlast l r lst) j)
                   (leq j r))
              (not (ilessp (get-nth j (qpart l r lst))
                           (get-nth i (qpart l r lst)))))
     ((use (qpart-ilessp-la1 (j i) (lst (swap l (quotient (plus l r) 2) lst)))
           (qpart-ilessp-la2 (lst (swap l (quotient (plus l r) 2) lst))))
      (enable qpart qlast) (disable quotient plus qlast-aux qpart-aux)))

(prove-lemma qsort-get3 (rewrite)
     (equal (get-nth j (qsort (add1 last) r (qsort l (sub1 last) lst)))
            (if (leq j (sub1 last))
                (get-nth j (qsort l (sub1 last) lst))
              (get-nth j (qsort (add1 last) r lst)))))

(prove-lemma sublsts-ileq-la2 (rewrite)
     (implies (and (sublsts-ileq l r lst l1 r1 lst1)
                   (leq l i)
                   (leq i r)
                   (leq l1 j)
                   (leq j r1))
              (not (ilessp (get-nth j lst1) (get-nth i lst))))
     ((expand (sublsts-ileq i r lst l1 r1 lst1)
              (sublsts-ileq 0 r lst l1 r1 lst1))
      (enable get-nth-0)))

(prove-lemma qpart-ilessp-closed-1 (rewrite)
     (implies (leq (qlast l r lst) last)
              (sublst-ileq (get-nth (qlast l r lst) (qpart l r lst))
                           last r (qpart l r lst)))
     ((induct (sublst-ileq x last r lst1))))

(prove-lemma qsort-ilessp-1 (rewrite)
     (let ((lst1 (qpart l r lst))
           (last (qlast l r lst)))
       (implies (and (leq last j)
                     (leq j r))
                (not (ilessp (get-nth j (qsort (add1 last) r lst1))
                             (get-nth last lst1))))))
     ((use (sublst-ileq-lemma (x (get-nth (qlast l r lst) (qpart l r lst)))
                              (l (qlast l r lst))
                              (lst (qsort (add1 (qlast l r lst))
                                          r
                                          (qpart l r lst)))))))

(prove-lemma qpart-ilessp-closed-2 (rewrite)
     (let ((lst1 (qpart l r lst)))
       (implies (and (leq (qlast l r lst) last)
                     (leq l j)
                     (leq j (sub1 (qlast l r lst))))
                (sublst-ileq (get-nth j lst1) last r lst1)))
     ((induct (sublst-ileq x last r lst1))))
```

```
(prove-lemma qpart-ilessp-closed-3 (rewrite)
     (let ((last (qlast l r lst))
           (lst1 (qpart l r lst)))
       (implies (leq l l1)
                (sublsts-ileq l1 (sub1 last) lst1 last r lst1))))

(prove-lemma qsort-ilessp-2 (rewrite)
     (let ((lst1 (qpart l r lst))
           (last (qlast l r lst)))
       (implies (and (leq l i)
                     (leq i (sub1 last))
                     (leq last j)
                     (leq j r))
                (not (ilessp (get-nth j (qsort (add1 last) r lst1))
                             (get-nth i (qsort l (sub1 last) lst1)))))))
     ((use (sublsts-ileq-la2 (r (sub1 (qlast l r lst)))
                             (lst (qsort l
                                         (sub1 (qlast l r lst))
                                         (qpart l r lst)))
                             (l1 (qlast l r lst))
                             (r1 r)
                             (lst1 (qsort (add1 (qlast l r lst))
                                          r
                                          (qpart l r lst))))
           (qpart-ilessp-closed-3 (l1 l)))))

(prove-lemma qsort-ordered (rewrite)
     (implies (and (leq left i)
                   (leq i j)
                   (leq j right))
              (not (ilessp (get-nth j (qsort left right lst))
                           (get-nth i (qsort left right lst)))))
     ((expand (qsort 0 right lst)
              (qsort left right lst))
      (enable get-nth-0)))

(defn orderedp1 (l r lst)
  (if (leq r l)
      t
    (and (ileq (get-nth l lst) (get-nth (add1 l) lst))
         (orderedp1 (add1 l) r lst)))
  ((lessp (difference r l))))

(prove-lemma qsort-orderedp1-la (rewrite)
     (implies (leq left left1)
              (orderedp1 left1 right (qsort left right lst))))

(prove-lemma qsort-orderedp1 (rewrite)
     (orderedp1 left right (qsort left right lst)))

(defn transwap (i lst)
  (swap i (add1 i) lst))

(defn lst-eq (l r lst lst1)
```

```
  (if (lessp r l)
      t
    (and (equal (get-nth l lst) (get-nth l lst1))
         (lst-eq (add1 l) r lst lst1)))
  ((lessp (difference (add1 r) l))))

(defn count-lst (x l r lst)
  (if (lessp r l)
      0
    (if (equal x (get-nth l lst))
        (add1 (count-lst x (add1 l) r lst))
      (count-lst x (add1 l) r lst)))
  ((lessp (difference (add1 r) l))))

(prove-lemma count-lst-0 (rewrite)
     (implies (not (numberp l))
              (equal (count-lst x l r lst)
                     (count-lst x 0 r lst)))
     ((expand (count-lst x l r lst))
      (enable get-nth-0)))

(prove-lemma count-swapii (rewrite)
     (equal (count-lst x l r (swap i i lst))
            (count-lst x l r lst))
     ((enable get-nth-0)))

(prove-lemma count-lst-put-1 (rewrite)
     (implies (lessp i l)
              (equal (count-lst x l r (put-nth v i lst))
                     (count-lst x l r lst))))

(prove-lemma count-lst-swap-1 (rewrite)
     (implies (and (lessp i l)
                   (lessp j l))
              (equal (count-lst x l r (swap i j lst))
                     (count-lst x l r lst)))
     ((enable swap)))

(prove-lemma count-transwap-0 (rewrite)
    (implies (lessp l r)
             (equal (count-lst x l r (swap l (add1 l) lst))
                    (count-lst x l r lst)))
     ((expand (count-lst x l r (swap l (add1 l) lst))
              (count-lst x (add1 l) r (swap l (add1 l) lst))
              (count-lst x l r lst)
              (count-lst x (add1 l) r lst))))

(prove-lemma swap-0 (rewrite)
     (implies (not (numberp i))
              (and (equal (swap i j lst) (swap 0 j lst))
                   (equal (swap j i lst) (swap j 0 lst))))
     ((enable swap put-nth get-nth)))

(prove-lemma count-transwap (rewrite)
```

```
     (implies (and (leq l i)
                   (lessp i r))
              (equal (count-lst x l r (transwap i lst))
                     (count-lst x l r lst)))
     ((enable get-nth-0)))

(prove-lemma swap-rec-la (rewrite)
     (implies (leq i j)
              (lst-eq l
                      r
                      (swap i (add1 j) lst)
                      (swap i j (swap j (add1 j) (swap i j lst))))))
     ((enable get-nth-0)))

(prove-lemma count-lst-eq ()
     (implies (lst-eq l r lst lst1)
              (equal (count-lst x l r lst)
                     (count-lst x l r lst1))))

(prove-lemma count-lst-swap-rec (rewrite)
     (implies (leq i j)
              (equal (count-lst x l r (swap i (add1 j) lst))
                     (count-lst x l r (swap i j (transwap j (swap i j lst))))))
     ((use (count-lst-eq (lst (swap i (add1 j) lst))
                         (lst1 (swap i j (transwap j (swap i j lst))))))))

(defn swap-induct (i j lst)
  (if (leq j i)
      t
    (and (swap-induct i (sub1 j) (transwap (sub1 j) (swap i (sub1 j) lst)))
         (swap-induct i (sub1 j) lst)))
  ((lessp (difference j i))))

(prove-lemma count-lst-swap (rewrite)
     (implies (and (leq l i)
                   (leq i j)
                   (leq j r))
              (equal (count-lst x l r (swap i j lst))
                     (count-lst x l r lst)))
     ((induct (swap-induct i j lst))
      (disable transwap)))

(prove-lemma count-lst-qpart-aux (rewrite)
     (implies (and (leq l1 last)
                   (lessp last i)
                   (leq last r1)
                   (leq l l1)
                   (leq l1 r)
                   (leq r1 r))
              (equal (count-lst x l r (qpart-aux l1 r1 lst last i))
                     (count-lst x l r lst)))
     ((induct (qpart-aux l1 r1 lst last i))))

(prove-lemma count-lst-qsort-la (rewrite)
```

```
     (implies (and (leq l l1)
                   (leq r1 r))
              (equal (count-lst x l r (qsort l1 r1 lst))
                     (count-lst x l r lst)))
     ((enable qpart)
      (disable count-lst quotient plus)))

(prove-lemma count-lst-qsort (rewrite)
     (equal (count-lst x l r (qsort l r lst))
            (count-lst x l r lst)))
```

C.5 Boyer-Moore Majority Voting

```
;         Proof of the Corectness of a Majority Voting Program
;
#|
The following C function MJRTY determines if there is a candidate who
has received a majority of votes cast in an election.

/* a majority voting algorithm invented by Boyer and Moore */
#define YES 1
#define NO  0

struct winner {
  int x;
  int y;
};

struct winner mjrty (int a[], int n)
{
  int cand, i, k;
  struct winner temp;

  k = 0;
  for (i = 0; i < n; i++)
    if (k == 0) {
      cand = a[i];
      k = 1;
    }
  else {
    if (cand == a[i])
      k++;
    else
      k--;
  };
  temp.x = cand;
  if (k == 0) {
    temp.y = NO;
    return temp;
  };
  if (k > n/2) {
    temp.y = YES;
```

```
    return temp;
  };
  k = 0;
  for (i = 0; i < n; i++)
    if (a[i] == cand)
      k++;
  if (k > n/2)
    temp.y = YES;
  else temp.y = NO;
  return temp;
}
```

Here is the MC68020 assembly code of the above C program. The code is generated by "gcc -O".

```
0x2310 <mjrty>:         linkw a6,#0
0x2314 <mjrty+4>:       moveml d2-d5,sp@-
0x2318 <mjrty+8>:       moveal a6@(8),a0
0x231c <mjrty+12>:      movel a6@(12),d2
0x2320 <mjrty+16>:      clrl d1
0x2322 <mjrty+18>:      clrl d0
0x2324 <mjrty+20>:      cmpl d0,d2
0x2326 <mjrty+22>:      ble 0x2346 <mjrty+54>
0x2328 <mjrty+24>:      tstl d1
0x232a <mjrty+26>:      bne 0x2334 <mjrty+36>
0x232c <mjrty+28>:      movel 0(a0)[d0.l*4],d3
0x2330 <mjrty+32>:      movel #1,d1
0x2332 <mjrty+34>:      bra 0x2340 <mjrty+48>
0x2334 <mjrty+36>:      cmpl 0(a0)[d0.l*4],d3
0x2338 <mjrty+40>:      bne 0x233e <mjrty+46>
0x233a <mjrty+42>:      addql #1,d1
0x233c <mjrty+44>:      bra 0x2340 <mjrty+48>
0x233e <mjrty+46>:      subl #1,d1
0x2340 <mjrty+48>:      addql #1,d0
0x2342 <mjrty+50>:      cmpl d0,d2
0x2344 <mjrty+52>:      bgt 0x2328 <mjrty+24>
0x2346 <mjrty+54>:      movel d3,d4
0x2348 <mjrty+56>:      tstl d1
0x234a <mjrty+58>:      beq 0x2382 <mjrty+114>
0x234c <mjrty+60>:      movel d2,d0
0x234e <mjrty+62>:      bge 0x2352 <mjrty+66>
0x2350 <mjrty+64>:      addql #1,d0
0x2352 <mjrty+66>:      asrl #1,d0
0x2354 <mjrty+68>:      cmpl d1,d0
0x2356 <mjrty+70>:      bge 0x235c <mjrty+76>
0x2358 <mjrty+72>:      movel #1,d5
0x235a <mjrty+74>:      bra 0x2384 <mjrty+116>
0x235c <mjrty+76>:      clrl d1
0x235e <mjrty+78>:      clrl d0
0x2360 <mjrty+80>:      cmpl d0,d2
0x2362 <mjrty+82>:      ble 0x2372 <mjrty+98>
0x2364 <mjrty+84>:      cmpl 0(a0)[d0.l*4],d3
0x2368 <mjrty+88>:      bne 0x236c <mjrty+92>
0x236a <mjrty+90>:      addql #1,d1
```

```
0x236c <mjrty+92>:      addql #1,d0
0x236e <mjrty+94>:      cmpl d0,d2
0x2370 <mjrty+96>:      bgt 0x2364 <mjrty+84>
0x2372 <mjrty+98>:      movel d2,d0
0x2374 <mjrty+100>:     bge 0x2378 <mjrty+104>
0x2376 <mjrty+102>:     addql #1,d0
0x2378 <mjrty+104>:     asrl #1,d0
0x237a <mjrty+106>:     cmpl d1,d0
0x237c <mjrty+108>:     bge 0x2382 <mjrty+114>
0x237e <mjrty+110>:     movel #1,d5
0x2380 <mjrty+112>:     bra 0x2384 <mjrty+116>
0x2382 <mjrty+114>:     clrl d5
0x2384 <mjrty+116>:     movel d4,d0
0x2386 <mjrty+118>:     movel d5,d1
0x2388 <mjrty+120>:     moveml a6@(-16),d2-d5
0x238e <mjrty+126>:     unlk a6
0x2390 <mjrty+128>:     rts

The machine code of the above program is:


<mjrty>:      0x4e56  0x0000  0x48e7  0x3c00  0x206e  0x0008  0x242e  0x000c
<mjrty+16>:   0x4281  0x4280  0xb480  0x6f1e  0x4a81  0x6608  0x2630  0x0c00
<mjrty+32>:   0x7201  0x600c  0xb6b0  0x0c00  0x6604  0x5281  0x6002  0x5381
<mjrty+48>:   0x5280  0xb480  0x6ee2  0x2803  0x4a81  0x6736  0x2002  0x6c02
<mjrty+64>:   0x5280  0xe280  0xb081  0x6c04  0x7a01  0x6028  0x4281  0x4280
<mjrty+80>:   0xb480  0x6f0e  0xb6b0  0x0c00  0x6602  0x5281  0x5280  0xb480
<mjrty+96>:   0x6ef2  0x2002  0x6c02  0x5280  0xe280  0xb081  0x6c04  0x7a01
<mjrty+112>:  0x6002  0x4285  0x2004  0x2205  0x4cee  0x003c  0xfff0  0x4e5e
<mjrty+128>:  0x4e75

In the Nqthm logic, this is:

'(78      86      0       0       72      231     60      0
  32      110     0       8       36      46      0       12
  66      129     66      128     180     128     111     30
  74      129     102     8       38      48      12      0
  114     1       96      12      182     176     12      0
  102     4       82      129     96      2       83      129
  82      128     180     128     110     226     40      3
  74      129     103     54      32      2       108     2
  82      128     226     128     176     129     108     4
  122     1       96      40      66      129     66      128
  180     128     111     14      182     176     12      0
  102     2       82      129     82      128     180     128
  110     242     32      2       108     2       82      128
  226     128     176     129     108     4       122     1
  96      2       66      133     32      4       34      5
  76      238     0       60      255     240     78      94
  78      117)
|#

; in the logic, the above program is defined by (mjrty-code).
(defn mjrty-code ()
```

```
  '(78      86      0       0       72      231     60      0
    32      110     0       8       36      46      0       12
    66      129     66      128     180     128     111     30
    74      129     102     8       38      48      12      0
    114     1       96      12      182     176     12      0
    102     4       82      129     96      2       83      129
    82      128     180     128     110     226     40      3
    74      129     103     54      32      2       108     2
    82      128     226     128     176     129     108     4
    122     1       96      40      66      129     66      128
    180     128     111     14      182     176     12      0
    102     2       82      129     82      128     180     128
    110     242     32      2       108     2       82      128
    226     128     176     129     108     4       122     1
    96      2       66      133     32      4       34      5
    76      238     0       60      255     240     78      94
    78      117))

; mjrty-cand is a function in the logic to simulate the candidate
; findhe above code.
(defn mjrty-cand (n lst cand i k)
  (if (lessp i n)
      (if (zerop k)
          (mjrty-cand n lst (get-nth i lst) (add1 i) 1)
        (if (equal cand (get-nth i lst))
            (mjrty-cand n lst cand (add1 i) (add1 k))
          (mjrty-cand n lst cand (add1 i) (sub1 k))))
    cand)
  ((lessp (difference n i))))

(defn mjrty-k (n lst cand i k)
  (if (lessp i n)
      (if (zerop k)
          (mjrty-k n lst (get-nth i lst) (add1 i) 1)
        (if (equal cand (get-nth i lst))
            (mjrty-k n lst cand (add1 i) (add1 k))
          (mjrty-k n lst cand (add1 i) (sub1 k))))
    k)
  ((lessp (difference n i))))

; cand-cnt is a function in the logic to simulate the process of
; counting the number of votes for the given candidate.
(defn cand-cnt (n lst cand i k)
  (if (lessp i n)
      (if (equal cand (get-nth i lst))
          (cand-cnt n lst cand (add1 i) (add1 k))
        (cand-cnt n lst cand (add1 i) k))
    k)
  ((lessp (difference n i))))

; mjrty-p determines if the given candidate cand has received a majority
; voting.
(defn mjrty-p (n lst cand i k)
  (if (zerop (mjrty-k n lst cand i k))
```

```
      f
    (if (lessp (quotient n 2) (mjrty-k n lst cand i k))
        t
      (lessp (quotient n 2)
             (cand-cnt n lst (mjrty-cand n lst cand i k) i k)))))

; the computation time.
(defn mjrty-cand-t (a n lst cand i k)
  (if (lessp i n)
      (if (zerop k)
          (let ((cand1 (get-nth i lst)))
            (splus 8 (mjrty-cand-t a n lst cand1 (add1 i) 1)))
        (if (equal cand (get-nth i lst))
            (splus 9 (mjrty-cand-t a n lst cand (add1 i) (add1 k)))
          (splus 8 (mjrty-cand-t a n lst cand (add1 i) (sub1 k)))))
    (if (equal cand (get-nth 0 lst))
        18 17))
  ((lessp (difference n i))))

(defn mjrty-sn-t (a n lst cand i k)
  (if (lessp i n)
      (if (zerop k)
          (let ((cand1 (get-nth i lst)))
            (splus 8 (mjrty-sn-t a n lst cand1 (add1 i) 1)))
        (if (equal cand (get-nth i lst))
            (splus 9 (mjrty-sn-t a n lst cand (add1 i) (add1 k)))
          (splus 8 (mjrty-sn-t a n lst cand (add1 i) (sub1 k)))))
    (if (zerop k) 11 17))
  ((lessp (difference n i))))

(defn cand-cnt-t (a n lst cand i k)
  (if (lessp i n)
      (if (equal cand (get-nth i lst))
          (splus 6 (cand-cnt-t a n lst cand (add1 i) (add1 k)))
        (splus 5 (cand-cnt-t a n lst cand (add1 i) k)))
    (if (lessp (quotient n 2) k) 14 13))
  ((lessp (difference n i))))

(defn mjrty-t (a n lst)
  (let ((cand (get-nth 0 lst)))
    (splus 14
           (if (or (zerop (mjrty-k n lst cand 1 1))
                   (lessp (quotient n 2) (mjrty-k n lst cand 1 1)))
               (mjrty-sn-t a n lst cand 1 1)
             (splus (mjrty-cand-t a n lst cand 1 1)
                    (if (equal cand (mjrty-cand n lst cand 1 1))
                        (cand-cnt-t a n lst (mjrty-cand n lst cand 1 1) 1 1)
                      (cand-cnt-t a n lst (mjrty-cand n lst cand 1 1) 1 0)))))))

; induction hints.
(defn mjrty-cand-induct (s n lst cand i k)
  (if (lessp i n)
      (if (zerop k)
          (let ((cand1 (get-nth i lst)))
```

```
            (mjrty-cand-induct (stepn s 8) n lst cand1 (add1 i) 1))
        (if (equal cand (get-nth i lst))
            (mjrty-cand-induct (stepn s 9) n lst cand (add1 i) (add1 k))
          (mjrty-cand-induct (stepn s 8) n lst cand (add1 i) (sub1 k))))
    t)
  ((lessp (difference n i))))

(defn cand-cnt-induct (s n lst cand i k)
  (if (lessp i n)
      (if (equal cand (get-nth i lst))
          (cand-cnt-induct (stepn s 6) n lst cand (add1 i) (add1 k))
        (cand-cnt-induct (stepn s 5) n lst cand (add1 i) k))
    t)
  ((lessp (difference n i))))

; the preconditions of the initial state.
(defn mjrty-statep (s a n lst)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (mc-pc s) (mc-mem s) 130)
       (mcode-addrp (mc-pc s) (mc-mem s) (mjrty-code))
       (ram-addrp (sub 32 20 (read-sp s)) (mc-mem s) 32)
       (ram-addrp a (mc-mem s) (times 4 n))
       (mem-ilst 4 a (mc-mem s) n lst)
       (disjoint a (times 4 n) (sub 32 20 (read-sp s)) 32)
       (equal a (read-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal n (iread-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
       (not (zerop n))))

; the conditions of the intermediate state s0.
(defn mjrty-s0p (s a n lst cand i k)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 50 (mc-pc s)) (mc-mem s) 130)
       (mcode-addrp (sub 32 50 (mc-pc s)) (mc-mem s) (mjrty-code))
       (ram-addrp (sub 32 16 (read-an 32 6 s)) (mc-mem s) 32)
       (ram-addrp a (mc-mem s) (times 4 n))
       (mem-ilst 4 a (mc-mem s) n lst)
       (disjoint a (times 4 n) (sub 32 16 (read-an 32 6 s)) 32)
       (equal a (read-rn 32 8 (mc-rfile s)))
       (equal n (nat-to-int (read-rn 32 2 (mc-rfile s)) 32))
       (equal cand (nat-to-int (read-rn 32 3 (mc-rfile s)) 32))
       (equal i (nat-to-int (read-rn 32 0 (mc-rfile s)) 32))
       (equal k (nat-to-int (read-rn 32 1 (mc-rfile s)) 32))
       (not (zerop n))
       (numberp i)
       (numberp k)
       (leq k i)))

; the conditions of the intermediate state s1.
(defn mjrty-s1p (s a n lst cand i k)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 94 (mc-pc s)) (mc-mem s) 130)
```

```
       (mcode-addrp (sub 32 94 (mc-pc s)) (mc-mem s) (mjrty-code))
       (ram-addrp (sub 32 16 (read-an 32 6 s)) (mc-mem s) 32)
       (ram-addrp a (mc-mem s) (times 4 n))
       (mem-ilst 4 a (mc-mem s) n lst)
       (disjoint a (times 4 n) (sub 32 16 (read-an 32 6 s)) 32)
       (equal a (read-rn 32 8 (mc-rfile s)))
       (equal n (nat-to-int (read-rn 32 2 (mc-rfile s)) 32))
       (equal cand (nat-to-int (read-rn 32 4 (mc-rfile s)) 32))
       (equal cand (nat-to-int (read-rn 32 3 (mc-rfile s)) 32))
       (equal i (nat-to-int (read-rn 32 0 (mc-rfile s)) 32))
       (equal k (nat-to-int (read-rn 32 1 (mc-rfile s)) 32))
       (not (zerop n))
       (numberp i)
       (numberp k)
       (leq k i)))

; the initial segment.  From the initial state to s0.
(prove-lemma mjrty-s-s0 (rewrite)
     (let ((cand (get-nth 0 lst)))
       (implies (mjrty-statep s a n lst)
                (and (mjrty-s0p (stepn s 14) a n lst cand 1 1)
                     (equal (linked-rts-addr (stepn s 14))
                            (rts-addr s))
                     (equal (linked-a6 (stepn s 14))
                            (read-an 32 6 s))
                     (equal (read-rn 32 14 (mc-rfile (stepn s 14)))
                            (sub 32 4 (read-sp s)))
                     (equal (movem-saved (stepn s 14) 4 16 4)
                            (readm-rn 32 '(2 3 4 5) (mc-rfile s)))))))
     ((disable times)))

(prove-lemma mjrty-s-s0-rfile (rewrite)
     (implies (and (mjrty-statep s a n lst)
                   (d6-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 14)))
                     (read-rn oplen rn (mc-rfile s))))
     ((disable times)))

(prove-lemma mjrty-s-s0-mem (rewrite)
     (implies (and (mjrty-statep s a n lst)
                   (disjoint (sub 32 20 (read-sp s)) 32 x k))
              (equal (read-mem x (mc-mem (stepn s 14)) k)
                     (read-mem x (mc-mem s) k)))
     ((disable times)))

; s0 --> exit.
; base case.
(prove-lemma mjrty-s0-sn-base-1 (rewrite)
     (implies (and (mjrty-s0p s a n lst cand i k)
                   (not (lessp i n))
                   (zerop k))
              (and (equal (mc-status (stepn s 11)) 'running)
                   (equal (mc-pc (stepn s 11)) (linked-rts-addr s))
                   (equal (iread-dn 32 0 (stepn s 11)) cand)
```

```
                  (equal (iread-dn 32 1 (stepn s 11)) 0)
                  (equal (read-rn 32 14 (mc-rfile (stepn s 11)))
                         (linked-a6 s))
                  (equal (read-rn 32 15 (mc-rfile (stepn s 11)))
                         (add 32 (read-an 32 6 s) 8))
                  (equal (read-mem x (mc-mem (stepn s 11)) l)
                         (read-mem x (mc-mem s) l)))))

(prove-lemma mjrty-s0-sn-base-2 (rewrite)
     (implies (and (mjrty-s0p s a n lst cand i k)
                   (not (lessp i n))
                   (not (zerop k))
                   (lessp (quotient n 2) k))
              (and (equal (mc-status (stepn s 17)) 'running)
                   (equal (mc-pc (stepn s 17)) (linked-rts-addr s))
                   (equal (iread-dn 32 0 (stepn s 17)) cand)
                   (equal (iread-dn 32 1 (stepn s 17)) 1)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 17)))
                          (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 17)))
                          (add 32 (read-an 32 6 s) 8))
                   (equal (read-mem x (mc-mem (stepn s 17)) l)
                          (read-mem x (mc-mem s) l))))
     ((enable iquotient)))

(prove-lemma mjrty-s0-sn-rfile-base-1 (rewrite)
     (implies
      (and (mjrty-s0p s a n lst cand i k)
           (not (lessp i n))
           (zerop k)
           (d2-7a2-5p rn)
           (leq oplen 32))
      (equal (read-rn oplen rn (mc-rfile (stepn s 11)))
             (if (d6-7a2-5p rn)
                 (read-rn oplen rn (mc-rfile s))
               (get-vlst oplen 0 rn '(2 3 4 5) (movem-saved s 4 16 4))))))

(prove-lemma mjrty-s0-sn-rfile-base-2 (rewrite)
     (implies
      (and (mjrty-s0p s a n lst cand i k)
           (not (lessp i n))
           (not (zerop k))
           (lessp (quotient n 2) k)
           (d2-7a2-5p rn)
           (leq oplen 32))
      (equal (read-rn oplen rn (mc-rfile (stepn s 17)))
             (if (d6-7a2-5p rn)
                 (read-rn oplen rn (mc-rfile s))
               (get-vlst oplen 0 rn '(2 3 4 5) (movem-saved s 4 16 4)))))
     ((enable iquotient)))

; induction case.
(prove-lemma add1-int-rangep (rewrite)
             (implies (lessp x (nat-to-int y n))
```

```
                    (int-rangep (add1 x) n))
             ((enable int-rangep nat-to-int)))

(enable iplus)

(prove-lemma mjrty-s0-s0-1 (rewrite)
     (let ((cand1 (get-nth i lst)))
       (implies (and (mjrty-s0p s a n lst cand i k)
                     (lessp i n)
                     (zerop k))
              (and (mjrty-s0p (stepn s 8) a n lst cand1 (add1 i) 1)
                   (equal (linked-rts-addr (stepn s 8))
                          (linked-rts-addr s))
                   (equal (linked-a6 (stepn s 8)) (linked-a6 s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 8)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (movem-saved (stepn s 8) 4 16 4)
                          (movem-saved s 4 16 4))
                   (equal (read-mem x (mc-mem (stepn s 8)) l)
                          (read-mem x (mc-mem s) l))))))
     ((disable times lessp)))

(prove-lemma add1-int-rangepxx (rewrite)
             (implies (and (leq i r)
                           (lessp r n)
                           (int-rangep n 32))
                      (int-rangep (add1 i) 32))
             ((enable int-rangep nat-to-int)))

(prove-lemma mjrty-s0-s0-2 (rewrite)
     (implies (and (mjrty-s0p s a n lst cand i k)
                   (lessp i n)
                   (not (zerop k))
                   (equal cand (get-nth i lst)))
              (and (mjrty-s0p (stepn s 9) a n lst cand (add1 i) (add1 k))
                   (equal (linked-rts-addr (stepn s 9))
                          (linked-rts-addr s))
                   (equal (linked-a6 (stepn s 9)) (linked-a6 s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 9)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (movem-saved (stepn s 9) 4 16 4)
                          (movem-saved s 4 16 4))
                   (equal (read-mem x (mc-mem (stepn s 9)) l)
                          (read-mem x (mc-mem s) l))))
     ((disable times lessp)))

(prove-lemma mjrty-s0-s0-3 (rewrite)
     (implies (and (mjrty-s0p s a n lst cand i k)
                   (lessp i n)
                   (not (zerop k))
                   (not (equal cand (get-nth i lst))))
              (and (mjrty-s0p (stepn s 8) a n lst cand (add1 i) (sub1 k))
                   (equal (linked-rts-addr (stepn s 8))
                          (linked-rts-addr s))
```

```
                    (equal (linked-a6 (stepn s 8)) (linked-a6 s))
                    (equal (read-rn 32 14 (mc-rfile (stepn s 8)))
                           (read-rn 32 14 (mc-rfile s)))
                    (equal (movem-saved (stepn s 8) 4 16 4)
                           (movem-saved s 4 16 4))
                    (equal (read-mem x (mc-mem (stepn s 8)) l)
                           (read-mem x (mc-mem s) l))))
     ((disable times lessp)))

(prove-lemma mjrty-s0-s0-rfile-1 (rewrite)
     (implies (and (mjrty-s0p s a n lst cand i k)
                    (lessp i n)
                    (zerop k)
                    (d6-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 8)))
                     (read-rn oplen rn (mc-rfile s))))
     ((disable times lessp)))

(prove-lemma mjrty-s0-s0-rfile-2 (rewrite)
     (implies (and (mjrty-s0p s a n lst cand i k)
                    (lessp i n)
                    (not (zerop k))
                    (equal cand (get-nth i lst))
                    (d6-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 9)))
                     (read-rn oplen rn (mc-rfile s))))
     ((disable times lessp)))

(prove-lemma mjrty-s0-s0-rfile-3 (rewrite)
     (implies (and (mjrty-s0p s a n lst cand i k)
                    (lessp i n)
                    (not (zerop k))
                    (not (equal cand (get-nth i lst)))
                    (d6-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 8)))
                     (read-rn oplen rn (mc-rfile s))))
     ((disable times lessp)))

; the proof of s0 --> exit.
(prove-lemma mjrty-s0-sn (rewrite)
     (let ((sn (stepn s (mjrty-sn-t a n lst cand i k))))
       (implies (and (mjrty-s0p s a n lst cand i k)
                     (or (zerop (mjrty-k n lst cand i k))
                         (lessp (quotient n 2) (mjrty-k n lst cand i k))))
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (iread-dn 32 0 sn) (mjrty-cand n lst cand i k))
                     (equal (iread-dn 32 1 sn)
                            (if (lessp (quotient n 2) (mjrty-k n lst cand i k))
                                1 0))
                     (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))
                     (equal (read-mem x (mc-mem sn) l)
```

```
                          (read-mem x (mc-mem s) l)))))
     ((induct (mjrty-cand-induct s n lst cand i k))
      (disable mjrty-s0p linked-rts-addr rts-addr linked-a6 iread-dn)))

(prove-lemma mjrty-s0-sn-rfile (rewrite)
     (let ((sn (stepn s (mjrty-sn-t a n lst cand i k))))
       (implies (and (mjrty-s0p s a n lst cand i k)
                     (or (zerop (mjrty-k n lst cand i k))
                         (lessp (quotient n 2) (mjrty-k n lst cand i k)))
                     (d2-7a2-5p rn)
                     (leq oplen 32))
                (equal (read-rn oplen rn (mc-rfile sn))
                       (if (d6-7a2-5p rn)
                           (read-rn oplen rn (mc-rfile s))
                         (get-vlst oplen 0 rn '(2 3 4 5)
                                   (movem-saved s 4 16 4)))))))
     ((induct (mjrty-cand-induct s n lst cand i k))
      (disable mjrty-s0p)))

; s0 --> s1.
; base case:
(prove-lemma mjrty-s0-s1-base1 (rewrite)
     (implies (and (mjrty-s0p s a n lst cand i k)
                   (not (lessp i n))
                   (not (zerop k))
                   (not (lessp (quotient n 2) k))
                   (equal cand (get-nth 0 lst)))
              (and (mjrty-s1p (stepn s 18) a n lst cand 1 1)
                   (equal (linked-rts-addr (stepn s 18)) (linked-rts-addr s))
                   (equal (linked-a6 (stepn s 18)) (linked-a6 s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 18)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (movem-saved (stepn s 18) 4 16 4)
                          (movem-saved s 4 16 4))
                   (equal (read-mem x (mc-mem (stepn s 18)) l)
                          (read-mem x (mc-mem s) l))))
     ((disable times)
      (enable iquotient)))

(prove-lemma mjrty-s0-s1-base2 (rewrite)
     (implies (and (mjrty-s0p s a n lst cand i k)
                   (not (lessp i n))
                   (not (zerop k))
                   (not (lessp (quotient n 2) k))
                   (not (equal cand (get-nth 0 lst))))
              (and (mjrty-s1p (stepn s 17) a n lst cand 1 0)
                   (equal (linked-rts-addr (stepn s 17)) (linked-rts-addr s))
                   (equal (linked-a6 (stepn s 17)) (linked-a6 s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 17)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (movem-saved (stepn s 17) 4 16 4)
                          (movem-saved s 4 16 4))
                   (equal (read-mem x (mc-mem (stepn s 17)) l)
                          (read-mem x (mc-mem s) l))))
```

```
     ((disable times)
      (enable iquotient)))

(prove-lemma mjrty-s0-s1-rfile-base1 (rewrite)
     (implies (and (mjrty-s0p s a n lst cand i k)
                   (not (lessp i n))
                   (not (zerop k))
                   (not (lessp (quotient n 2) k))
                   (equal cand (get-nth 0 lst))
                   (d6-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 18)))
                     (read-rn oplen rn (mc-rfile s))))
     ((disable times)
      (enable iquotient)))

(prove-lemma mjrty-s0-s1-rfile-base2 (rewrite)
     (implies (and (mjrty-s0p s a n lst cand i k)
                   (not (lessp i n))
                   (not (zerop k))
                   (not (lessp (quotient n 2) k))
                   (not (equal cand (get-nth 0 lst)))
                   (d6-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 17)))
                     (read-rn oplen rn (mc-rfile s))))
     ((enable iquotient)))

; the proof of s0 --> s1.
(prove-lemma mjrty-s0-s1 (rewrite)
     (let ((s1 (stepn s (mjrty-cand-t a n lst cand i k))))
       (implies (and (mjrty-s0p s a n lst cand i k)
                     (not (zerop (mjrty-k n lst cand i k)))
                     (not (lessp (quotient n 2) (mjrty-k n lst cand i k)))
                     (equal cand0 (mjrty-cand n lst cand i k))
                     (equal k0 (if (equal (mjrty-cand n lst cand i k)
                                          (get-nth 0 lst))
                                   1 0)))
                (and (mjrty-s1p s1 a n lst cand0 1 k0)
                     (equal (linked-rts-addr s1) (linked-rts-addr s))
                     (equal (linked-a6 s1) (linked-a6 s))
                     (equal (read-rn 32 14 (mc-rfile s1))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (movem-saved s1 4 16 4)
                            (movem-saved s 4 16 4))
                     (equal (read-mem x (mc-mem s1) l)
                            (read-mem x (mc-mem s) l))))))
     ((induct (mjrty-cand-induct s n lst cand i k))
      (disable mjrty-s0p mjrty-s1p movem-saved linked-rts-addr linked-a6)))

(prove-lemma mjrty-s0-s1-rfile (rewrite)
     (let ((s1 (stepn s (mjrty-cand-t a n lst cand i k))))
       (implies (and (mjrty-s0p s a n lst cand i k)
                     (not (zerop (mjrty-k n lst cand i k)))
                     (not (lessp (quotient n 2) (mjrty-k n lst cand i k)))
                     (d6-7a2-5p rn))
```

```
                (equal (read-rn oplen rn (mc-rfile s1))
                       (read-rn oplen rn (mc-rfile s)))))
     ((induct (mjrty-cand-induct s n lst cand i k))
      (disable mjrty-s0p)))

; s1 --> exit.
; base case.
(prove-lemma mjrty-s1-sn-1 (rewrite)
     (implies (and (mjrty-s1p s a n lst cand i k)
                   (not (lessp i n))
                   (lessp (quotient n 2) k))
              (and (equal (mc-status (stepn s 14)) 'running)
                   (equal (mc-pc (stepn s 14)) (linked-rts-addr s))
                   (equal (iread-dn 32 0 (stepn s 14)) cand)
                   (equal (iread-dn 32 1 (stepn s 14)) 1)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 14)))
                          (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 14)))
                          (add 32 (read-an 32 6 s) 8))
                   (equal (read-mem x (mc-mem (stepn s 14)) l)
                          (read-mem x (mc-mem s) l))))
     ((enable iquotient)))

(prove-lemma mjrty-s1-sn-rfile-1 (rewrite)
     (implies (and (mjrty-s1p s a n lst cand i k)
                   (not (lessp i n))
                   (lessp (quotient n 2) k)
                   (d2-7a2-5p rn)
                   (leq oplen 32))
              (equal (read-rn oplen rn (mc-rfile (stepn s 14)))
                     (if (d6-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (get-vlst oplen 0 rn '(2 3 4 5)
                                 (movem-saved s 4 16 4)))))
     ((enable iquotient)))

(prove-lemma mjrty-s1-sn-2 (rewrite)
     (implies (and (mjrty-s1p s a n lst cand i k)
                   (not (lessp i n))
                   (not (lessp (quotient n 2) k)))
              (and (equal (mc-status (stepn s 13)) 'running)
                   (equal (mc-pc (stepn s 13)) (linked-rts-addr s))
                   (equal (iread-dn 32 0 (stepn s 13)) cand)
                   (equal (iread-dn 32 1 (stepn s 13)) 0)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 13)))
                          (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 13)))
                          (add 32 (read-an 32 6 s) 8))
                   (equal (read-mem x (mc-mem (stepn s 13)) l)
                          (read-mem x (mc-mem s) l))))
     ((enable iquotient)))

(prove-lemma mjrty-s1-sn-rfile-2 (rewrite)
     (implies
```

```
      (and (mjrty-s1p s a n lst cand i k)
           (not (lessp i n))
           (not (lessp (quotient n 2) k))
           (d2-7a2-5p rn)
           (leq oplen 32))
      (equal (read-rn oplen rn (mc-rfile (stepn s 13)))
             (if (d6-7a2-5p rn)
                 (read-rn oplen rn (mc-rfile s))
               (get-vlst oplen 0 rn '(2 3 4 5) (movem-saved s 4 16 4)))))
     ((enable iquotient)))

; induction case.
(prove-lemma mjrty-s1-s1-1 (rewrite)
     (implies (and (mjrty-s1p s a n lst cand i k)
                   (lessp i n)
                   (equal cand (get-nth i lst)))
              (and (mjrty-s1p (stepn s 6) a n lst cand (add1 i) (add1 k))
                   (equal (linked-rts-addr (stepn s 6))
                          (linked-rts-addr s))
                   (equal (linked-a6 (stepn s 6)) (linked-a6 s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 6)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (movem-saved (stepn s 6) 4 16 4)
                          (movem-saved s 4 16 4))
                   (equal (read-mem x (mc-mem (stepn s 6)) l)
                          (read-mem x (mc-mem s) l))))
     ((disable times lessp)))

(prove-lemma mjrty-s1-s1-2 (rewrite)
     (implies (and (mjrty-s1p s a n lst cand i k)
                   (lessp i n)
                   (not (equal cand (get-nth i lst))))
              (and (mjrty-s1p (stepn s 5) a n lst cand (add1 i) k)
                   (equal (linked-rts-addr (stepn s 5))
                          (linked-rts-addr s))
                   (equal (linked-a6 (stepn s 5)) (linked-a6 s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 5)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (movem-saved (stepn s 5) 4 16 4)
                          (movem-saved s 4 16 4))
                   (equal (read-mem x (mc-mem (stepn s 5)) l)
                          (read-mem x (mc-mem s) l))))
     ((disable times lessp)))

(prove-lemma mjrty-s1-s1-rfile-1 (rewrite)
     (implies (and (mjrty-s1p s a n lst cand i k)
                   (lessp i n)
                   (equal cand (get-nth i lst))
                   (d6-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 6)))
                     (read-rn oplen rn (mc-rfile s))))
     ((disable times lessp)))

(prove-lemma mjrty-s1-s1-rfile-2 (rewrite)
```

```
     (implies (and (mjrty-s1p s a n lst cand i k)
                   (lessp i n)
                   (not (equal cand (get-nth i lst)))
                   (d6-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 5)))
                     (read-rn oplen rn (mc-rfile s))))
     ((disable times lessp)))

(prove-lemma mjrty-s1-sn (rewrite)
     (let ((sn (stepn s (cand-cnt-t a n lst cand i k))))
       (implies (mjrty-s1p s a n lst cand i k)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (iread-dn 32 0 sn) cand)
                     (equal (iread-dn 32 1 sn)
                            (if (lessp (quotient n 2)
                                       (cand-cnt n lst cand i k))
                                1 0))
                     (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))
                     (equal (read-mem x (mc-mem sn) l)
                            (read-mem x (mc-mem s) l))))))
     ((induct (cand-cnt-induct s n lst cand i k))
      (disable mjrty-s1p linked-rts-addr rts-addr linked-a6 iread-dn)))

(prove-lemma mjrty-s1-sn-rfile (rewrite)
     (let ((sn (stepn s (cand-cnt-t a n lst cand i k))))
       (implies (and (mjrty-s1p s a n lst cand i k)
                     (d2-7a2-5p rn)
                     (leq oplen 32))
                (equal (read-rn oplen rn (mc-rfile sn))
                       (if (d6-7a2-5p rn)
                           (read-rn oplen rn (mc-rfile s))
                         (get-vlst oplen 0 rn '(2 3 4 5)
                                   (movem-saved s 4 16 4)))))))
     ((induct (cand-cnt-induct s n lst cand i k))
      (disable mjrty-s1p)))

; the correctness of MJRTY.
(prove-lemma mjrty-statep-info ()
     (implies (mjrty-statep s a n lst)
              (not (zerop n))))

(prove-lemma mjrty-correctness (rewrite)
     (let ((sn (stepn s (mjrty-t a n lst))))
       (implies (mjrty-statep s a n lst)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-sp s) 4))
                     (implies (and (d2-7a2-5p rn)
```

```
                                  (leq oplen 32))
                             (equal (read-rn oplen rn (mc-rfile sn))
                                    (read-rn oplen rn (mc-rfile s))))
                    (implies (disjoint (sub 32 20 (read-sp s)) 32 x k)
                             (equal (read-mem x (mc-mem sn) k)
                                    (read-mem x (mc-mem s) k)))
                    (equal (iread-dn 32 0 sn) (mjrty-cand n lst 0 0 0))
                    (equal (iread-dn 32 1 sn)
                           (if (mjrty-p n lst 0 0 0) 1 0)))))
     ((use (mjrty-statep-info))
      (disable iread-dn linked-a6 linked-rts-addr mjrty-statep mjrty-s0p
               mjrty-s1p)))

(disable mjrty-t)

; in the logic, mjrty is expected to have these properties:
;    1.  mjrty-thm-1: if mjrty-p returns 1,  cand wins the majority.
;    2.  mjrty-thm-2: if mjrty-p returns 0,  no one wins the majority.
(prove-lemma mjrty-cand-0 (rewrite)
     (equal (mjrty-cand n lst x n k) x)
     ((expand (mjrty-cand n lst x n k))))

(prove-lemma mjrty-cand-1 (rewrite)
     (equal (mjrty-cand (add1 n) lst x n k)
            (if (zerop k) (get-nth n lst) x))
     ((expand (mjrty-cand (add1 n) lst x n k))))

(prove-lemma mjrty-k-0 (rewrite)
     (equal (mjrty-k n lst x n k) k)
     ((expand (mjrty-k n lst x n k))))

(prove-lemma mjrty-k-1 (rewrite)
     (equal (mjrty-k (add1 n) lst x n k)
            (if (zerop k)
                1
              (if (equal x (get-nth n lst)) (add1 k) (sub1 k))))
     ((expand (mjrty-k (add1 n) lst x n k))))

(prove-lemma cand-cnt-0 (rewrite)
     (equal (cand-cnt n lst x n k) k)
     ((expand (cand-cnt n lst x n k))))

(prove-lemma cand-cnt-1 (rewrite)
     (equal (cand-cnt (add1 n) lst x n k)
            (if (equal x (get-nth n lst)) (add1 k) k))
     ((expand (cand-cnt (add1 n) lst x n k))))

(prove-lemma mjrty-k-lemma (rewrite)
     (implies (and (leq i n)
                   (leq j i)
                   (numberp i))
              (equal (mjrty-k n lst (mjrty-cand i lst x j k) i
                              (mjrty-k i lst x j k))
                     (mjrty-k n lst x j k)))
```

```
     ((enable get-nth)))

(prove-lemma mjrty-cand-lemma (rewrite)
     (implies (and (leq i n)
                   (leq j i)
                   (numberp i))
              (equal (mjrty-cand n lst (mjrty-cand i lst x j k) i
                                 (mjrty-k i lst x j k))
                     (mjrty-cand n lst x j k)))
     ((enable get-nth)))

(prove-lemma cand-cnt-lemma (rewrite)
     (implies (and (leq i n)
                   (leq j i)
                   (numberp i))
              (equal (cand-cnt n lst x i (cand-cnt i lst x j k))
                     (cand-cnt n lst x j k)))
     ((enable get-nth)))

(prove-lemma mjrty-cand-rec (rewrite)
     (implies (numberp n)
              (equal (mjrty-cand (add1 n) lst x 0 0)
                     (mjrty-cand (add1 n) lst (mjrty-cand n lst x 0 0) n
                                 (mjrty-k n lst x 0 0)))))

(prove-lemma mjrty-k-rec (rewrite)
     (implies (numberp n)
              (equal (mjrty-k (add1 n) lst x 0 0)
                     (mjrty-k (add1 n) lst (mjrty-cand n lst x 0 0) n
                              (mjrty-k n lst x 0 0)))))

(prove-lemma cand-cnt-rec (rewrite)
     (implies (numberp n)
              (equal (cand-cnt (add1 n) lst x 0 0)
                     (cand-cnt (add1 n) lst x n (cand-cnt n lst x 0 0)))))

(disable mjrty-cand-lemma)
(disable mjrty-k-lemma)
(disable cand-cnt-lemma)

(prove-lemma mjrty-lemma1 (rewrite)
     (not (lessp (cand-cnt n lst (mjrty-cand n lst 0 0 0) 0 0)
                 (mjrty-k n lst 0 0 0)))
     ((induct (plus n y))))

(defn mjrty-lemma2-induct (n lst x)
  (if (zerop n)
      t
    (and (mjrty-lemma2-induct (sub1 n) lst x)
         (mjrty-lemma2-induct (sub1 n) lst (get-nth (sub1 n) lst)))))

(prove-lemma mjrty-lemma2 (rewrite)
     (and (not (lessp (plus n (mjrty-k n lst 0 0 0))
                      (times 2 (cand-cnt n lst (mjrty-cand n lst 0 0 0) 0 0))))
```

```
          (implies (not (equal x (mjrty-cand n lst 0 0 0)))
                   (not (lessp n
                               (plus (mjrty-k n lst 0 0 0)
                                     (times 2 (cand-cnt n lst x 0 0)))))))
     ((induct (mjrty-lemma2-induct n lst x))))

(disable mjrty-cand-rec)
(disable mjrty-k-rec)
(disable cand-cnt-rec)

(prove-lemma mjrty-thm1 (rewrite)
     (implies (mjrty-p n lst 0 0 0)
              (lessp (quotient n 2)
                     (cand-cnt n lst (mjrty-cand n lst 0 0 0) 0 0))))

(prove-lemma mjrty-thm2 (rewrite)
     (implies (not (mjrty-p n lst 0 0 0))
              (not (lessp (quotient n 2) (cand-cnt n lst x 0 0))))
     ((use (mjrty-lemma2))))

; a simple time analysis.
(prove-lemma mjrty-t-crock (rewrite)
     (equal (times z (difference (sub1 x) y))
            (difference (times z (difference x y)) z)))

(prove-lemma mjrty-cand-t-0 (rewrite)
     (and (equal (mjrty-cand-t a 0 lst cand i k)
                 (if (equal cand (get-nth 0 lst)) 18 17))
          (equal (mjrty-cand-t a 1 lst cand 1 k)
                 (if (equal cand (get-nth 0 lst)) 18 17))))

(prove-lemma mjrty-cand-t-1 (rewrite)
     (equal (mjrty-cand-t a 1 lst cand i k)
            (if (zerop i)
                (if (zerop k)
                    26
                  (if (equal cand (get-nth i lst))
                      (if (equal cand (get-nth 0 lst)) 27 26)
                    (if (equal cand (get-nth 0 lst)) 26 25)))
              (if (equal cand (get-nth 0 lst)) 18 17)))
     ((expand (mjrty-cand-t a 1 lst cand i k))
      (enable get-nth-0)))

(prove-lemma mjrty-cand-t-ubound (rewrite)
    (not (lessp (plus 18 (times 9 (difference n i)))
                (mjrty-cand-t a n lst cand i k)))
     ((enable splus times)
      (expand (mjrty-cand-t a 1 lst cand i k))))

(prove-lemma mjrty-sn-t-ubound (rewrite)
     (not (lessp (plus 17 (times 9 (difference n i)))
                 (mjrty-sn-t a n lst cand i k)))
     ((enable splus times)))
```

```
(prove-lemma cand-cnt-t-0 (rewrite)
     (and (equal (cand-cnt-t a 0 lst cand i k)
                 (if (lessp 0 k) 14 13))
          (equal (cand-cnt-t a n lst cand n k)
                 (if (lessp (quotient n 2) k) 14 13)))
     ((expand (cand-cnt-t a n lst cand n k))))

(prove-lemma cand-cnt-t-1 (rewrite)
     (equal (cand-cnt-t a 1 lst cand i k)
            (if (zerop i)
                (if (equal cand (get-nth i lst))
                    20
                  (if (lessp 0 k) 19 18))
              (if (lessp 0 k) 14 13)))
     ((expand (cand-cnt-t a 1 lst cand i k))))

(prove-lemma cand-cnt-t-ubound (rewrite)
     (not (lessp (plus 14 (times 6 (difference n i)))
                 (cand-cnt-t a n lst cand i k)))
     ((enable splus times)))

(prove-lemma mjrty-t-ubound ()
     (leq (mjrty-t a n lst)
          (plus 46 (times 15 (sub1 n))))
     ((enable splus mjrty-t)))
```

C.6 A Case Study of Subroutine Call

```
;            Proof of the Correctness of a GCD Program
#|

The following C program computes the greatest common divisor of
three nonnegative integers a, b and c. We investigate the machine
code of this program generated by a widely used C compiler gcc,
and verify the correctness of the code.  The aim here is to see
how to handle subroutine calls.

gcd3(a, b, c)
long int a, b, c;
{
  gcd(gcd(a, b), c);
}

Here is the MC68020 assembly code of the above GCD program.  The code is
generated by gcc.

0x2324 <gcd3>:          linkw a6,#0
0x2328 <gcd3+4>:        movel a2,sp@-
0x232a <gcd3+6>:        movel a6@(16),sp@-
0x232e <gcd3+10>:       movel a6@(12),sp@-
0x2332 <gcd3+14>:       movel a6@(8),sp@-
0x2336 <gcd3+18>:       lea @#0x2350 <gcd>,a2
```

```
0x233c <gcd3+24>:       jsr a2@
0x233e <gcd3+26>:       addqw #8,sp
0x2340 <gcd3+28>:       movel d0,sp@-
0x2342 <gcd3+30>:       jsr a2@
0x2344 <gcd3+32>:       moveal a6@(-4),a2
0x2348 <gcd3+36>:       unlk a6
0x234a <gcd3+38>:       rts

The machine code of the above program is:

<gcd3>:       0x4e56  0x0000  0x2f0a  0x2f2e  0x0010  0x2f2e  0x000c  0x2f2e
<gcd3+16>:    0x0008  0x45f9  0x0000  0x2350  0x4e92  0x504f  0x2f00  0x4e92
<gcd3+32>:    0x246e  0xfffc  0x4e5e  0x4e75

'(78      86      0       0       47      10      47      46
  0       16      47      46      0       12      47      46
  0       8       69      249     0       0       35      80
  78      146     80      79      47      0       78      146
  36      110     255     252     78      94      78      117)
|#

; now we start to verify this GCD3 program,  defined by (gcd3-code).
(defn gcd3-code ()
  '(78      86      0       0       47      10      47      46
    0       16      47      46      0       12      47      46
    0       8       69      249     -1      -1      -1      -1
    78      146     80      79      47      0       78      146
    36      110     255     252     78      94      78      117))

(constrain gcd3-load (rewrite)
   (equal (gcd3-loadp s)
          (and (evenp (gcd3-addr))
               (numberp (gcd3-addr))
               (nat-rangep (gcd3-addr) 32)
               (rom-addrp (gcd3-addr) (mc-mem s) 40)
               (mcode-addrp (gcd3-addr) (mc-mem s) (gcd3-code))
               (gcd-loadp s)
               (equal (pc-read-mem (add 32 (gcd3-addr) 20) (mc-mem s) 4)
                      (gcd-addr))))
    ((gcd3-loadp (lambda (s) f))
     (gcd3-addr (lambda () 1))))

(prove-lemma stepn-gcd3-loadp (rewrite)
     (equal (gcd3-loadp (stepn s n))
            (gcd3-loadp s)))

(defn gcd3 (a b c)
  (gcd (gcd a b) c))

(defn gcd3-t0 (a b c)
  7)

(defn gcd3-t1 (a b c)
  (gcd-t a b))
```

```
(defn gcd3-t2 (a b c)
  3)

(defn gcd3-t3 (a b c)
  (gcd-t (gcd a b) c))

(defn gcd3-t4 (a b c)
  3)

(defn gcd3-t (a b c)
  (splus (gcd3-t0 a b c)
         (splus (gcd3-t1 a b c)
                (splus (gcd3-t2 a b c)
                       (splus (gcd3-t3 a b c)
                              (gcd3-t4 a b c))))))

; the initial state.
(defn gcd3-statep (s a b c)
  (and (equal (mc-status s) 'running)
       (gcd3-loadp s)
       (equal (mc-pc s) (gcd3-addr))
       (ram-addrp (sub 32 36 (read-sp s)) (mc-mem s) 52)
       (equal a (iread-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal b (iread-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
       (equal c (iread-mem (add 32 (read-sp s) 12) (mc-mem s) 4))
       (lessp 0 a)
       (lessp 0 b)
       (lessp 0 c)))

; the state after the execution of the first JSR instruction, but before
; the execution of the subroutine GCD.
(defn gcd3-s0p (s a b c)
  (and (equal (mc-status s) 'running)
       (gcd3-loadp s)
       (equal (mc-pc s) (gcd-addr))
       (equal (read-an 32 2 s) (gcd-addr))
       (equal (rts-addr s) (add 32 (gcd3-addr) 26))
       (ram-addrp (sub 32 12 (read-sp s)) (mc-mem s) 52)
       (equal* (read-an 32 6 s) (add 32 (read-sp s) 20))
       (equal a (iread-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal b (iread-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
       (equal c (iread-mem (add 32 (read-sp s) 12) (mc-mem s) 4))
       (lessp 0 a)
       (lessp 0 b)
       (lessp 0 c)))

; the state right after return from the first call to subroutine GCD.
(defn gcd3-s1p (s a b c)
  (and (equal (mc-status s) 'running)
       (gcd3-loadp s)
       (equal (read-an 32 2 s) (gcd-addr))
       (equal (mc-pc s) (add 32 (gcd3-addr) 26))
       (ram-addrp (sub 32 16 (read-sp s)) (mc-mem s) 52)
```

```
       (equal* (read-an 32 6 s) (add 32 (read-sp s) 16))
       (equal (iread-dn 32 0 s) (gcd a b))
       (equal c (iread-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
       (lessp 0 a)
       (lessp 0 b)
       (lessp 0 c)))

; the state after the execution of the second JSR, but before the
; execution of the subroutine GCD.
(defn gcd3-s2p (s a b c)
  (and (equal (mc-status s) 'running)
       (gcd3-loadp s)
       (equal (mc-pc s) (gcd-addr))
       (equal (rts-addr s) (add 32 (gcd3-addr) 32))
       (ram-addrp (sub 32 16 (read-sp s)) (mc-mem s) 52)
       (equal* (read-an 32 6 s) (add 32 (read-sp s) 16))
       (equal (gcd a b) (iread-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal c (iread-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
       (lessp 0 a)
       (lessp 0 b)
       (lessp 0 c)))

; the state returned from the second call to the subroutine GCD.
(defn gcd3-s3p (s a b c)
  (and (equal (mc-status s) 'running)
       (gcd3-loadp s)
       (equal (mc-pc s) (add 32 (gcd3-addr) 32))
       (ram-addrp (sub 32 12 (read-sp s)) (mc-mem s) 44)
       (equal* (read-an 32 6 s) (add 32 (read-sp s) 12))
       (equal (gcd (gcd a b) c) (iread-dn 32 0 s))
       (lessp 0 a)
       (lessp 0 b)
       (lessp 0 c)))

; from the initial state to s0.
(prove-lemma gcd3-s-s0 (rewrite)
     (let ((s0 (stepn s (gcd3-t0 a b c))))
       (implies (gcd3-statep s a b c)
                (and (gcd3-s0p s0 a b c)
                     (equal (linked-rts-addr s0) (rts-addr s))
                     (equal (linked-a6 s0) (read-an 32 6 s))
                     (equal (read-rn 32 14 (mc-rfile s0))
                            (sub 32 4 (read-sp s)))
                     (equal (rn-saved s0) (read-an 32 2 s))))))

(prove-lemma gcd3-s-s0-rfile (rewrite)
     (implies (and (gcd3-statep s a b c)
                   (d2-7a3-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s (gcd3-t0 a b c))))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma gcd3-s-s0-mem (rewrite)
     (implies (and (gcd3-statep s a b c)
                   (disjoint x k (sub 32 36 (read-sp s)) 52))
```

```
              (equal (read-mem x (mc-mem (stepn s (gcd3-t0 a b c))) k)
                     (read-mem x (mc-mem s) k))))

; from s0 to s1.
(prove-lemma gcd3-s0-s1 (rewrite)
     (let ((s1 (stepn s (gcd3-t1 a b c))))
       (implies (gcd3-s0p s a b c)
                (and (gcd3-s1p s1 a b c)
                     (equal (linked-rts-addr s1) (linked-rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile s1))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (linked-a6 s1) (linked-a6 s))
                     (equal (rn-saved s1) (rn-saved s))))))
     ((disable rts-addr iread-dn)
      (enable gcd-statep)))

(prove-lemma gcd3-s0-s1-rfile (rewrite)
     (implies (and (gcd3-s0p s a b c)
                   (d2-7a2-5p rn)
                   (leq oplen 32))
              (equal (read-rn oplen rn (mc-rfile (stepn s (gcd3-t1 a b c))))
                     (read-rn oplen rn (mc-rfile s))))
     ((enable gcd-statep)))

(prove-lemma gcd3-s0-s1-mem (rewrite)
     (implies (and (gcd3-s0p s a b c)
                   (disjoint x k (sub 32 32 (read-an 32 6 s)) 52))
              (equal (read-mem x (mc-mem (stepn s (gcd3-t1 a b c))) k)
                     (read-mem x (mc-mem s) k)))
     ((enable gcd-statep)))

; from s1 to s2.
(prove-lemma gcd3-s1-s2 (rewrite)
     (let ((s2 (stepn s (gcd3-t2 a b c))))
       (implies (gcd3-s1p s a b c)
                (and (gcd3-s2p s2 a b c)
                     (equal (linked-rts-addr s2) (linked-rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile s2))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (linked-a6 s2) (linked-a6 s))
                     (equal (rn-saved s2) (rn-saved s))))))

(prove-lemma gcd3-s1-s2-rfile (rewrite)
     (implies (and (gcd3-s1p s a b c)
                   (d2-7a3-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s (gcd3-t2 a b c))))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma gcd3-s1-s2-mem (rewrite)
     (implies (and (gcd3-s1p s a b c)
                   (disjoint x k (sub 32 32 (read-an 32 6 s)) 52))
              (equal (read-mem x (mc-mem (stepn s (gcd3-t2 a b c))) k)
                     (read-mem x (mc-mem s) k))))
```

```
; from s2 to s3.
(prove-lemma gcd-nonzero (rewrite)
     (implies (and (not (zerop a))
                   (not (zerop b)))
              (not (equal (gcd a b) 0))))

(prove-lemma gcd3-s2-s3 (rewrite)
     (let ((s3 (stepn s (gcd3-t3 a b c))))
       (implies (gcd3-s2p s a b c)
                (and (gcd3-s3p s3 a b c)
                     (equal (linked-rts-addr s3) (linked-rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile s3))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (linked-a6 s3) (linked-a6 s))
                     (equal (rn-saved s3) (rn-saved s))))))
     ((disable rts-addr iread-dn)
      (enable gcd-statep)))

(prove-lemma gcd3-s2-s3-rfile (rewrite)
     (implies (and (gcd3-s2p s a b c)
                   (d2-7a2-5p rn)
                   (leq oplen 32))
              (equal (read-rn oplen rn (mc-rfile (stepn s (gcd3-t3 a b c))))
                     (read-rn oplen rn (mc-rfile s))))
     ((enable gcd-statep)))

(prove-lemma gcd3-s2-s3-mem (rewrite)
     (implies (and (gcd3-s2p s a b c)
                   (disjoint x k (sub 32 32 (read-an 32 6 s)) 52))
              (equal (read-mem x (mc-mem (stepn s (gcd3-t3 a b c))) k)
                     (read-mem x (mc-mem s) k)))
     ((enable gcd-statep)))

; from s3 to exit.
(prove-lemma gcd3-s3-sn (rewrite)
     (let ((sn (stepn s (gcd3-t4 a b c))))
       (implies (gcd3-s3p s a b c)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (iread-dn 32 0 sn) (gcd (gcd a b) c))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))))))

(prove-lemma gcd3-s3-sn-rfile (rewrite)
     (implies (and (gcd3-s3p s a b c)
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s (gcd3-t4 a b c))))
                     (if (equal rn 10)
                         (head (rn-saved s) oplen)
                       (read-rn oplen rn (mc-rfile s))))))
```

```
(prove-lemma gcd3-s3-sn-mem (rewrite)
     (implies (and (gcd3-s3p s a b c)
                   (disjoint x k (sub 32 32 (read-an 32 6 s)) 52))
              (equal (read-mem x (mc-mem (stepn s (gcd3-t4 a b c))) k)
                     (read-mem x (mc-mem s) k))))

(disable gcd3-t0)
(disable gcd3-t1)
(disable gcd3-t2)
(disable gcd3-t3)
(disable gcd3-t4)

; the correctness of the program GCD3.
(prove-lemma gcd3-correctness (rewrite)
     (let ((sn (stepn s (gcd3-t a b c))))
       (implies (gcd3-statep s a b c)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-rn 32 15 (mc-rfile s)) 4))
                     (implies (and (leq oplen 32)
                                   (d2-7a2-5p rn))
                              (equal (read-rn oplen rn (mc-rfile sn))
                                     (read-rn oplen rn (mc-rfile s))))
                     (implies (disjoint x k (sub 32 36 (read-sp s)) 52)
                              (equal (read-mem x (mc-mem sn) k)
                                     (read-mem x (mc-mem s) k)))
                     (equal (iread-dn 32 0 sn) (gcd (gcd a b) c))))))
     ((disable iread-dn gcd3-statep linked-rts-addr)))

(disable gcd3-t)

; in the logic, the function gcd3 does computes the greatest common divisor
; of its three arguments.
(prove-lemma remainder-trans (rewrite)
             (implies (and (equal (remainder a b) 0)
                           (equal (remainder b c) 0))
                      (equal (remainder a c) 0)))

(prove-lemma gcd3-is-cd (rewrite)
             (and (equal (remainder a (gcd3 a b c)) 0)
                  (equal (remainder b (gcd3 a b c)) 0)
                  (equal (remainder c (gcd3 a b c)) 0))
             ((use (remainder-trans (b (gcd a b)) (c (gcd3 a b c)))
                   (remainder-trans (a b) (b (gcd a b)) (c (gcd3 a b c))))))

(prove-lemma cd-divides-gcd ()
             (implies (and (equal (remainder a x) 0)
                           (equal (remainder b x) 0))
                      (equal (remainder (gcd a b) x) 0))
             ((induct (gcd a b))
              (enable remainder-difference)))
```

```
(prove-lemma gcd3-the-greatest (rewrite)
             (implies (and (not (zerop a))
                           (not (zerop b))
                           (not (zerop c))
                           (equal (remainder a x) 0)
                           (equal (remainder b x) 0)
                           (equal (remainder c x) 0))
                      (not (lessp (gcd3 a b c) x)))
             ((use (cd-divides-gcd))))

(disable remainder-trans)
```

C.7 A Case Study of Switch Statement

```
;               Case study: Switch Statement
#|

The purpose of this trivial C function here is to study the switch construct
in C.\index{switch@switch statement}

int foo(int n)
{
  int i;

  switch(n) {
  case 0: i = 0; break;
  case 1: i = 1; break;
  case 2: i = 4; break;
  case 3: i = 9; break;
  case 4: i = 16; break;
  default: i = n; break;
  };
  return i;
}

Here is the MC68020 assembly code of the above function.  The code is
generated by gcc with optimization option.

0x23b2 <foo>:      linkw a6,#0
0x23b6 <foo+4>:    movel a6@(8),d0
0x23ba <foo+8>:    movel #4,d1
0x23bc <foo+10>:   cmpl d1,d0
0x23be <foo+12>:   bhi 0x23e4 <foo+50>
0x23c0 <foo+14>:   movew 0x23c8[d0.l*2],d1
0x23c4 <foo+18>:   jmp 0x23c8[d1.w]
0x23c8 <foo+22>:   orb #14,a2
0x23cc <foo+26>:   orb #22,a2@
0x23d0 <foo+30>:   orb #-128,a2@+
0x23d4 <foo+34>:   bra 0x23e4 <foo+50>
0x23d6 <foo+36>:   movel #1,d0
0x23d8 <foo+38>:   bra 0x23e4 <foo+50>
```

```
0x23da <foo+40>:   movel #4,d0
0x23dc <foo+42>:   bra 0x23e4 <foo+50>
0x23de <foo+44>:   movel #9,d0
0x23e0 <foo+46>:   bra 0x23e4 <foo+50>
0x23e2 <foo+48>:   movel #16,d0
0x23e4 <foo+50>:   unlk a6
0x23e6 <foo+52>:   rts

The machine code of the above program is:

<foo>:      0x4e56  0x0000  0x202e  0x0008  0x7204  0xb081  0x6224  0x323b
<foo+16>:   0x0a06  0x4efb  0x1002  0x000a  0x000e  0x0012  0x0016  0x001a
<foo+32>:   0x4280  0x600e  0x7001  0x600a  0x7004  0x6006  0x7009  0x6002
<foo+48>:   0x7010  0x4e5e  0x4e75

'(78      86      0       0       32      46      0       8
  114     4       176     129     98      36      50      59
  10      6       78      251     16      2       0       10
  0       14      0       18      0       22      0       26
  66      128     96      14      112     1       96      10
  112     4       96      6       112     9       96      2
  112     16      78      94      78      117)
|#

; in the logic, the above program is specified as (foo-code).
(defn foo-code ()
  '(78      86      0       0       32      46      0       8
    114     4       176     129     98      36      50      59
    10      6       78      251     16      2       0       10
    0       14      0       18      0       22      0       26
    66      128     96      14      112     1       96      10
    112     4       96      6       112     9       96      2
    112     16      78      94      78      117))

(defn foo (n)
  (if (between-ileq 0 n 4)
      (times n n)
    n))

(defn foo-t (n)
  (if (or (equal n 0)
          (equal n 1)
          (equal n 2)
          (equal n 3))
      11
    (if (equal n 4)
        10
      7)))

(defn foo-statep (s n)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (mc-pc s) (mc-mem s) 54)
       (mcode-addrp (mc-pc s) (mc-mem s) (foo-code))
```

```
       (ram-addrp (sub 32 4 (read-sp s)) (mc-mem s) 12)
       (disjoint (mc-pc s) 54 (sub 32 4 (read-sp s)) 12)
       (equal n (iread-mem (add 32 (read-sp s) 4) (mc-mem s) 4))))

(defn foo-snp (s sn n oplen rn x k)
  (and (equal (mc-status sn) 'running)
       (equal (mc-pc sn) (rts-addr s))
       (equal (iread-dn 32 0 sn) (foo n))
       (equal (read-rn 32 14 (mc-rfile sn))
              (read-rn 32 14 (mc-rfile s)))
       (equal (read-rn 32 15 (mc-rfile sn))
              (add 32 (read-an 32 7 s) 4))
       (equal (read-rn oplen rn (mc-rfile sn))
              (read-rn oplen rn (mc-rfile s)))
       (equal (read-mem x (mc-mem sn) k)
              (read-mem x (mc-mem s) k)))))

(prove-lemma foo-s-sn ()
     (implies (and (foo-statep s n)
                   (d2-7a2-5p rn)
                   (disjoint x k (sub 32 4 (read-sp s)) 12))
              (foo-snp s (stepn s (foo-t n)) n oplen rn x k)))

(prove-lemma foo-correctness (rewrite)
     (let ((sn (stepn s (foo-t n))))
       (implies (foo-statep s n)
                (and (equal (mc-status sn) 'running)
                   (equal (mc-pc sn) (rts-addr s))
                   (equal (read-rn 32 14 (mc-rfile sn))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (read-rn 32 15 (mc-rfile sn))
                          (add 32 (read-an 32 7 s) 4))
                   (implies (d2-7a2-5p rn)
                            (equal (read-rn oplen rn (mc-rfile sn))
                                   (read-rn oplen rn (mc-rfile s))))
                   (implies (disjoint x k (sub 32 4 (read-sp s)) 12)
                            (equal (read-mem x (mc-mem sn) k)
                                   (read-mem x (mc-mem s) k)))
                   (equal (iread-dn 32 0 sn) (foo n))))))
     ((use (foo-s-sn (rn 3) (x (mc-pc s)) (k 1))
           (foo-s-sn (x (mc-pc s)) (k 1))
           (foo-s-sn (rn 3)))
      (disable foo-t foo)))
```

C.8 A Case Study of Functional Parameters

```
;           Function Pointer Constrains with a Witness GT
#|
/* greater than relation */
gt(int a, int b)
{
  if (a == b)
```

```
    return 0;
  else if (a > b)
    return 1;
  else return -1;
}

The MC68020 assembly code of the above GCD program.  The code is generated
by "gcc -O".

0x22de <gt>:    linkw fp,#0
0x22e2 <gt+4>:  movel fp@(8),d1
0x22e6 <gt+8>:  movel fp@(12),d0
0x22ea <gt+12>: cmpl d1,d0
0x22ec <gt+14>: bne 0x22f2 <gt+20>
0x22ee <gt+16>: clrl d0
0x22f0 <gt+18>: bra 0x22fc <gt+30>
0x22f2 <gt+20>: cmpl d1,d0
0x22f4 <gt+22>: bge 0x22fa <gt+28>
0x22f6 <gt+24>: movel #1,d0
0x22f8 <gt+26>: bra 0x22fc <gt+30>
0x22fa <gt+28>: movel #-1,d0
0x22fc <gt+30>: unlk fp
0x22fe <gt+32>: rts

The machine code:

0x22de <gt>:    0x4e56  0x0000  0x222e  0x0008  0x202e  0x000c  0xb081  0x6604
0x22ee <gt+16>: 0x4280  0x600a  0xb081  0x6c04  0x7001  0x6002  0x70ff  0x4e5e
0x22fe <gt+32>: 0x4e75

'(78      86      0       0       34      46      0       8
  32      46      0       12      176     129     102     4
  66      128     96      10      176     129     108     4
  112     1       96      2       112     255     78      94
  78      117)
|#

; In Nqthm logic, gt-code defines the programs.
(defn gt-code ()
  '(78      86      0       0       34      46      0       8
    32      46      0       12      176     129     102     4
    66      128     96      10      176     129     108     4
    112     1       96      2       112     255     78      94
    78      117))

(defn gt (a b)
  (if (equal a b)
      0
    (if (ilessp b a) 1 -1)))

(defn gt-t (a b)
  (if (equal a b)
      9
    (if (ilessp b a) 11 10)))
```

```
; preconditions on the initial state.
(defn gt-statep (s a b)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (mc-pc s) (mc-mem s) 34)
       (mcode-addrp (mc-pc s) (mc-mem s) (gt-code))
       (ram-addrp (sub 32 4 (read-sp s)) (mc-mem s) 16)
       (equal a (iread-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal b (iread-mem (add 32 (read-sp s) 8) (mc-mem s) 4))))

; correctness.
(disable ilessp)

(prove-lemma gt-s-sn (rewrite)
     (let ((sn (stepn s (gt-t a b))))
       (implies (gt-statep s a b)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-sp s) 4))
                     (implies (d2-7a2-5p rn)
                              (equal (read-rn oplen rn (mc-rfile sn))
                                     (read-rn oplen rn (mc-rfile s))))
                     (implies (disjoint x k (sub 32 4 (read-sp s)) 4)
                              (equal (read-mem x (mc-mem sn) k)
                                     (read-mem x (mc-mem s) k)))
                     (equal (iread-dn 32 0 sn) (gt a b))))))

(disable gt-t)
(disable gt)

; function constrain with gt as witness.
(constrain p-disjointness (rewrite)
   (implies (and (p-disjoint x n s)
                 (leq (plus j (index-n y x)) n))
            (p-disjoint y j s))
   ((p-disjoint (lambda (x n s) f))))

(constrain fn*-correctness (rewrite)
   (implies (fn*-statep s a b)
            (let ((sn (stepn s (fn*-t a b))))
              (and (equal (mc-status sn) 'running)
                   (equal (mc-pc sn) (rts-addr s))
                   (equal (read-rn 32 14 (mc-rfile sn))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (read-rn 32 15 (mc-rfile sn))
                          (add 32 (read-sp s) 4))
                   (implies (and (leq oplen 32)
                                 (d2-7a2-5p rn))
                            (equal (read-rn oplen rn (mc-rfile sn))
                                   (read-rn oplen rn (mc-rfile s))))
```

```
                  (implies (p-disjoint x k s)
                           (equal (read-mem x (mc-mem sn) k)
                                  (read-mem x (mc-mem s) k)))
                  (equal (iread-dn 32 0 sn) (fn* a b)))))
   ((fn*-statep (lambda (s a b) f))
    (fn*-t (lambda (a b) 0))
    (fn* (lambda (a b) 0))))

;          Proof of the Correctness of a MAX Function
#|
The following C function max computes the maximum of integers a and b,
according to the comparison function comp.  Here, we study the problem
of function pointer.

max(int a, int b, int (*comp)(int, int))
{
  if ((*comp)(a, b) < 0)
    return b;
  else return a;
}

The MC68020 assembly code of the C function max on SUN-3 is given as
follows.  This binary is generated by "gcc -O".

0x2320 <max>:           linkw fp,#0
0x2324 <max+4>:         moveml d2-d3,sp@-
0x2328 <max+8>:         movel fp@(8),d3
0x232c <max+12>:        movel fp@(12),d2
0x2330 <max+16>:        movel d2,sp@-
0x2332 <max+18>:        movel d3,sp@-
0x2334 <max+20>:        moveal fp@(16),a0
0x2338 <max+24>:        jsr a0@
0x233a <max+26>:        tstl d0
0x233c <max+28>:        bge 0x2342 <max+34>
0x233e <max+30>:        movel d2,d0
0x2340 <max+32>:        bra 0x2344 <max+36>
0x2342 <max+34>:        movel d3,d0
0x2344 <max+36>:        moveml fp@(-8),d2-d3
0x234a <max+42>:        unlk fp
0x234c <max+44>:        rts

The machine code of the above program is:

<max>:      0x4e56  0x0000  0x48e7  0x3000  0x262e  0x0008  0x242e  0x000c
<max+16>:   0x2f02  0x2f03  0x206e  0x0010  0x4e90  0x4a80  0x6c04  0x2002
<max+32>:   0x6002  0x2003  0x4cee  0x000c  0xfff8  0x4e5e  0x4e75

'(78      86      0       0       72      231     48      0
  38      46      0       8       36      46      0       12
  47      2       47      3       32      110     0       16
  78      144     74      128     108     4       32      2
  96      2       32      3       76      238     0       12
  255     248     78      94      78      117)
|#
```

```
(defn max-code ()
  '(78      86      0       0       72      231     48      0
    38      46      0       8       36      46      0       12
    47      2       47      3       32      110     0       16
    78      144     74      128     108     4       32      2
    96      2       32      3       76      238     0       12
    255     248     78      94      78      117))

; imax is the Nqthm counterpart of the program (max-code).
(defn max-fn* (a b)
  (if (negativep (fn* a b)) b a))

; the computation time of the program.
(defn max-t0 (a b) 8)

(defn max-t (a b)
  (splus (max-t0 a b)
         (splus (fn*-t a b)
                (if (negativep (fn* a b)) 7 6))))

; the assumptions of the initial state.
(defn max-sp (s a b)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (mc-pc s) (mc-mem s) 46)
       (mcode-addrp (mc-pc s) (mc-mem s) (max-code))
       (equal a (iread-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal b (iread-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
       (ram-addrp (sub 32 24 (read-sp s)) (mc-mem s) 40)))

(constrain max-state (rewrite)
   (and (implies (max-statep s a b)
                 (fn*-statep (stepn s (max-t0 a b)) a b))
        (implies (max-statep s a b)
                 (p-disjoint (add 32
                                  (read-rn 32 15 (mc-rfile s))
                                  4294967272)
                             40
                             (stepn s (max-t0 a b))))
        (equal (max-statep s a b)
               (and (max-sp s a b)
                    (comp-loadp s a b))))
   ((max-statep (lambda (s a b) f))
    (comp-loadp (lambda (s a b) f))))

; an intermediate state.
(defn max-s0p (s a b)
  (and (equal (mc-status s) 'running)
       (equal (mc-pc s) (read-mem (add 32 (read-an 32 6 s) 16) (mc-mem s) 4))
       (evenp (rts-addr s))
       (rom-addrp (sub 32 26 (rts-addr s)) (mc-mem s) 46)
       (mcode-addrp (sub 32 26 (rts-addr s)) (mc-mem s) (max-code))
       (ram-addrp (read-sp s) (mc-mem s) 40)
       (equal* (read-sp s) (sub 32 20 (read-an 32 6 s)))
```

```
       (equal a (iread-dn 32 3 s))
       (equal b (iread-dn 32 2 s))
       (equal (iread-mem (add 32 (read-sp s) 4) (mc-mem s) 4) a)
       (equal (iread-mem (add 32 (read-sp s) 8) (mc-mem s) 4) b)))

; an intermediate state.
(defn max-s1p (s a b)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 26 (mc-pc s)) (mc-mem s) 46)
       (mcode-addrp (sub 32 26 (mc-pc s)) (mc-mem s) (max-code))
       (ram-addrp (sub 32 20 (read-an 32 6 s)) (mc-mem s) 40)
       (equal a (iread-dn 32 3 s))
       (equal b (iread-dn 32 2 s))
       (equal (iread-dn 32 0 s) (fn* a b))))

(constrain max-disjointness (rewrite)
   (implies (and (max-statep s a b)
                 (max-disjoint x k s))
            (p-disjoint x k (stepn s (max-t0 a b))))
   ((max-disjoint (lambda (x k s) f))))

; from the initial state to s0.  s --> s0.
(prove-lemma max-s-s0 ()
     (implies (max-sp s a b)
              (max-s0p (stepn s (max-t0 a b)) a b)))

(prove-lemma max-s-s0-else (rewrite)
     (let ((s0 (stepn s (max-t0 a b))))
       (implies (max-sp s a b)
                (and (equal (linked-rts-addr s0) (rts-addr s))
                     (equal (linked-a6 s0) (read-an 32 6 s))
                     (equal (read-rn 32 14 (mc-rfile s0))
                            (sub 32 4 (read-sp s)))
                     (equal (movem-saved s0 4 8 2)
                            (readm-rn 32 '(2 3) (mc-rfile s)))))))

(prove-lemma max-s-s0-rfile (rewrite)
     (implies (and (max-sp s a b)
                   (d4-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s (max-t0 a b))))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma max-s-s0-mem (rewrite)
     (implies (and (max-sp s a b)
                   (disjoint x k (sub 32 24 (read-sp s)) 24))
              (equal (read-mem x (mc-mem (stepn s (max-t0 a b))) k)
                     (read-mem x (mc-mem s) k))))

; from s0 to s1.  s0 --> s1.  This is basically a fact about the
; subroutine comp.
(prove-lemma max-s0-s1 ()
     (implies (and (max-s0p s a b)
                   (fn*-statep s a b))
```

```
              (max-s1p (stepn s (fn*-t a b)) a b)))

(prove-lemma max-s0-s1-else (rewrite)
     (implies (and (max-s0p s a b)
                   (fn*-statep s a b)
                   (p-disjoint (sub 32 20 (read-an 32 6 s)) 40 s))
              (and (equal (linked-rts-addr (stepn s (fn*-t a b)))
                          (linked-rts-addr s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s (fn*-t a b))))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (linked-a6 (stepn s (fn*-t a b)))
                          (linked-a6 s))
                   (equal (movem-saved (stepn s (fn*-t a b)) 4 8 2)
                          (movem-saved s 4 8 2)))))

(prove-lemma max-s0-s1-rfile (rewrite)
     (implies (and (max-s0p s a b)
                   (fn*-statep s a b)
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s (fn*-t a b))))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma max-s0-s1-mem (rewrite)
     (implies (and (max-s0p s a b)
                   (fn*-statep s a b)
                   (p-disjoint x k s))
              (equal (read-mem x (mc-mem (stepn s (fn*-t a b))) k)
                     (read-mem x (mc-mem s) k))))

; from s1 to exit.  s1 --> exit.
(prove-lemma max-s1-sn-1 (rewrite)
     (implies (and (max-s1p s a b)
                   (negativep (fn* a b)))
              (and (equal (mc-status (stepn s 7)) 'running)
                   (equal (mc-pc (stepn s 7)) (linked-rts-addr s))
                   (equal (iread-dn 32 0 (stepn s 7)) b)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 7)))
                          (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 7)))
                          (add 32 (read-an 32 6 s) 8))
                   (equal (read-mem x (mc-mem (stepn s 7)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma max-s1-sn-rfile-1 (rewrite)
     (implies (and (max-s1p s a b)
                   (negativep (fn* a b))
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 7)))
                     (if (d4-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (get-vlst oplen 0 rn '(2 3) (movem-saved s 4 8 2))))))
```

```
(prove-lemma max-s1-sn-2 (rewrite)
     (implies (and (max-s1p s a b)
                   (not (negativep (fn* a b))))
              (and (equal (mc-status (stepn s 6)) 'running)
                   (equal (mc-pc (stepn s 6)) (linked-rts-addr s))
                   (equal (iread-dn 32 0 (stepn s 6)) a)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 6)))
                          (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 6)))
                          (add 32 (read-an 32 6 s) 8))
                   (equal (read-mem x (mc-mem (stepn s 6)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma max-s1-sn-rfile-2 (rewrite)
     (implies (and (max-s1p s a b)
                   (not (negativep (fn* a b)))
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 6)))
                     (if (d4-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (get-vlst oplen 0 rn '(2 3) (movem-saved s 4 8 2))))))

; the correctness of max.
(disable max-t0)

(prove-lemma max-correctness (rewrite)
     (let ((sn (stepn s (max-t a b))))
       (implies (max-statep s a b)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-sp s) 4))
                     (implies (and (leq oplen 32)
                                   (d2-7a2-5p rn))
                              (equal (read-rn oplen rn (mc-rfile sn))
                                     (read-rn oplen rn (mc-rfile s))))
                     (implies (and (disjoint x k (sub 32 24 (read-sp s)) 40)
                                   (max-disjoint x k s))
                              (equal (read-mem x (mc-mem sn) k)
                                     (read-mem x (mc-mem s) k)))
                     (equal (iread-dn 32 0 sn) (max-fn* a b))))))
     ((use (max-s-s0)
           (max-s0-s1 (s (stepn s (max-t0 a b)))))
      (disable max-sp max-s0p max-s1p iread-dn linked-rts-addr
               stepn-rewriter)))

(prove-lemma disjoint-la4 (rewrite)
     (implies (and (disjoint a m b n)
                   (leq (plus j (index-n a1 a)) m)
                   (leq (plus l (index-n b1 b)) n))
              (disjoint a1 j b1 l)))
```

```
     ((use (disjoint-la3 (i (index-n a1 a)) (k (index-n b1 b))))
      (enable index-n)))

; instantiation.
(defn max-gt-statep (s a b)
  (let ((comp (read-mem (add 32 (read-sp s) 12) (mc-mem s) 4)))
    (and (equal (mc-status s) 'running)
         (evenp (mc-pc s))
         (rom-addrp (mc-pc s) (mc-mem s) 46)
         (mcode-addrp (mc-pc s) (mc-mem s) (max-code))
         (equal a (iread-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
         (equal b (iread-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
         (evenp comp)
         (rom-addrp comp (mc-mem s) (len (gt-code)))
         (mcode-addrp comp (mc-mem s) (gt-code))
         (ram-addrp (sub 32 28 (read-sp s)) (mc-mem s) 44))))

(defn max-gt-t (a b)
  (splus (max-t0 a b)
         (splus (gt-t a b)
                (if (negativep (gt a b)) 7 6))))

(defn max-gt (a b)
   (if (negativep (gt a b)) b a))

(prove-lemma max-gt-statep-s0 (rewrite)
     (let ((s0 (stepn s (max-t0 a b))))
       (implies (max-gt-statep s a b)
                (and (equal (mc-status s0) 'running)
                     (equal (mc-pc s0)
                            (read-mem (add 32 (read-sp s) 12) (mc-mem s) 4))
                     (equal (read-rn 32 15 (mc-rfile s0))
                            (sub 32 24 (read-sp s)))
                     (equal (iread-mem (add 32 (read-rn 32 15 (mc-rfile s0)) 4)
                                       (mc-mem s0) 4)
                            a)
                     (equal (iread-mem (add 32 (read-rn 32 15 (mc-rfile s0)) 8)
                                       (mc-mem s0) 4)
                            b))))
     ((use (max-s-s0)))))

(functionally-instantiate max-gt-correctness (rewrite)
     (implies (max-gt-statep s a b)
              (let ((sn (stepn s (max-gt-t a b))))
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32  (read-sp s) 4))
                     (implies (and (leq oplen 32)
                                   (d2-7a2-5p rn))
                              (equal (read-rn oplen rn (mc-rfile sn))
                                     (read-rn oplen rn (mc-rfile s))))
```

```
                (implies (and (disjoint x k (sub 32 24 (read-sp s)) 40)
                              (disjoint x k (sub 32 28 (read-sp s)) 4))
                         (equal (read-mem x (mc-mem sn) k)
                                (read-mem x (mc-mem s) k)))
                (equal (iread-dn 32 0 sn) (max-gt a b)))))
    max-correctness
    ((max-statep max-gt-statep)
     (max-disjoint (lambda (x k s) (disjoint x k (sub 32 28 (read-sp s)) 4)))
     (p-disjoint (lambda (x k s) (disjoint x k (sub 32 4 (read-sp s)) 4)))
     (comp-loadp
      (lambda (s a b)
        (let ((comp (read-mem (add 32 (read-sp s) 12) (mc-mem s) 4)))
          (and (evenp comp)
               (rom-addrp comp (mc-mem s) (len (gt-code)))
               (mcode-addrp comp (mc-mem s) (gt-code))
               (ram-addrp (sub 32 28 (read-sp s)) (mc-mem s) 44)))))
     (max-t max-gt-t)
     (max-fn* max-gt)
     (fn*-statep gt-statep)
     (fn*-t gt-t)
     (fn* gt))
    ((disable iread-mem)))
```

C.9 A Case Study of Embedded Assembly Code

```
;          Proof of the Correctness of a FOO Function
#|
Here is a trivial example to illustrate our ability to handle embedded
assembly code\index{embedded assembly code} in a high level language.

foo returns either a or b depending on the memory value at location 10000.

int foo (int a, int b)
{
  asm("tstl 10000:w ");
  asm("beq l1 ");
  asm("movl a6@(12), d0 ");
  asm("jmp end ");
  asm("l1: movl a6@(8), d0 ");
  asm("end: nop ");
}

The MC68020 assembly code of the above C function on SUN-3 is given as
follows.  This binary is generated by "gcc -O".

0x243a <foo>:           linkw fp,#0
0x243e <foo+4>:         tstl @#0x2710
0x2442 <foo+8>:         beq 0x244e <foo+20>
0x2446 <foo+12>:        movel fp@(12),d0
0x244a <foo+16>:        jmp 0x2452 <foo+24>
0x244e <foo+20>:        movel fp@(8),d0
0x2452 <foo+24>:        nop
```

```
0x2454 <foo+26>:        unlk fp
0x2456 <foo+28>:        rts

The machine code of the above program is:

<foo>:      0x4e56  0x0000  0x4ab8  0x2710  0x6700  0x000a  0x202e  0x000c
<foo+16>:   0x4efa  0x0006  0x202e  0x0008  0x4e71  0x4e5e  0x4e75

'(78      86      0       0       74      184     39      16
  103     0       0       10      32      46      0       12
  78      250     0       6       32      46      0       8
  78      113     78      94      78      117)
|#

; in the logic, the above program is defined by (foo-code).
(defn foo-code ()
  '(78      86      0       0       74      184     39      16
    103     0       0       10      32      46      0       12
    78      250     0       6       32      46      0       8
    78      113     78      94      78      117))

; the Nqthm counterpart of foo.
(defn foo (a b x) (if (equal x 0) a b))

; the computation time of the program.
(defn foo-t (x) (if (equal x 0) 7 8))

; the preconditions of the initial state.
(defn foo-statep (s a b)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (mc-pc s) (mc-mem s) 30)
       (mcode-addrp (mc-pc s) (mc-mem s) (foo-code))
       (ram-addrp (sub 32 4 (read-sp s)) (mc-mem s) 16)
       (ram-addrp 10000 (mc-mem s) 4)
       (disjoint 10000 4 (sub 32 4 (read-sp s)) 16)
       (equal a (iread-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal b (iread-mem (add 32 (read-sp s) 8) (mc-mem s) 4))))

; from the initial state to exit: s --> exit.
(prove-lemma foo-correctness (rewrite)
     (let ((x (iread-mem 10000 (mc-mem s) 4)))
       (implies (foo-statep s a b)
                (and (equal (mc-status (stepn s (foo-t x))) 'running)
                     (equal (mc-pc (stepn s (foo-t x))) (rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile (stepn s (foo-t x))))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (read-rn 32 15 (mc-rfile (stepn s (foo-t x))))
                            (add 32 (read-an 32 7 s) 4))
                     (implies
                      (d2-7a2-5p rn)
                      (equal (read-rn oplen rn (mc-rfile (stepn s (foo-t x))))
                             (read-rn oplen rn (mc-rfile s))))
                     (implies
```

```
                    (disjoint x k (sub 32 4 (read-sp s)) 16)
                    (equal (read-mem x (mc-mem (stepn s (foo-t x))) k)
                           (read-mem x (mc-mem s) k)))
                   (equal (iread-dn 32 0 (stepn s (foo-t x)))
                          (foo a b x))))))
```

C.10 The memchr Function

```
;          Proof of the Correctness of the MEMCHR Function
#|
This is part of our effort to verify the Berkeley string library.  The
Berkeley string library is widely used as part of the Berkeley Unix OS.

This is the source code of memchr function in the Berkeley string library.

/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 */
void *
memchr(s, c, n)
        const void *s;
        register unsigned char c;
        register size_t n;
{
        if (n != 0) {
                register const unsigned char *p = s;

                do {
                        if (*p++ == c)
                                return ((void *)(p - 1));
                } while (--n != 0);
        }
        return (NULL);
}

The MC68020 assembly code of the C function memchr on SUN-3 is given as
follows.  This binary is generated by "gcc -O".

0x27e8 <memchr>:        linkw fp,#0
0x27ec <memchr+4>:      moveb fp@(15),d1
0x27f0 <memchr+8>:      movel fp@(16),d0
0x27f4 <memchr+12>:     beq 0x2808 <memchr+32>
0x27f6 <memchr+14>:     moveal fp@(8),a0
0x27fa <memchr+18>:     cmpb a0@+,d1
0x27fc <memchr+20>:     bne 0x2804 <memchr+28>
0x27fe <memchr+22>:     movel a0,d0
0x2800 <memchr+24>:     subl #1,d0
0x2802 <memchr+26>:     bra 0x280a <memchr+34>
0x2804 <memchr+28>:     subl #1,d0
0x2806 <memchr+30>:     bne 0x27fa <memchr+18>
0x2808 <memchr+32>:     clrl d0
```

```
0x280a <memchr+34>:     unlk fp
0x280c <memchr+36>:     rts

The machine code of the above program is:

<memchr>:        0x4e56  0x0000  0x122e  0x000f  0x202e  0x0010  0x6712  0x206e
<memchr+16>:     0x0008  0xb218  0x6606  0x2008  0x5380  0x6006  0x5380  0x66f2
<memchr+32>:     0x4280  0x4e5e  0x4e75

'(78      86      0       0       18      46      0       15
  32      46      0       16      103     18      32      110
  0       8       178     24      102     6       32      8
  83      128     96      6       83      128     102     242
  66      128     78      94      78      117)
|#

; in the logic, the above program is defined by (memchr-code).
(defn memchr-code ()
  '(78      86      0       0       18      46      0       15
    32      46      0       16      103     18      32      110
    0       8       178     24      102     6       32      8
    83      128     96      6       83      128     102     242
    66      128     78      94      78      117))

; the computation time of the program.
(defn memchr-t1 (i n lst ch)
  (if (equal (get-nth i lst) ch)
      7
    (if (equal (sub1 n) 0)
        7
      (splus 4 (memchr-t1 (add1 i) (sub1 n) lst ch)))))

(defn memchr-t (n lst ch)
  (if (equal n 0)
      7
    (splus 5 (memchr-t1 0 n lst ch))))

; an induction hint.
(defn memchr-induct (s i* i n lst ch)
  (if (equal (get-nth i lst) ch)
      t
    (if (equal (sub1 n) 0)
        t
      (memchr-induct (stepn s 4) (add 32 i* 1) (add1 i) (sub1 n) lst ch))))

; the preconditions of the initial state.
(defn memchr-statep (s str n lst ch)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (mc-pc s) (mc-mem s) 38)
       (mcode-addrp (mc-pc s) (mc-mem s) (memchr-code))
       (ram-addrp (sub 32 4 (read-sp s)) (mc-mem s) 20)
       (ram-addrp str (mc-mem s) n)
       (mem-lst 1 str (mc-mem s) n lst)
```

```
       (disjoint (sub 32 4 (read-sp s)) 20 str n)
       (equal str (read-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal ch (uread-mem (add 32 (read-sp s) 11) (mc-mem s) 1))
       (equal n (uread-mem (add 32 (read-sp s) 12) (mc-mem s) 4))
       (not (equal (nat-to-uint str) 0))
       (uint-rangep (plus (nat-to-uint str) n) 32)))

; an intermediate state.
(defn memchr-s0p (s i* i str n lst ch n_)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 18 (mc-pc s)) (mc-mem s) 38)
       (mcode-addrp (sub 32 18 (mc-pc s)) (mc-mem s) (memchr-code))
       (ram-addrp (read-an 32 6 s) (mc-mem s) 20)
       (ram-addrp str (mc-mem s) n_)
       (mem-lst 1 str (mc-mem s) n_ lst)
       (disjoint (read-an 32 6 s) 20 str n_)
       (equal* (read-an 32 0 s) (add 32 str i*))
       (equal ch (nat-to-uint (read-dn 8 1 s)))
       (equal n (nat-to-uint (read-dn 32 0 s)))
       (equal i (nat-to-uint i*))
       (leq (plus i n) n_)
       (not (equal n 0))
       (numberp i*)
       (numberp n_)
       (nat-rangep i* 32)
       (uint-rangep n_ 32)))

; from the intial state s to exit: s --> sn.
(prove-lemma memchr-s-sn (rewrite)
     (implies (and (memchr-statep s str n lst ch)
                   (equal n 0))
              (and (equal (mc-status (stepn s 7)) 'running)
                   (equal (mc-pc (stepn s 7)) (rts-addr s))
                   (equal (read-dn 32 0 (stepn s 7)) 0)
                   (equal (read-rn 32 15 (mc-rfile (stepn s 7)))
                          (add 32 (read-an 32 7 s) 4))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 7)))
                          (read-an 32 6 s)))))

(prove-lemma memchr-s-sn-rfile (rewrite)
     (implies (and (memchr-statep s str n lst ch)
                   (equal n 0)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 7)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memchr-s-sn-mem (rewrite)
     (implies (and (memchr-statep s str n lst ch)
                   (equal n 0)
                   (disjoint x k (sub 32 4 (read-sp s)) 20))
              (equal (read-mem x (mc-mem (stepn s 7)) k)
                     (read-mem x (mc-mem s) k))))
```

```
; from the initial state s to s0: s --> s0.
(prove-lemma memchr-s-s0 ()
     (implies (and (memchr-statep s str n lst ch)
                   (not (equal n 0)))
              (memchr-s0p (stepn s 5) 0 0 str n lst ch n)))

(prove-lemma memchr-s-s0-else (rewrite)
     (implies (and (memchr-statep s str n lst ch)
                   (not (equal n 0)))
              (and (equal (linked-rts-addr (stepn s 5)) (rts-addr s))
                   (equal (linked-a6 (stepn s 5)) (read-an 32 6 s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 5)))
                          (sub 32 4 (read-sp s))))))

(prove-lemma memchr-s-s0-rfile (rewrite)
     (implies (and (memchr-statep s str n lst ch)
                   (not (equal n 0))
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 5)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memchr-s-s0-mem (rewrite)
     (implies (and (memchr-statep s str n lst ch)
                   (not (equal n 0))
                   (disjoint x k (sub 32 4 (read-sp s)) 20))
              (equal (read-mem x (mc-mem (stepn s 5)) k)
                     (read-mem x (mc-mem s) k))))

; from s0 to exit: s0 --> sn.
; base case 1: s0 --> sn, when lst[i] = ch.
(prove-lemma memchr-s0-sn-base1 (rewrite)
     (implies (and (memchr-s0p s i* i str n lst ch n_)
                   (equal (get-nth i lst) ch))
              (and (equal (mc-status (stepn s 7)) 'running)
                   (equal (mc-pc (stepn s 7)) (linked-rts-addr s))
                   (equal (read-dn 32 0 (stepn s 7)) (add 32 str i*))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 7)))
                          (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 7)))
                          (add 32 (read-an 32 6 s) 8))
                   (equal (read-mem x (mc-mem (stepn s 7)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma memchr-s0-sn-rfile-base1 (rewrite)
     (implies (and (memchr-s0p s i* i str n lst ch n_)
                   (equal (get-nth i lst) ch)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 7)))
                     (read-rn oplen rn (mc-rfile s)))))

; base case 2: s0 --> sn, when lst[i] =- ch, n-1 = 0.
(prove-lemma memchr-s0-sn-base2 (rewrite)
     (implies (and (memchr-s0p s i* i str n lst ch n_)
                   (not (equal (get-nth i lst) ch))
```

```
                (equal (sub1 n) 0))
           (and (equal (mc-status (stepn s 7)) 'running)
                (equal (mc-pc (stepn s 7)) (linked-rts-addr s))
                (equal (read-dn 32 0 (stepn s 7)) 0)
                (equal (read-rn 32 14 (mc-rfile (stepn s 7)))
                       (linked-a6 s))
                (equal (read-rn 32 15 (mc-rfile (stepn s 7)))
                       (add 32 (read-an 32 6 s) 8))
                (equal (read-mem x (mc-mem (stepn s 7)) k)
                       (read-mem x (mc-mem s) k)))))

(prove-lemma memchr-s0-sn-rfile-base2 (rewrite)
     (implies (and (memchr-s0p s i* i str n lst ch n_)
                   (not (equal (get-nth i lst) ch))
                   (equal (sub1 n) 0)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 7)))
                     (read-rn oplen rn (mc-rfile s)))))

; induction case: s0 --> s0.
(prove-lemma memchr-s0-s0 (rewrite)
     (implies (and (memchr-s0p s i* i str n lst ch n_)
                   (not (equal (get-nth i lst) ch))
                   (not (equal (sub1 n) 0)))
              (and (memchr-s0p (stepn s 4) (add 32 i* 1) (add1 i) str (sub1 n)
                               lst ch n_)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 4)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (linked-a6 (stepn s 4)) (linked-a6 s))
                   (equal (linked-rts-addr (stepn s 4))
                          (linked-rts-addr s))
                   (equal (read-mem x (mc-mem (stepn s 4)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma memchr-s0-s0-rfile (rewrite)
     (implies (and (memchr-s0p s i* i str n lst ch n_)
                   (not (equal (get-nth i lst) ch))
                   (not (equal (sub1 n) 0))
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 4)))
                     (read-rn oplen rn (mc-rfile s)))))

; put together (s0 --> exit).
(prove-lemma memchr-s0-sn (rewrite)
     (let ((sn (stepn s (memchr-t1 i n lst ch))))
       (implies (memchr-s0p s i* i str n lst ch n_)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn)
                            (if (memchr1 i n lst ch)
                                (add 32 str (memchr* i* i n lst ch))
                              0))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (linked-a6 s))
```

```
                    (equal (read-rn 32 15 (mc-rfile sn))
                           (add 32 (read-an 32 6 s) 8))
                    (equal (read-mem x (mc-mem sn) k)
                           (read-mem x (mc-mem s) k)))))
     ((induct (memchr-induct s i* i n lst ch))
      (disable memchr-s0p read-dn)))

(prove-lemma memchr-s0-sn-rfile (rewrite)
     (implies
      (and (memchr-s0p s i* i str n lst ch n_)
           (d2-7a2-5p rn))
      (equal (read-rn oplen rn (mc-rfile (stepn s (memchr-t1 i n lst ch))))
             (read-rn oplen rn (mc-rfile s))))
     ((induct (memchr-induct s i* i n lst ch))
      (disable memchr-s0p)))

; the correctness of the MEMCHR program.
(prove-lemma memchr-correctness (rewrite)
     (let ((sn (stepn s (memchr-t n lst ch))))
       (implies (memchr-statep s str n lst ch)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-sp s) 4))
                     (implies (d2-7a2-5p rn)
                              (equal (read-rn oplen rn (mc-rfile sn))
                                     (read-rn oplen rn (mc-rfile s))))
                     (implies (disjoint x k (sub 32 4 (read-sp s)) 20)
                              (equal (read-mem x (mc-mem sn) k)
                                     (read-mem x (mc-mem s) k)))
                     (equal (read-dn 32 0 sn)
                            (if (memchr n lst ch)
                                (add 32 str (memchr* 0 0 n lst ch))
                              0)))))
     ((use (memchr-s-s0))
      (disable memchr-statep memchr-s0p linked-rts-addr linked-a6 read-dn)))

(disable memchr-t)

; memchr* --> memchr.
(prove-lemma memchr*-memchr1 (rewrite)
     (implies (and (memchr1 i n lst ch)
                   (equal i (nat-to-uint i*))
                   (nat-rangep i* 32)
                   (uint-rangep (plus i n) 32))
              (equal (nat-to-uint (memchr* i* i n lst ch))
                     (memchr1 i n lst ch)))
     ((induct (memchr* i* i n lst ch))))

(prove-lemma memchr-non-zerop-la ()
     (let ((sn (stepn s (memchr-t n lst ch))))
       (implies (and (memchr-statep s str n lst ch)
```

```
                    (numberp n)
                    (nat-rangep str 32)
                    (not (equal (nat-to-uint str) 0))
                    (uint-rangep (plus (nat-to-uint str) n) 32)
                    (memchr n lst ch))
               (not (equal (nat-to-uint (read-dn 32 0 sn)) 0))))
     ((enable nat-rangep-la)
      (disable memchr-statep memchr-t read-dn)))

(prove-lemma memchr-non-zerop (rewrite)
     (let ((sn (stepn s (memchr-t n lst ch))))
       (implies (and (memchr-statep s str n lst ch)
                     (memchr n lst ch))
                (not (equal (nat-to-uint (read-dn 32 0 sn)) 0))))
     ((use (memchr-non-zerop-la))))

(disable memchr*)

; some properties of memchr.
; see file cstring.events.
```

C.11 The memcmp Function

```
;          Proof of the Correctness of the MEMCMP Function
#|
This is part of our effort to verify the Berkeley string library.  The
Berkeley string library is widely used as part of the Berkeley Unix OS.

This is the source code of memcmp function in the Berkeley string library.

/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 */
int
memcmp(s1, s2, n)
        const void *s1, *s2;
        size_t n;
{
        if (n != 0) {
                register const unsigned char *p1 = s1, *p2 = s2;

                do {
                        if (*p1++ != *p2++)
                                return (*--p1 - *--p2);
                } while (--n != 0);
        }
        return (0);
}

The MC68020 assembly code of the C function memcmp on SUN-3 is given as
follows.  This binary is generated by "gcc -O".
```

```
0x2810 <memcmp>:        linkw fp,#0
0x2814 <memcmp+4>:      movel d2,sp@-
0x2816 <memcmp+6>:      movel fp@(16),d0
0x281a <memcmp+10>:     beq 0x2842 <memcmp+50>
0x281c <memcmp+12>:     moveal fp@(8),a1
0x2820 <memcmp+16>:     moveal fp@(12),a0
0x2824 <memcmp+20>:     andil #255,d1
0x282a <memcmp+26>:     andil #255,d2
0x2830 <memcmp+32>:     cmpmb a0@+,a1@+
0x2832 <memcmp+34>:     beq 0x283e <memcmp+46>
0x2834 <memcmp+36>:     moveb a1@-,d1
0x2836 <memcmp+38>:     moveb a0@-,d2
0x2838 <memcmp+40>:     movel d1,d0
0x283a <memcmp+42>:     subl d2,d0
0x283c <memcmp+44>:     bra 0x2844 <memcmp+52>
0x283e <memcmp+46>:     subl #1,d0
0x2840 <memcmp+48>:     bne 0x2830 <memcmp+32>
0x2842 <memcmp+50>:     clrl d0
0x2844 <memcmp+52>:     movel fp@(-4),d2
0x2848 <memcmp+56>:     unlk fp
0x284a <memcmp+58>:     rts

The machine code of the above program is:

<memcmp>:      0x4e56  0x0000  0x2f02  0x202e  0x0010  0x6726  0x226e  0x0008
<memcmp+16>:   0x206e  0x000c  0x0281  0x0000  0x00ff  0x0282  0x0000  0x00ff
<memcmp+32>:   0xb308  0x670a  0x1221  0x1420  0x2001  0x9082  0x6006  0x5380
<memcmp+48>:   0x66ee  0x4280  0x242e  0xfffc  0x4e5e  0x4e75

'(78      86      0       0       47      2       32      46
  0       16      103     38      34      110     0       8
  32      110     0       12      2       129     0       0
  0       255     2       130     0       0       0       255
  179     8       103     10      18      33      20      32
  32      1       144     130     96      6       83      128
  102     238     66      128     36      46      255     252
  78      94      78      117)
|#

; in the logic, the above program is defined by (memcmp-code).
(defn memcmp-code ()
  '(78      86      0       0       47      2       32      46
    0       16      103     38      34      110     0       8
    32      110     0       12      2       129     0       0
    0       255     2       130     0       0       0       255
    179     8       103     10      18      33      20      32
    32      1       144     130     96      6       83      128
    102     238     66      128     36      46      255     252
    78      94      78      117)  )

; the computation time of the program.
(defn memcmp-t1 (i n lst1 lst2)
  (if (equal (get-nth i lst1) (get-nth i lst2))
```

```
      (if (equal (sub1 n) 0)
          8
        (splus 4 (memcmp-t1 (add1 i) (sub1 n) lst1 lst2)))
    10))

(defn memcmp-t (n lst1 lst2)
  (if (equal n 0)
      8
    (splus 8 (memcmp-t1 0 n lst1 lst2))))

; an induction hint.
(defn memcmp-induct (s i* i n lst1 lst2)
  (if (equal (get-nth i lst1) (get-nth i lst2))
      (if (equal (sub1 n) 0)
          t
        (memcmp-induct (stepn s 4) (add 32 i* 1)  (add1 i) (sub1 n)
                       lst1 lst2))
    t))

; the preconditions of the initial state.
(defn memcmp-statep (s str1 n lst1 str2 lst2)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (mc-pc s) (mc-mem s) 60)
       (mcode-addrp (mc-pc s) (mc-mem s) (memcmp-code))
       (ram-addrp (sub 32 8 (read-sp s)) (mc-mem s) 24)
       (ram-addrp str1 (mc-mem s) n)
       (mem-lst 1 str1 (mc-mem s) n lst1)
       (ram-addrp str2 (mc-mem s) n)
       (mem-lst 1 str2 (mc-mem s) n lst2)
       (disjoint (sub 32 8 (read-sp s)) 24 str1 n)
       (disjoint (sub 32 8 (read-sp s)) 24 str2 n)
       (equal str1 (read-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal str2 (read-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
       (equal n (uread-mem (add 32 (read-sp s) 12) (mc-mem s) 4))))

; an intermediate state.
(defn memcmp-s0p (s i* i str1 n lst1 str2 lst2 n_)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 32 (mc-pc s)) (mc-mem s) 60)
       (mcode-addrp (sub 32 32 (mc-pc s)) (mc-mem s) (memcmp-code))
       (ram-addrp (sub 32 4 (read-an 32 6 s)) (mc-mem s) 24)
       (ram-addrp str1 (mc-mem s) n_)
       (mem-lst 1 str1 (mc-mem s) n_ lst1)
       (ram-addrp str2 (mc-mem s) n_)
       (mem-lst 1 str2 (mc-mem s) n_ lst2)
       (disjoint (sub 32 4 (read-an 32 6 s)) 24 str1 n_)
       (disjoint (sub 32 4 (read-an 32 6 s)) 24 str2 n_)
       (equal* (read-an 32 1 s) (add 32 str1 i*))
       (equal* (read-an 32 0 s) (add 32 str2 i*))
       (nat-rangep (read-rn 32 1 (mc-rfile s)) 8)
       (nat-rangep (read-rn 32 2 (mc-rfile s)) 8)
       (equal n (nat-to-uint (read-dn 32 0 s)))
```

```
       (equal i (nat-to-uint i*))
       (leq (plus i n) n_)
       (not (equal n 0))
       (numberp i*)
       (nat-rangep i* 32)
       (uint-rangep n_ 32)))

; from the initial state s to exit: s --> sn, when n = 0.
(prove-lemma memcmp-s-sn (rewrite)
     (implies (and (memcmp-statep s str1 n lst1 str2 lst2)
                   (equal n 0))
              (and (equal (mc-status (stepn s 8)) 'running)
                   (equal (mc-pc (stepn s 8)) (rts-addr s))
                   (equal (iread-dn 32 0 (stepn s 8)) 0)
                   (equal (read-rn 32 15 (mc-rfile (stepn s 8)))
                          (add 32 (read-an 32 7 s) 4))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 8)))
                          (read-an 32 6 s)))))

(prove-lemma memcmp-s-sn-rfile (rewrite)
     (implies (and (memcmp-statep s str1 n lst1 str2 lst2)
                   (equal n 0)
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 8)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memcmp-s-sn-mem (rewrite)
     (implies (and (memcmp-statep s str1 n lst1 str2 lst2)
                   (equal n 0)
                   (disjoint x k (sub 32 8 (read-sp s)) 24))
              (equal (read-mem x (mc-mem (stepn s 8)) k)
                     (read-mem x (mc-mem s) k))))

; from the initial state s to s0: s --> s0.
(prove-lemma memcmp-s-s0 ()
     (implies (and (memcmp-statep s str1 n lst1 str2 lst2)
                   (not (equal n 0)))
              (memcmp-s0p (stepn s 8) 0 0 str1 n lst1 str2 lst2 n)))

(prove-lemma memcmp-s-s0-else (rewrite)
     (implies (and (memcmp-statep s str1 n lst1 str2 lst2)
                   (not (equal n 0)))
              (and (equal (linked-rts-addr (stepn s 8)) (rts-addr s))
                   (equal (linked-a6 (stepn s 8)) (read-an 32 6 s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 8)))
                          (sub 32 4 (read-sp s)))
                   (equal (rn-saved (stepn s 8)) (read-dn 32 2 s)))))

(prove-lemma memcmp-s-s0-rfile (rewrite)
     (implies (and (memcmp-statep s str1 n lst1 str2 lst2)
                   (not (equal n 0))
                   (d3-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 8)))
```

```
                    (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memcmp-s-s0-mem (rewrite)
     (implies (and (memcmp-statep s str1 n lst1 str2 lst2)
                   (not (equal n 0))
                   (disjoint x k (sub 32 8 (read-sp s)) 24))
              (equal (read-mem x (mc-mem (stepn s 8)) k)
                     (read-mem x (mc-mem s) k))))

; from s0 to exit: s0 --> sn.
; base case 1:  s0 --> sn, when lst1[i] =- lst2[i].
(prove-lemma memcmp-s0-sn-base1 (rewrite)
     (implies (and (memcmp-s0p s i* i str1 n lst1 str2 lst2 n_)
                   (not (equal (get-nth i lst1) (get-nth i lst2))))
              (and (equal (mc-status (stepn s 10)) 'running)
                   (equal (mc-pc (stepn s 10)) (linked-rts-addr s))
                   (equal (iread-dn 32 0 (stepn s 10))
                          (idifference (get-nth i lst1) (get-nth i lst2)))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 10)))
                          (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 10)))
                          (add 32 (read-an 32 6 s) 8))
                   (equal (read-mem x (mc-mem (stepn s 10)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma memcmp-s0-sn-rfile-base1 (rewrite)
     (implies (and (memcmp-s0p s i* i str1 n lst1 str2 lst2 n_)
                   (not (equal (get-nth i lst1) (get-nth i lst2)))
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 10)))
                     (if (d3-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (head (rn-saved s) oplen)))))

; base case 2: s0 --> sn, when lst[i] = lst2[i], lst[i] =- 0, and n-1 = 0.
(prove-lemma memcmp-s0-sn-base2 (rewrite)
     (implies (and (memcmp-s0p s i* i str1 n lst1 str2 lst2 n_)
                   (equal (get-nth i lst1) (get-nth i lst2))
                   (equal (sub1 n) 0))
              (and (equal (mc-status (stepn s 8)) 'running)
                   (equal (mc-pc (stepn s 8)) (linked-rts-addr s))
                   (equal (iread-dn 32 0 (stepn s 8)) 0)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 8)))
                          (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 8)))
                          (add 32 (read-an 32 6 s) 8))
                   (equal (read-mem x (mc-mem (stepn s 8)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma memcmp-s0-sn-rfile-base2 (rewrite)
     (implies (and (memcmp-s0p s i* i str1 n lst1 str2 lst2 n_)
                   (equal (get-nth i lst1) (get-nth i lst2))
                   (equal (sub1 n) 0)
```

```
                (leq oplen 32)
                (d2-7a2-5p rn))
           (equal (read-rn oplen rn (mc-rfile (stepn s 8)))
                  (if (d3-7a2-5p rn)
                      (read-rn oplen rn (mc-rfile s))
                    (head (rn-saved s) oplen)))))

; induction case: s0 --> s0, lst[i] = lst2[i], lst[i] =- 0 and n-1 =- 0.
(prove-lemma memcmp-s0-s0 (rewrite)
     (implies (and (memcmp-s0p s i* i str1 n lst1 str2 lst2 n_)
                   (equal (get-nth i lst1) (get-nth i lst2))
                   (not (equal (sub1 n) 0)))
              (and (memcmp-s0p (stepn s 4) (add 32 i* 1) (add1 i)
                               str1 (sub1 n) lst1 str2 lst2 n_)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 4)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (linked-a6 (stepn s 4)) (linked-a6 s))
                   (equal (linked-rts-addr (stepn s 4)) (linked-rts-addr s))
                   (equal (rn-saved (stepn s 4)) (rn-saved s))
                   (equal (read-mem x (mc-mem (stepn s 4)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma memcmp-s0-s0-rfile (rewrite)
     (implies (and (memcmp-s0p s i* i str1 n lst1 str2 lst2 n_)
                   (equal (get-nth i lst1) (get-nth i lst2))
                   (not (equal (sub1 n) 0))
                   (d3-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 4)))
                     (read-rn oplen rn (mc-rfile s)))))

; put together. s0 --> exit.
(prove-lemma memcmp-s0-sn (rewrite)
     (let ((sn (stepn s (memcmp-t1 i n lst1 lst2))))
       (implies (memcmp-s0p s i* i str1 n lst1 str2 lst2 n_)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (iread-dn 32 0 sn) (memcmp1 i n lst1 lst2))
                     (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))
                     (equal (read-mem x (mc-mem sn) k)
                            (read-mem x (mc-mem s) k)))))
     ((induct (memcmp-induct s i* i n lst1 lst2))
      (disable memcmp-s0p iread-dn)))

(prove-lemma memcmp-s0-sn-rfile (rewrite)
     (implies
      (and (memcmp-s0p s i* i str1 n lst1 str2 lst2 n_)
           (d2-7a2-5p rn)
           (leq oplen 32))
      (equal (read-rn oplen rn (mc-rfile (stepn s (memcmp-t1 i n lst1 lst2))))
             (if (d3-7a2-5p rn)
                 (read-rn oplen rn (mc-rfile s))
               (head (rn-saved s) oplen)))))
```

```
     ((induct (memcmp-induct s i* i n lst1 lst2))
      (disable memcmp-s0p)))

; the correctness of memcmp.
(prove-lemma memcmp-correctness (rewrite)
     (let ((sn (stepn s (memcmp-t n lst1 lst2))))
       (implies (memcmp-statep s str1 n lst1 str2 lst2)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 7 s) 4))
                     (implies (and (d2-7a2-5p rn)
                                   (leq oplen 32))
                              (equal (read-rn oplen rn (mc-rfile sn))
                                     (read-rn oplen rn (mc-rfile s))))
                     (implies (disjoint x k (sub 32 8 (read-sp s)) 24)
                              (equal (read-mem x (mc-mem sn) k)
                                     (read-mem x (mc-mem s) k)))
                     (equal (iread-dn 32 0 sn) (memcmp n lst1 lst2))))))
     ((use (memcmp-s-s0))
      (disable memcmp-statep memcmp-s0p stepn-rewriter linked-rts-addr
               linked-a6 iread-dn)))

(disable memcmp-t)

; some properties of memcmp.
; see file cstring.events.
```

C.12 The memcpy Function

```
;          Proof of the Correctness of the MEMCPY Function
#|
This is part of our effort to verify the Berkeley string library.  The
Berkeley string library is widely used as part of the Berkeley Unix OS.

This is the source code of memcpy function in the Berkeley string library.

/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 */
typedef int word;               /* "word" used for optimal copy speed */

#define wsize   sizeof(word)
#define wmask   (wsize - 1)

/*
 * Copy a block of memory, handling overlap.
 * This is the routine that actually implements
 * (the portable versions of) bcopy, memcpy, and memmove.
```

```
 */
void *
memcpy(dst0, src0, length)
        void *dst0;
        const void *src0;
        register size_t length;
{
        register char *dst = dst0;
        register const char *src = src0;
        register size_t t;

        if (length == 0 || dst == src)          /* nothing to do */
                goto done;

        /*
         * Macros: loop-t-times; and loop-t-times, t>0
         */
#define TLOOP(s) if (t) TLOOP1(s)
#define TLOOP1(s) do { s; } while (--t)

        if ((unsigned long)dst < (unsigned long)src) {
                /*
                 * Copy forward.
                 */
                t = (int)src;   /* only need low bits */
                if ((t | (int)dst) & wmask) {
                        /*
                         * Try to align operands.  This cannot be done
                         * unless the low bits match.
                         */
                        if ((t ^ (int)dst) & wmask || length < wsize)
                                t = length;
                        else
                                t = wsize - (t & wmask);
                        length -= t;
                        TLOOP1(*dst++ = *src++);
                }
                /*
                 * Copy whole words, then mop up any trailing bytes.
                 */
                t = length / wsize;
                TLOOP(*(word *)dst = *(word *)src; src += wsize; dst += wsize);
                t = length & wmask;
                TLOOP(*dst++ = *src++);
        } else {
                /*
                 * Copy backwards.  Otherwise essentially the same.
                 * Alignment works as before, except that it takes
                 * (t&wmask) bytes to align, not wsize-(t&wmask).
                 */
                src += length;
                dst += length;
                t = (int)src;
                if ((t | (int)dst) & wmask) {
```

```
                        if ((t ^ (int)dst) & wmask || length <= wsize)
                                t = length;
                        else
                                t &= wmask;
                        length -= t;
                        TLOOP1(*--dst = *--src);
                }
                t = length / wsize;
                TLOOP(src -= wsize; dst -= wsize; *(word *)dst = *(word *)src);
                t = length & wmask;
                TLOOP(*--dst = *--src);
        }
done:
        return (dst0);
}
```

The MC68020 assembly code of the C function memcpy on SUN-3 is given as follows. This binary is generated by "gcc -O".

```
0x2610 <memcpy>:        linkw fp,#0
0x2614 <memcpy+4>:      moveml d2-d4,sp@-
0x2618 <memcpy+8>:      movel fp@(8),d3
0x261c <memcpy+12>:     movel fp@(16),d2
0x2620 <memcpy+16>:     moveal d3,a1
0x2622 <memcpy+18>:     moveal fp@(12),a0
0x2626 <memcpy+22>:     beq 0x26c4 <memcpy+180>
0x262a <memcpy+26>:     cmpal d3,a0
0x262c <memcpy+28>:     beq 0x26c4 <memcpy+180>
0x2630 <memcpy+32>:     bls 0x267c <memcpy+108>
0x2632 <memcpy+34>:     movel a0,d1
0x2634 <memcpy+36>:     movel d1,d0
0x2636 <memcpy+38>:     orl d3,d0
0x2638 <memcpy+40>:     movel #3,d4
0x263a <memcpy+42>:     andl d4,d0
0x263c <memcpy+44>:     beq 0x2662 <memcpy+82>
0x263e <memcpy+46>:     movel d1,d0
0x2640 <memcpy+48>:     eorl d3,d0
0x2642 <memcpy+50>:     movel #3,d4
0x2644 <memcpy+52>:     andl d4,d0
0x2646 <memcpy+54>:     bne 0x264e <memcpy+62>
0x2648 <memcpy+56>:     movel #3,d4
0x264a <memcpy+58>:     cmpl d2,d4
0x264c <memcpy+60>:     bcs 0x2652 <memcpy+66>
0x264e <memcpy+62>:     movel d2,d1
0x2650 <memcpy+64>:     bra 0x265a <memcpy+74>
0x2652 <memcpy+66>:     movel #3,d0
0x2654 <memcpy+68>:     andl d1,d0
0x2656 <memcpy+70>:     movel #4,d1
0x2658 <memcpy+72>:     subl d0,d1
0x265a <memcpy+74>:     subl d1,d2
0x265c <memcpy+76>:     moveb a0@+,a1@+
0x265e <memcpy+78>:     subl #1,d1
0x2660 <memcpy+80>:     bne 0x265c <memcpy+76>
0x2662 <memcpy+82>:     movel d2,d1
```

```
0x2664 <memcpy+84>:     lsrl #2,d1
0x2666 <memcpy+86>:     beq 0x266e <memcpy+94>
0x2668 <memcpy+88>:     movel a0@+,a1@+
0x266a <memcpy+90>:     subl #1,d1
0x266c <memcpy+92>:     bne 0x2668 <memcpy+88>
0x266e <memcpy+94>:     movel #3,d1
0x2670 <memcpy+96>:     andl d2,d1
0x2672 <memcpy+98>:     beq 0x26c4 <memcpy+180>
0x2674 <memcpy+100>:    moveb a0@+,a1@+
0x2676 <memcpy+102>:    subl #1,d1
0x2678 <memcpy+104>:    bne 0x2674 <memcpy+100>
0x267a <memcpy+106>:    bra 0x26c4 <memcpy+180>
0x267c <memcpy+108>:    addal d2,a0
0x267e <memcpy+110>:    addal d2,a1
0x2680 <memcpy+112>:    movel a0,d1
0x2682 <memcpy+114>:    movel a1,d0
0x2684 <memcpy+116>:    orl d1,d0
0x2686 <memcpy+118>:    movel #3,d4
0x2688 <memcpy+120>:    andl d4,d0
0x268a <memcpy+122>:    beq 0x26ac <memcpy+156>
0x268c <memcpy+124>:    movel a1,d0
0x268e <memcpy+126>:    eorl d1,d0
0x2690 <memcpy+128>:    movel #3,d4
0x2692 <memcpy+130>:    andl d4,d0
0x2694 <memcpy+132>:    bne 0x269c <memcpy+140>
0x2696 <memcpy+134>:    movel #4,d4
0x2698 <memcpy+136>:    cmpl d2,d4
0x269a <memcpy+138>:    bcs 0x26a0 <memcpy+144>
0x269c <memcpy+140>:    movel d2,d1
0x269e <memcpy+142>:    bra 0x26a4 <memcpy+148>
0x26a0 <memcpy+144>:    movel #3,d4
0x26a2 <memcpy+146>:    andl d4,d1
0x26a4 <memcpy+148>:    subl d1,d2
0x26a6 <memcpy+150>:    moveb a0@-,a1@-
0x26a8 <memcpy+152>:    subl #1,d1
0x26aa <memcpy+154>:    bne 0x26a6 <memcpy+150>
0x26ac <memcpy+156>:    movel d2,d1
0x26ae <memcpy+158>:    lsrl #2,d1
0x26b0 <memcpy+160>:    beq 0x26b8 <memcpy+168>
0x26b2 <memcpy+162>:    movel a0@-,a1@-
0x26b4 <memcpy+164>:    subl #1,d1
0x26b6 <memcpy+166>:    bne 0x26b2 <memcpy+162>
0x26b8 <memcpy+168>:    movel #3,d1
0x26ba <memcpy+170>:    andl d2,d1
0x26bc <memcpy+172>:    beq 0x26c4 <memcpy+180>
0x26be <memcpy+174>:    moveb a0@-,a1@-
0x26c0 <memcpy+176>:    subl #1,d1
0x26c2 <memcpy+178>:    bne 0x26be <memcpy+174>
0x26c4 <memcpy+180>:    movel d3,d0
0x26c6 <memcpy+182>:    moveml fp@(-12),d2-d4
0x26cc <memcpy+188>:    unlk fp
0x26ce <memcpy+190>:    rts
```

The machine code of the above program is:

```
<memcpy>:        0x4e56  0x0000  0x48e7  0x3800  0x262e  0x0008  0x242e  0x0010
<memcpy+16>:     0x2243  0x206e  0x000c  0x6700  0x009c  0xb1c3  0x6700  0x0096
<memcpy+32>:     0x634a  0x2208  0x2001  0x8083  0x7803  0xc084  0x6724  0x2001
<memcpy+48>:     0xb780  0x7803  0xc084  0x6606  0x7803  0xb882  0x6504  0x2202
<memcpy+64>:     0x6008  0x7003  0xc081  0x7204  0x9280  0x9481  0x12d8  0x5381
<memcpy+80>:     0x66fa  0x2202  0xe489  0x6706  0x22d8  0x5381  0x66fa  0x7203
<memcpy+96>:     0xc282  0x6750  0x12d8  0x5381  0x66fa  0x6048  0xd1c2  0xd3c2
<memcpy+112>:    0x2208  0x2009  0x8081  0x7803  0xc084  0x6720  0x2009  0xb380
<memcpy+128>:    0x7803  0xc084  0x6606  0x7804  0xb882  0x6504  0x2202  0x6004
<memcpy+144>:    0x7803  0xc284  0x9481  0x1320  0x5381  0x66fa  0x2202  0xe489
<memcpy+160>:    0x6706  0x2320  0x5381  0x66fa  0x7203  0xc282  0x6706  0x1320
<memcpy+176>:    0x5381  0x66fa  0x2003  0x4cee  0x001c  0xfff4  0x4e5e  0x4e75

'(78      86      0       0       72      231     56      0
  38      46      0       8       36      46      0       16
  34      67      32      110     0       12      103     0
  0       156     177     195     103     0       0       150
  99      74      34      8       32      1       128     131
  120     3       192     132     103     36      32      1
  183     128     120     3       192     132     102     6
  120     3       184     130     101     4       34      2
  96      8       112     3       192     129     114     4
  146     128     148     129     18      216     83      129
  102     250     34      2       228     137     103     6
  34      216     83      129     102     250     114     3
  194     130     103     80      18      216     83      129
  102     250     96      72      209     194     211     194
  34      8       32      9       128     129     120     3
  192     132     103     32      32      9       179     128
  120     3       192     132     102     6       120     4
  184     130     101     4       34      2       96      4
  120     3       194     132     148     129     19      32
  83      129     102     250     34      2       228     137
  103     6       35      32      83      129     102     250
  114     3       194     130     103     6       19      32
  83      129     102     250     32      3       76      238
  0       28      255     244     78      94      78      117)
|#

; in the logic, the above program is defined by (memcpy-code).
(defn memcpy-code ()
  '(78      86      0       0       72      231     56      0
    38      46      0       8       36      46      0       16
    34      67      32      110     0       12      103     0
    0       156     177     195     103     0       0       150
    99      74      34      8       32      1       128     131
    120     3       192     132     103     36      32      1
    183     128     120     3       192     132     102     6
    120     3       184     130     101     4       34      2
    96      8       112     3       192     129     114     4
    146     128     148     129     18      216     83      129
    102     250     34      2       228     137     103     6
    34      216     83      129     102     250     114     3
```

```
    194     130     103     80      18      216     83      129
    102     250     96      72      209     194     211     194
    34      8       32      9       128     129     120     3
    192     132     103     32      32      9       179     128
    120     3       192     132     102     6       120     4
    184     130     101     4       34      2       96      4
    120     3       194     132     148     129     19      32
    83      129     102     250     34      2       228     137
    103     6       35      32      83      129     102     250
    114     3       194     130     103     6       19      32
    83      129     102     250     32      3       76      238
    0       28      255     244     78      94      78      117))

; the initial state.
(defn memcpy-statep (s str1 n lst1 str2 lst2)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (mc-pc s) (mc-mem s) 192)
       (mcode-addrp (mc-pc s) (mc-mem s) (memmove-code))
       (ram-addrp (sub 32 16 (read-sp s)) (mc-mem s) 32)
       (ram-addrp str1 (mc-mem s) n)
       (mem-lst 1 str1 (mc-mem s) n lst1)
       (ram-addrp str2 (mc-mem s) n)
       (mem-lst 1 str2 (mc-mem s) n lst2)
       (disjoint (sub 32 16 (read-sp s)) 32 str1 n)
       (disjoint (sub 32 16 (read-sp s)) 32 str2 n)
       (equal str1 (read-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal str2 (read-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
       (equal n (uread-mem (add 32 (read-sp s) 12) (mc-mem s) 4))
       (uint-rangep (plus (nat-to-uint str1) n) 32)
       (uint-rangep (plus (nat-to-uint str2) n) 32)))

; the time function.
(defn memcpy-t (str1 n lst1 str2 lst2)
  (memmove-t str1 n lst1 str2 lst2))

; the behavior.
(defn memcpy (str1 str2 n lst1 lst2)
  (memmove str1 str2 n lst1 lst2))

; memcpy and memmove are identical.
(prove-lemma memcpy-memmove-statep (rewrite)
     (equal (memcpy-statep s str1 n lst1 str2 lst2)
            (memmove-statep s str1 n lst1 str2 lst2)))

; the correctness.
(prove-lemma memcpy-correctness (rewrite)
     (let ((sn (stepn s (memcpy-t str1 n lst1 str2 lst2))))
       (implies (memcpy-statep s str1 n lst1 str2 lst2)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (read-rn 32 15 (mc-rfile sn))
```

```
                        (add 32 (read-sp s) 4))
                  (implies (and (d2-7a2-5p rn)
                                (leq oplen 32))
                           (equal (read-rn oplen rn (mc-rfile sn))
                                  (read-rn oplen rn (mc-rfile s))))
                  (implies (and (disjoint x k (sub 32 16 (read-sp s)) 32)
                                (disjoint x k str1 n))
                           (equal (read-mem x (mc-mem sn) k)
                                  (read-mem x (mc-mem s) k)))
                  (equal (read-dn 32 0 sn) str1)
                  (mem-lst 1 str1 (mc-mem sn) n
                           (memcpy str1 str2 n lst1 lst2)))))
    ((disable memmove-statep memcpy-statep read-dn memmove)))

(disable memcpy-t)

; some properties of memcpy.
; the same as memmove.
```

C.13 The memmove Function

```
;          Proof of the Correctness of the MEMMOVE Function
#|
This is part of our effort to verify the Berkeley string library.  The
Berkeley string library is widely used as part of the Berkeley Unix OS.

This is the source code of memmove function in the Berkeley string library.

/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 */
typedef int word;               /* "word" used for optimal copy speed */

#define wsize   sizeof(word)
#define wmask   (wsize - 1)

/*
 * Copy a block of memory, handling overlap.
 * This is the routine that actually implements
 * (the portable versions of) bcopy, memcpy, and memmove.
 */
void *
memmove(dst0, src0, length)
        void *dst0;
        const void *src0;
        register size_t length;
{
        register char *dst = dst0;
        register const char *src = src0;
        register size_t t;
```

```
        if (length == 0 || dst == src)          /* nothing to do */
                goto done;

        /*
         * Macros: loop-t-times; and loop-t-times, t>0
         */
#define TLOOP(s) if (t) TLOOP1(s)
#define TLOOP1(s) do { s; } while (--t)

        if ((unsigned long)dst < (unsigned long)src) {
                /*
                 * Copy forward.
                 */
                t = (int)src;   /* only need low bits */
                if ((t | (int)dst) & wmask) {
                        /*
                         * Try to align operands.  This cannot be done
                         * unless the low bits match.
                         */
                        if ((t ^ (int)dst) & wmask || length < wsize)
                                t = length;
                        else
                                t = wsize - (t & wmask);
                        length -= t;
                        TLOOP1(*dst++ = *src++);
                }
                /*
                 * Copy whole words, then mop up any trailing bytes.
                 */
                t = length / wsize;
                TLOOP(*(word *)dst = *(word *)src; src += wsize; dst += wsize);
                t = length & wmask;
                TLOOP(*dst++ = *src++);
        } else {
                /*
                 * Copy backwards.  Otherwise essentially the same.
                 * Alignment works as before, except that it takes
                 * (t&wmask) bytes to align, not wsize-(t&wmask).
                 */
                src += length;
                dst += length;
                t = (int)src;
                if ((t | (int)dst) & wmask) {
                        if ((t ^ (int)dst) & wmask || length <= wsize)
                                t = length;
                        else
                                t &= wmask;
                        length -= t;
                        TLOOP1(*--dst = *--src);
                }
                t = length / wsize;
                TLOOP(src -= wsize; dst -= wsize; *(word *)dst = *(word *)src);
                t = length & wmask;
                TLOOP(*--dst = *--src);
```

```
        }
done:
        return (dst0);
}
```

The MC68020 assembly code of the C function memmove on SUN-3 is given as follows. This binary is generated by "gcc -O".

```
0x2550 <memmove>:       linkw fp,#0
0x2554 <memmove+4>:     moveml d2-d4,sp@-
0x2558 <memmove+8>:     movel fp@(8),d3
0x255c <memmove+12>:    movel fp@(16),d2
0x2560 <memmove+16>:    moveal d3,a1
0x2562 <memmove+18>:    moveal fp@(12),a0
0x2566 <memmove+22>:    beq 0x2604 <memmove+180>
0x256a <memmove+26>:    cmpal d3,a0
0x256c <memmove+28>:    beq 0x2604 <memmove+180>
0x2570 <memmove+32>:    bls 0x25bc <memmove+108>
0x2572 <memmove+34>:    movel a0,d1
0x2574 <memmove+36>:    movel d1,d0
0x2576 <memmove+38>:    orl d3,d0
0x2578 <memmove+40>:    movel #3,d4
0x257a <memmove+42>:    andl d4,d0
0x257c <memmove+44>:    beq 0x25a2 <memmove+82>
0x257e <memmove+46>:    movel d1,d0
0x2580 <memmove+48>:    eorl d3,d0
0x2582 <memmove+50>:    movel #3,d4
0x2584 <memmove+52>:    andl d4,d0
0x2586 <memmove+54>:    bne 0x258e <memmove+62>
0x2588 <memmove+56>:    movel #3,d4
0x258a <memmove+58>:    cmpl d2,d4
0x258c <memmove+60>:    bcs 0x2592 <memmove+66>
0x258e <memmove+62>:    movel d2,d1
0x2590 <memmove+64>:    bra 0x259a <memmove+74>
0x2592 <memmove+66>:    movel #3,d0
0x2594 <memmove+68>:    andl d1,d0
0x2596 <memmove+70>:    movel #4,d1
0x2598 <memmove+72>:    subl d0,d1
0x259a <memmove+74>:    subl d1,d2
0x259c <memmove+76>:    moveb a0@+,a1@+
0x259e <memmove+78>:    subl #1,d1
0x25a0 <memmove+80>:    bne 0x259c <memmove+76>
0x25a2 <memmove+82>:    movel d2,d1
0x25a4 <memmove+84>:    lsrl #2,d1
0x25a6 <memmove+86>:    beq 0x25ae <memmove+94>
0x25a8 <memmove+88>:    movel a0@+,a1@+
0x25aa <memmove+90>:    subl #1,d1
0x25ac <memmove+92>:    bne 0x25a8 <memmove+88>
0x25ae <memmove+94>:    movel #3,d1
0x25b0 <memmove+96>:    andl d2,d1
0x25b2 <memmove+98>:    beq 0x2604 <memmove+180>
0x25b4 <memmove+100>:   moveb a0@+,a1@+
0x25b6 <memmove+102>:   subl #1,d1
0x25b8 <memmove+104>:   bne 0x25b4 <memmove+100>
```

```
0x25ba <memmove+106>:   bra 0x2604 <memmove+180>
0x25bc <memmove+108>:   addal d2,a0
0x25be <memmove+110>:   addal d2,a1
0x25c0 <memmove+112>:   movel a0,d1
0x25c2 <memmove+114>:   movel a1,d0
0x25c4 <memmove+116>:   orl d1,d0
0x25c6 <memmove+118>:   movel #3,d4
0x25c8 <memmove+120>:   andl d4,d0
0x25ca <memmove+122>:   beq 0x25ec <memmove+156>
0x25cc <memmove+124>:   movel a1,d0
0x25ce <memmove+126>:   eorl d1,d0
0x25d0 <memmove+128>:   movel #3,d4
0x25d2 <memmove+130>:   andl d4,d0
0x25d4 <memmove+132>:   bne 0x25dc <memmove+140>
0x25d6 <memmove+134>:   movel #4,d4
0x25d8 <memmove+136>:   cmpl d2,d4
0x25da <memmove+138>:   bcs 0x25e0 <memmove+144>
0x25dc <memmove+140>:   movel d2,d1
0x25de <memmove+142>:   bra 0x25e4 <memmove+148>
0x25e0 <memmove+144>:   movel #3,d4
0x25e2 <memmove+146>:   andl d4,d1
0x25e4 <memmove+148>:   subl d1,d2
0x25e6 <memmove+150>:   moveb a0@-,a1@-
0x25e8 <memmove+152>:   subl #1,d1
0x25ea <memmove+154>:   bne 0x25e6 <memmove+150>
0x25ec <memmove+156>:   movel d2,d1
0x25ee <memmove+158>:   lsrl #2,d1
0x25f0 <memmove+160>:   beq 0x25f8 <memmove+168>
0x25f2 <memmove+162>:   movel a0@-,a1@-
0x25f4 <memmove+164>:   subl #1,d1
0x25f6 <memmove+166>:   bne 0x25f2 <memmove+162>
0x25f8 <memmove+168>:   movel #3,d1
0x25fa <memmove+170>:   andl d2,d1
0x25fc <memmove+172>:   beq 0x2604 <memmove+180>
0x25fe <memmove+174>:   moveb a0@-,a1@-
0x2600 <memmove+176>:   subl #1,d1
0x2602 <memmove+178>:   bne 0x25fe <memmove+174>
0x2604 <memmove+180>:   movel d3,d0
0x2606 <memmove+182>:   moveml fp@(-12),d2-d4
0x260c <memmove+188>:   unlk fp
0x260e <memmove+190>:   rts
```

The machine code of the above program is:

```
<memmove>:       0x4e56  0x0000  0x48e7  0x3800  0x262e  0x0008  0x242e  0x0010
<memmove+16>:    0x2243  0x206e  0x000c  0x6700  0x009c  0xb1c3  0x6700  0x0096
<memmove+32>:    0x634a  0x2208  0x2001  0x8083  0x7803  0xc084  0x6724  0x2001
<memmove+48>:    0xb780  0x7803  0xc084  0x6606  0x7803  0xb882  0x6504  0x2202
<memmove+64>:    0x6008  0x7003  0xc081  0x7204  0x9280  0x9481  0x12d8  0x5381
<memmove+80>:    0x66fa  0x2202  0xe489  0x6706  0x22d8  0x5381  0x66fa  0x7203
<memmove+96>:    0xc282  0x6750  0x12d8  0x5381  0x66fa  0x6048  0xd1c2  0xd3c2
<memmove+112>:   0x2208  0x2009  0x8081  0x7803  0xc084  0x6720  0x2009  0xb380
<memmove+128>:   0x7803  0xc084  0x6606  0x7804  0xb882  0x6504  0x2202  0x6004
<memmove+144>:   0x7803  0xc284  0x9481  0x1320  0x5381  0x66fa  0x2202  0xe489
```

```
<memmove+160>:   0x6706  0x2320  0x5381  0x66fa  0x7203  0xc282  0x6706  0x1320
<memmove+176>:   0x5381  0x66fa  0x2003  0x4cee  0x001c  0xfff4  0x4e5e  0x4e75

'(78      86      0       0       72      231     56      0
  38      46      0       8       36      46      0       16
  34      67      32      110     0       12      103     0
  0       156     177     195     103     0       0       150
  99      74      34      8       32      1       128     131
  120     3       192     132     103     36      32      1
  183     128     120     3       192     132     102     6
  120     3       184     130     101     4       34      2
  96      8       112     3       192     129     114     4
  146     128     148     129     18      216     83      129
  102     250     34      2       228     137     103     6
  34      216     83      129     102     250     114     3
  194     130     103     80      18      216     83      129
  102     250     96      72      209     194     211     194
  34      8       32      9       128     129     120     3
  192     132     103     32      32      9       179     128
  120     3       192     132     102     6       120     4
  184     130     101     4       34      2       96      4
  120     3       194     132     148     129     19      32
  83      129     102     250     34      2       228     137
  103     6       35      32      83      129     102     250
  114     3       194     130     103     6       19      32
  83      129     102     250     32      3       76      238
  0       28      255     244     78      94      78      117)
|#

;  in the logic, the above program is defined by (memmove-code).
(defn memmove-code ()
  '(78      86      0       0       72      231     56      0
    38      46      0       8       36      46      0       16
    34      67      32      110     0       12      103     0
    0       156     177     195     103     0       0       150
    99      74      34      8       32      1       128     131
    120     3       192     132     103     36      32      1
    183     128     120     3       192     132     102     6
    120     3       184     130     101     4       34      2
    96      8       112     3       192     129     114     4
    146     128     148     129     18      216     83      129
    102     250     34      2       228     137     103     6
    34      216     83      129     102     250     114     3
    194     130     103     80      18      216     83      129
    102     250     96      72      209     194     211     194
    34      8       32      9       128     129     120     3
    192     132     103     32      32      9       179     128
    120     3       192     132     102     6       120     4
    184     130     101     4       34      2       96      4
    120     3       194     132     148     129     19      32
    83      129     102     250     34      2       228     137
    103     6       35      32      83      129     102     250
    114     3       194     130     103     6       19      32
    83      129     102     250     32      3       76      238
```

```
    0       28      255     244     78      94      78      117))

; the preconditions of the initial state.
(defn memmove-statep (s str1 n lst1 str2 lst2)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (mc-pc s) (mc-mem s) 192)
       (mcode-addrp (mc-pc s) (mc-mem s) (memmove-code))
       (ram-addrp (sub 32 16 (read-sp s)) (mc-mem s) 32)
       (ram-addrp str1 (mc-mem s) n)
       (mem-lst 1 str1 (mc-mem s) n lst1)
       (ram-addrp str2 (mc-mem s) n)
       (mem-lst 1 str2 (mc-mem s) n lst2)
       (disjoint (sub 32 16 (read-sp s)) 32 str1 n)
       (disjoint (sub 32 16 (read-sp s)) 32 str2 n)
       (equal str1 (read-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal str2 (read-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
       (equal n (uread-mem (add 32 (read-sp s) 12) (mc-mem s) 4))
       (uint-rangep (plus (nat-to-uint str1) n) 32)
       (uint-rangep (plus (nat-to-uint str2) n) 32)))

; intermediate states.
(defn memmove-s0p (s i* i str1 n lst1 str2 lst2 nt n_)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 76 (mc-pc s)) (mc-mem s) 192)
       (mcode-addrp (sub 32 76 (mc-pc s)) (mc-mem s) (memmove-code))
       (ram-addrp (sub 32 12 (read-an 32 6 s)) (mc-mem s) 32)
       (ram-addrp str1 (mc-mem s) n_)
       (mem-lst 1 str1 (mc-mem s) n_ lst1)
       (ram-addrp (add 32 str2 i*) (mc-mem s) (difference n_ i))
       (mem-lst 1 (add 32 str2 i*) (mc-mem s) (difference n_ i) (mcdr i lst2))
       (disjoint (sub 32 12 (read-an 32 6 s)) 32 str1 n_)
       (disjoint (sub 32 12 (read-an 32 6 s)) 32 str2 n_)
       (equal* (read-an 32 1 s) (add 32 str1 i*))
       (equal* (read-an 32 0 s) (add 32 str2 i*))
       (equal str1 (read-dn 32 3 s))
       (equal n (uread-dn 32 2 s))
       (equal nt (uread-dn 32 1 s))
       (lessp (plus (nat-to-uint str1) n_) 4294967296)
       (lessp (plus (nat-to-uint str2) n_) 4294967296)
       (lessp (nat-to-uint str1) (nat-to-uint str2))
       (nat-rangep str2 32)
       (leq (plus i nt n) n_)
       (not (equal nt 0))
       (numberp i*)
       (nat-rangep i* 32)
       (equal i (nat-to-uint i*))
       (not (zerop n_))))

(defn memmove-s1p (s i* i str1 n lst1 str2 lst2 nt n_)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 88 (mc-pc s)) (mc-mem s) 192)
```

```
       (mcode-addrp (sub 32 88 (mc-pc s)) (mc-mem s) (memmove-code))
       (ram-addrp (sub 32 12 (read-an 32 6 s)) (mc-mem s) 32)
       (ram-addrp str1 (mc-mem s) n_)
       (mem-lst 1 str1 (mc-mem s) n_ lst1)
       (ram-addrp (add 32 str2 i*) (mc-mem s) (difference n_ i))
       (mem-lst 1 (add 32 str2 i*) (mc-mem s) (difference n_ i) (mcdr i lst2))
       (disjoint (sub 32 12 (read-an 32 6 s)) 32 str1 n_)
       (disjoint (sub 32 12 (read-an 32 6 s)) 32 str2 n_)
       (equal* (read-an 32 1 s) (add 32 str1 i*))
       (equal* (read-an 32 0 s) (add 32 str2 i*))
       (equal str1 (read-dn 32 3 s))
       (equal n (uread-dn 32 2 s))
       (equal nt (uread-dn 32 1 s))
       (lessp (plus (nat-to-uint str1) n_) 4294967296)
       (lessp (plus (nat-to-uint str2) n_) 4294967296)
       (lessp (nat-to-uint str1) (nat-to-uint str2))
       (nat-rangep str2 32)
       (leq (plus i (times 4 nt) (remainder n 4)) n_)
       (not (equal nt 0))
       (numberp i*)
       (nat-rangep i* 32)
       (equal i (nat-to-uint i*))
       (not (zerop n_))))

(defn memmove-s2p (s i* i str1 n lst1 str2 lst2 nt n_)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 100 (mc-pc s)) (mc-mem s) 192)
       (mcode-addrp (sub 32 100 (mc-pc s)) (mc-mem s) (memmove-code))
       (ram-addrp (sub 32 12 (read-an 32 6 s)) (mc-mem s) 32)
       (ram-addrp str1 (mc-mem s) n_)
       (mem-lst 1 str1 (mc-mem s) n_ lst1)
       (ram-addrp (add 32 str2 i*) (mc-mem s) (difference n_ i))
       (mem-lst 1 (add 32 str2 i*) (mc-mem s) (difference n_ i) (mcdr i lst2))
       (disjoint (sub 32 12 (read-an 32 6 s)) 32 str1 n_)
       (disjoint (sub 32 12 (read-an 32 6 s)) 32 str2 n_)
       (equal* (read-an 32 1 s) (add 32 str1 i*))
       (equal* (read-an 32 0 s) (add 32 str2 i*))
       (equal str1 (read-dn 32 3 s))
       (equal nt (uread-dn 32 1 s))
       (lessp (plus (nat-to-uint str1) n_) 4294967296)
       (lessp (plus (nat-to-uint str2) n_) 4294967296)
       (lessp (nat-to-uint str1) (nat-to-uint str2))
       (nat-rangep str2 32)
       (leq (plus i nt) n_)
       (not (equal nt 0))
       (numberp i*)
       (nat-rangep i* 32)
       (equal i (nat-to-uint i*))
       (not (zerop n_))))

(defn memmove-s3p (s i* i str1 n lst1 str2 lst2 nt n_)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
```

```
       (rom-addrp (sub 32 150 (mc-pc s)) (mc-mem s) 192)
       (mcode-addrp (sub 32 150 (mc-pc s)) (mc-mem s) (memmove-code))
       (ram-addrp (sub 32 12 (read-an 32 6 s)) (mc-mem s) 32)
       (ram-addrp str1 (mc-mem s) n_)
       (mem-lst 1 str1 (mc-mem s) n_ lst1)
       (ram-addrp str2 (mc-mem s) i)
       (mem-lst 1 str2 (mc-mem s) i (mcar i lst2))
       (disjoint (sub 32 12 (read-an 32 6 s)) 32 str1 n_)
       (disjoint (sub 32 12 (read-an 32 6 s)) 32 str2 n_)
       (equal* (read-an 32 1 s) (add 32 str1 i*))
       (equal* (read-an 32 0 s) (add 32 str2 i*))
       (equal str1 (read-dn 32 3 s))
       (equal n (uread-dn 32 2 s))
       (equal nt (uread-dn 32 1 s))
       (lessp (plus (nat-to-uint str1) n_) 4294967296)
       (lessp (plus (nat-to-uint str2) n_) 4294967296)
       (lessp (nat-to-uint str2) (nat-to-uint str1))
       (nat-rangep str2 32)
       (numberp i*)
       (nat-rangep i* 32)
       (equal i (nat-to-uint i*))
       (leq i n_)
       (leq (plus nt n) i)
       (not (zerop nt))
       (not (zerop i))
       (not (zerop n_))))

(defn memmove-s4p (s i* i str1 n lst1 str2 lst2 nt n_)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 162 (mc-pc s)) (mc-mem s) 192)
       (mcode-addrp (sub 32 162 (mc-pc s)) (mc-mem s) (memmove-code))
       (ram-addrp (sub 32 12 (read-an 32 6 s)) (mc-mem s) 32)
       (ram-addrp str1 (mc-mem s) n_)
       (mem-lst 1 str1 (mc-mem s) n_ lst1)
       (ram-addrp str2 (mc-mem s) i)
       (mem-lst 1 str2 (mc-mem s) i (mcar i lst2))
       (disjoint (sub 32 12 (read-an 32 6 s)) 32 str1 n_)
       (disjoint (sub 32 12 (read-an 32 6 s)) 32 str2 n_)
       (equal* (read-an 32 1 s) (add 32 str1 i*))
       (equal* (read-an 32 0 s) (add 32 str2 i*))
       (equal str1 (read-dn 32 3 s))
       (equal n (uread-dn 32 2 s))
       (equal nt (uread-dn 32 1 s))
       (lessp (plus (nat-to-uint str1) n_) 4294967296)
       (lessp (plus (nat-to-uint str2) n_) 4294967296)
       (lessp (nat-to-uint str2) (nat-to-uint str1))
       (nat-rangep str2 32)
       (numberp i*)
       (nat-rangep i* 32)
       (equal i (nat-to-uint i*))
       (leq i n_)
       (leq (plus (times 4 nt) (remainder n 4)) i)
       (not (lessp i 4))
```

```
       (not (zerop nt))
       (not (zerop n_))))

(defn memmove-s5p (s i* i str1 n lst1 str2 lst2 nt n_)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 174 (mc-pc s)) (mc-mem s) 192)
       (mcode-addrp (sub 32 174 (mc-pc s)) (mc-mem s) (memmove-code))
       (ram-addrp (sub 32 12 (read-an 32 6 s)) (mc-mem s) 32)
       (ram-addrp str1 (mc-mem s) n_)
       (mem-lst 1 str1 (mc-mem s) n_ lst1)
       (ram-addrp str2 (mc-mem s) i)
       (mem-lst 1 str2 (mc-mem s) i (mcar i lst2))
       (disjoint (sub 32 12 (read-an 32 6 s)) 32 str1 n_)
       (disjoint (sub 32 12 (read-an 32 6 s)) 32 str2 n_)
       (equal* (read-an 32 1 s) (add 32 str1 i*))
       (equal* (read-an 32 0 s) (add 32 str2 i*))
       (equal str1 (read-dn 32 3 s))
       (equal nt (uread-dn 32 1 s))
       (lessp (plus (nat-to-uint str1) n_) 4294967296)
       (lessp (plus (nat-to-uint str2) n_) 4294967296)
       (lessp (nat-to-uint str2) (nat-to-uint str1))
       (nat-rangep str2 32)
       (numberp i*)
       (nat-rangep i* 32)
       (equal i (nat-to-uint i*))
       (leq i n_)
       (leq nt i)
       (not (zerop nt))
       (not (zerop i))
       (not (zerop n_))))

; enable a few events for this proof.
(enable disjoint-leq-uint)
(enable disjoint-leq1-uint)
(enable add-uintxxx)
(enable plus-times-sub1)

; from the initial state to exit:  s --> sn, when n == 0.
(prove-lemma memmove-s-sn-1 (rewrite)
     (let ((sn (stepn s 11)))
       (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                     (zerop n))
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (mem-lst 1 str1 (mc-mem sn) n lst1)
                     (equal (read-dn 32 0 sn) str1)
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (read-an 32 6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 7 s) 4)))))))

(prove-lemma memmove-s-sn-rfile-1 (rewrite)
     (implies (and (memmove-statep s str1 n lst1 str2 lst2)
```

```
                (zerop n)
                (leq oplen 32)
                (d2-7a2-5p rn))
           (equal (read-rn oplen rn (mc-rfile (stepn s 11)))
                  (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memmove-s-sn-mem-1 (rewrite)
     (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                (zerop n)
                (disjoint x k (sub 32 16 (read-sp s)) 32))
           (equal (read-mem x (mc-mem (stepn s 11)) k)
                  (read-mem x (mc-mem s) k))))

; from the initial state to exit:  s --> sn, when n =- 0 and str1 == str2.
(prove-lemma memmove-s-sn-2 (rewrite)
     (let ((sn (stepn s 13)))
       (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                     (not (zerop n))
                     (equal (nat-to-uint str1) (nat-to-uint str2)))
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (mem-lst 1 str1 (mc-mem sn) n lst2)
                     (equal (read-dn 32 0 sn) str1)
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (read-an 32 6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 7 s) 4)))))
     ((enable mem-lst-same)))

(prove-lemma memmove-s-sn-rfile-2 (rewrite)
     (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                (not (zerop n))
                (equal (nat-to-uint str1) (nat-to-uint str2))
                (leq oplen 32)
                (d2-7a2-5p rn))
           (equal (read-rn oplen rn (mc-rfile (stepn s 13)))
                  (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memmove-s-sn-mem-2 (rewrite)
     (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                (not (zerop n))
                (equal (nat-to-uint str1) (nat-to-uint str2))
                (disjoint x k (sub 32 16 (read-sp s)) 32))
           (equal (read-mem x (mc-mem (stepn s 13)) k)
                  (read-mem x (mc-mem s) k))))

; from the initial state s to s0:  s --> s0.
(prove-lemma memmove-s-s0-1 (rewrite)
     (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                (not (zerop n))
                (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                (lessp (nat-to-uint str1) (nat-to-uint str2))
                (not (equal (remainder (nat-to-uint str1) 4)
                            (remainder (nat-to-uint str2) 4)))
```

```
                (or (not (equal (remainder (nat-to-uint str1) 4) 0))
                    (not (equal (remainder (nat-to-uint str2) 4) 0))))
            (memmove-s0p (stepn s 24) 0 0 str1 0 lst1 str2 lst2 n n)))

(prove-lemma memmove-s-s0-else-1 (rewrite)
     (let ((s0 (stepn s 24)))
       (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                     (not (zerop n))
                     (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                     (lessp (nat-to-uint str1) (nat-to-uint str2))
                     (not (equal (remainder (nat-to-uint str1) 4)
                                 (remainder (nat-to-uint str2) 4)))
                     (or (not (equal (remainder (nat-to-uint str1) 4) 0))
                         (not (equal (remainder (nat-to-uint str2) 4) 0))))
                (and (equal (linked-rts-addr s0) (rts-addr s))
                     (equal (linked-a6 s0) (read-an 32 6 s))
                     (equal (read-rn 32 14 (mc-rfile s0))
                            (sub 32 4 (read-sp s)))
                     (equal (movem-saved s0 4 12 3)
                            (readm-rn 32 '(2 3 4) (mc-rfile s)))))))

(prove-lemma memmove-s-s0-rfile-1 (rewrite)
     (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                   (not (zerop n))
                   (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                   (lessp (nat-to-uint str1) (nat-to-uint str2))
                   (not (equal (remainder (nat-to-uint str1) 4)
                               (remainder (nat-to-uint str2) 4)))
                   (or (not (equal (remainder (nat-to-uint str1) 4) 0))
                       (not (equal (remainder (nat-to-uint str2) 4) 0)))
                   (d5-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 24)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memmove-s-s0-mem-1 (rewrite)
     (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                   (not (zerop n))
                   (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                   (lessp (nat-to-uint str1) (nat-to-uint str2))
                   (not (equal (remainder (nat-to-uint str1) 4)
                               (remainder (nat-to-uint str2) 4)))
                   (or (not (equal (remainder (nat-to-uint str1) 4) 0))
                       (not (equal (remainder (nat-to-uint str2) 4) 0)))
                   (disjoint x k (sub 32 16 (read-sp s)) 32))
              (equal (read-mem x (mc-mem (stepn s 24)) k)
                     (read-mem x (mc-mem s) k))))

(prove-lemma memmove-s-s0-2 (rewrite)
     (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                   (not (zerop n))
                   (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                   (lessp (nat-to-uint str1) (nat-to-uint str2))
                   (equal (remainder (nat-to-uint str1) 4)
                          (remainder (nat-to-uint str2) 4))
```

```
                (leq n 3)
                (or (not (equal (remainder (nat-to-uint str1) 4) 0))
                    (not (equal (remainder (nat-to-uint str2) 4) 0))))
           (memmove-s0p (stepn s 27) 0 0 str1 0 lst1 str2 lst2 n n)))

(prove-lemma memmove-s-s0-else-2 (rewrite)
     (let ((s0 (stepn s 27)))
       (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                     (not (zerop n))
                     (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                     (lessp (nat-to-uint str1) (nat-to-uint str2))
                     (equal (remainder (nat-to-uint str1) 4)
                            (remainder (nat-to-uint str2) 4))
                     (leq n 3)
                     (or (not (equal (remainder (nat-to-uint str1) 4) 0))
                         (not (equal (remainder (nat-to-uint str2) 4) 0))))
                (and (equal (linked-rts-addr s0) (rts-addr s))
                     (equal (linked-a6 s0) (read-an 32 6 s))
                     (equal (read-rn 32 14 (mc-rfile s0))
                            (sub 32 4 (read-sp s)))
                     (equal (movem-saved s0 4 12 3)
                            (readm-rn 32 '(2 3 4) (mc-rfile s)))))))

(prove-lemma memmove-s-s0-rfile-2 (rewrite)
     (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                   (not (zerop n))
                   (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                   (lessp (nat-to-uint str1) (nat-to-uint str2))
                   (equal (remainder (nat-to-uint str1) 4)
                          (remainder (nat-to-uint str2) 4))
                   (leq n 3)
                   (or (not (equal (remainder (nat-to-uint str1) 4) 0))
                       (not (equal (remainder (nat-to-uint str2) 4) 0)))
                   (d5-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 27)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memmove-s-s0-mem-2 (rewrite)
     (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                   (not (zerop n))
                   (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                   (lessp (nat-to-uint str1) (nat-to-uint str2))
                   (equal (remainder (nat-to-uint str1) 4)
                          (remainder (nat-to-uint str2) 4))
                   (leq n 3)
                   (or (not (equal (remainder (nat-to-uint str1) 4) 0))
                       (not (equal (remainder (nat-to-uint str2) 4) 0)))
                   (disjoint x k (sub 32 16 (read-sp s)) 32))
              (equal (read-mem x (mc-mem (stepn s 27)) k)
                     (read-mem x (mc-mem s) k))))

(prove-lemma memmove-s-s0-3 (rewrite)
     (let ((r (remainder (nat-to-uint str1) 4)))
       (implies (and (memmove-statep s str1 n lst1 str2 lst2)
```

```
                    (not (zerop n))
                    (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                    (lessp (nat-to-uint str1) (nat-to-uint str2))
                    (equal (remainder (nat-to-uint str1) 4)
                           (remainder (nat-to-uint str2) 4))
                    (lessp 3 n)
                    (or (not (equal (remainder (nat-to-uint str1) 4) 0))
                        (not (equal (remainder (nat-to-uint str2) 4) 0))))
               (memmove-s0p (stepn s 29) 0 0 str1 (difference (plus n r) 4)
                            lst1 str2 lst2 (difference 4 r) n))))

(prove-lemma memmove-s-s0-else-3 (rewrite)
     (let ((s0 (stepn s 29)))
       (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                     (not (zerop n))
                     (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                     (lessp (nat-to-uint str1) (nat-to-uint str2))
                     (equal (remainder (nat-to-uint str1) 4)
                            (remainder (nat-to-uint str2) 4))
                     (lessp 3 n)
                     (or (not (equal (remainder (nat-to-uint str1) 4) 0))
                         (not (equal (remainder (nat-to-uint str2) 4) 0))))
                (and (equal (linked-rts-addr s0) (rts-addr s))
                     (equal (linked-a6 s0) (read-an 32 6 s))
                     (equal (read-rn 32 14 (mc-rfile s0))
                            (sub 32 4 (read-sp s)))
                     (equal (movem-saved s0 4 12 3)
                            (readm-rn 32 '(2 3 4) (mc-rfile s)))))))

(prove-lemma memmove-s-s0-rfile-3 (rewrite)
     (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                   (not (zerop n))
                   (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                   (lessp (nat-to-uint str1) (nat-to-uint str2))
                   (equal (remainder (nat-to-uint str1) 4)
                          (remainder (nat-to-uint str2) 4))
                   (lessp 3 n)
                   (or (not (equal (remainder (nat-to-uint str1) 4) 0))
                       (not (equal (remainder (nat-to-uint str2) 4) 0)))
                   (d5-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 29)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memmove-s-s0-mem-3 (rewrite)
     (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                   (not (zerop n))
                   (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                   (lessp (nat-to-uint str1) (nat-to-uint str2))
                   (equal (remainder (nat-to-uint str1) 4)
                          (remainder (nat-to-uint str2) 4))
                   (lessp 3 n)
                   (or (not (equal (remainder (nat-to-uint str1) 4) 0))
                       (not (equal (remainder (nat-to-uint str2) 4) 0)))
                   (disjoint x k (sub 32 16 (read-sp s)) 32))
```

```
              (equal (read-mem x (mc-mem (stepn s 29)) k)
                     (read-mem x (mc-mem s) k))))

; from the initial state s to s1:  s --> s1.
(prove-lemma memmove-s-s1 (rewrite)
     (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                   (not (zerop n))
                   (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                   (lessp (nat-to-uint str1) (nat-to-uint str2))
                   (equal (remainder (nat-to-uint str1) 4) 0)
                   (equal (remainder (nat-to-uint str2) 4) 0)
                   (not (lessp n 4)))
              (memmove-s1p (stepn s 19) 0 0 str1 n lst1
                           str2 lst2 (quotient n 4) n)))

(prove-lemma memmove-s-s1-else (rewrite)
     (let ((s1 (stepn s 19)))
       (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                     (not (zerop n))
                     (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                     (lessp (nat-to-uint str1) (nat-to-uint str2))
                     (equal (remainder (nat-to-uint str1) 4) 0)
                     (equal (remainder (nat-to-uint str2) 4) 0)
                     (not (lessp n 4)))
                (and (equal (linked-rts-addr s1) (rts-addr s))
                     (equal (linked-a6 s1) (read-an 32 6 s))
                     (equal (read-rn 32 14 (mc-rfile s1))
                            (sub 32 4 (read-sp s)))
                     (equal (movem-saved s1 4 12 3)
                            (readm-rn 32 '(2 3 4) (mc-rfile s)))))))

(prove-lemma memmove-s-s1-rfile (rewrite)
     (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                   (not (zerop n))
                   (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                   (lessp (nat-to-uint str1) (nat-to-uint str2))
                   (equal (remainder (nat-to-uint str1) 4) 0)
                   (equal (remainder (nat-to-uint str2) 4) 0)
                   (not (lessp n 4))
                   (d5-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 19)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memmove-s-s1-mem (rewrite)
     (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                   (not (zerop n))
                   (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                   (lessp (nat-to-uint str1) (nat-to-uint str2))
                   (equal (remainder (nat-to-uint str1) 4) 0)
                   (equal (remainder (nat-to-uint str2) 4) 0)
                   (not (lessp n 4))
                   (disjoint x k (sub 32 16 (read-sp s)) 32))
              (equal (read-mem x (mc-mem (stepn s 19)) k)
                     (read-mem x (mc-mem s) k))))
```

```
; from s to s2.  s --> s2.
(prove-lemma memmove-s-s2 (rewrite)
     (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                   (not (zerop n))
                   (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                   (lessp (nat-to-uint str1) (nat-to-uint str2))
                   (equal (remainder (nat-to-uint str1) 4) 0)
                   (equal (remainder (nat-to-uint str2) 4) 0)
                   (lessp n 4))
              (memmove-s2p (stepn s 22) 0 0 str1 n lst1 str2 lst2 n n)))

(prove-lemma memmove-s-s2-else (rewrite)
     (let ((s2 (stepn s 22)))
       (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                     (not (zerop n))
                     (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                     (lessp (nat-to-uint str1) (nat-to-uint str2))
                     (equal (remainder (nat-to-uint str1) 4) 0)
                     (equal (remainder (nat-to-uint str2) 4) 0)
                     (lessp n 4))
                (and (equal (linked-rts-addr s2) (rts-addr s))
                     (equal (linked-a6 s2) (read-an 32 6 s))
                     (equal (read-rn 32 14 (mc-rfile s2))
                            (sub 32 4 (read-sp s)))
                     (equal (movem-saved s2 4 12 3)
                            (readm-rn 32 '(2 3 4) (mc-rfile s)))))))

(prove-lemma memmove-s-s2-rfile (rewrite)
     (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                   (not (zerop n))
                   (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                   (lessp (nat-to-uint str1) (nat-to-uint str2))
                   (equal (remainder (nat-to-uint str1) 4) 0)
                   (equal (remainder (nat-to-uint str2) 4) 0)
                   (lessp n 4)
                   (d5-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 22)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memmove-s-s2-mem (rewrite)
     (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                   (not (zerop n))
                   (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                   (lessp (nat-to-uint str1) (nat-to-uint str2))
                   (equal (remainder (nat-to-uint str1) 4) 0)
                   (equal (remainder (nat-to-uint str2) 4) 0)
                   (lessp n 4)
                   (disjoint x k (sub 32 16 (read-sp s)) 32))
              (equal (read-mem x (mc-mem (stepn s 22)) k)
                     (read-mem x (mc-mem s) k))))

; s0 --> s1.
(prove-lemma memmove-s0-s1 ()
```

```
     (let ((s1 (stepn s 6)))
       (implies (and (memmove-s0p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (not (lessp n 4)))
                (memmove-s1p s1 (add 32 i* 1) (add1 i) str1 n
                             (put-nth (get-nth i lst2) i lst1)
                             str2 lst2 (quotient n 4) n_)))
     ((disable mcdr)))

(prove-lemma memmove-s0-s1-else (rewrite)
     (let ((s1 (stepn s 6)))
       (implies (and (memmove-s0p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (not (lessp n 4)))
                (and (equal (linked-rts-addr s1) (linked-rts-addr s))
                     (equal (linked-a6 s1) (linked-a6 s))
                     (equal (read-rn oplen 14 (mc-rfile s1))
                            (read-rn oplen 14 (mc-rfile s)))
                     (equal (movem-saved s1 4 12 3)
                            (movem-saved s 4 12 3))))))

(prove-lemma memmove-s0-s1-rfile (rewrite)
     (implies (and (memmove-s0p s i* i str1 n lst1 str2 lst2 nt n_)
                   (equal (sub1 nt) 0)
                   (not (lessp n 4))
                   (d5-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 6)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memmove-s0-s1-mem (rewrite)
     (implies (and (memmove-s0p s i* i str1 n lst1 str2 lst2 nt n_)
                   (equal (sub1 nt) 0)
                   (not (lessp n 4))
                   (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                   (disjoint x k str1 n_))
              (equal (read-mem x (mc-mem (stepn s 6)) k)
                     (read-mem x (mc-mem s) k)))))

; s0 --> s2.
(prove-lemma memmove-s0-s2 ()
     (let ((s2 (stepn s 9)))
       (implies (and (memmove-s0p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (lessp n 4)
                     (not (zerop n)))
                (memmove-s2p s2 (add 32 i* 1) (add1 i) str1 n
                             (put-nth (get-nth i lst2) i lst1) str2
                             lst2 n n_)))
     ((disable mcdr)))

(prove-lemma memmove-s0-s2-else (rewrite)
     (let ((s2 (stepn s 9)))
       (implies (and (memmove-s0p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
```

```
                    (lessp n 4)
                    (not (zerop n)))
               (and (equal (linked-rts-addr s2) (linked-rts-addr s))
                    (equal (linked-a6 s2) (linked-a6 s))
                    (equal (read-rn oplen 14 (mc-rfile s2))
                           (read-rn oplen 14 (mc-rfile s)))
                    (equal (movem-saved s2 4 12 3)
                           (movem-saved s 4 12 3))))))

(prove-lemma memmove-s0-s2-rfile (rewrite)
     (implies (and (memmove-s0p s i* i str1 n lst1 str2 lst2 nt n_)
                   (equal (sub1 nt) 0)
                   (lessp n 4)
                   (not (zerop n))
                   (d5-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 9)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memmove-s0-s2-mem (rewrite)
     (implies (and (memmove-s0p s i* i str1 n lst1 str2 lst2 nt n_)
                   (equal (sub1 nt) 0)
                   (lessp n 4)
                   (not (zerop n))
                   (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                   (disjoint x k str1 n_))
              (equal (read-mem x (mc-mem (stepn s 9)) k)
                     (read-mem x (mc-mem s) k))))

; s0 --> s0.
(prove-lemma memmove-s0-s0 (rewrite)
     (let ((s0 (stepn s 3)))
       (implies (and (memmove-s0p s i* i str1 n lst1 str2 lst2 nt n_)
                     (not (equal (sub1 nt) 0)))
                (and (memmove-s0p s0 (add 32 i* 1) (add1 i) str1 n
                                  (put-nth (get-nth i lst2) i lst1) str2
                                  lst2 (sub1 nt) n_)
                     (equal (linked-rts-addr s0) (linked-rts-addr s))
                     (equal (linked-a6 s0) (linked-a6 s))
                     (equal (read-rn oplen 14 (mc-rfile s0))
                            (read-rn oplen 14 (mc-rfile s)))
                     (equal (movem-saved s0 4 12 3)
                            (movem-saved s 4 12 3))))))
     ((disable mcdr)))

(prove-lemma memmove-s0-s0-rfile (rewrite)
     (implies (and (memmove-s0p s i* i str1 n lst1 str2 lst2 nt n_)
                   (not (equal (sub1 nt) 0))
                   (d5-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 3)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memmove-s0-s0-mem (rewrite)
     (implies (and (memmove-s0p s i* i str1 n lst1 str2 lst2 nt n_)
                   (not (equal (sub1 nt) 0))
```

```
                (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                (disjoint x k str1 n_))
           (equal (read-mem x (mc-mem (stepn s 3)) k)
                  (read-mem x (mc-mem s) k))))

; s1 --> s2.
(prove-lemma memmove-s1-s2 ()
     (let ((s2 (stepn s 6)))
       (implies (and (memmove-s1p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (not (equal (remainder n 4) 0)))
                (memmove-s2p s2 (add 32 i* 4) (plus 4 i) str1 n
                             (movn-lst 4 lst1 lst2 i) str2 lst2
                             (remainder n 4) n_)))
     ((disable mcdr plus)))

(prove-lemma memmove-s1-s2-else (rewrite)
     (let ((s2 (stepn s 6)))
       (implies (and (memmove-s1p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (not (equal (remainder n 4) 0)))
                (and (equal (linked-rts-addr s2) (linked-rts-addr s))
                     (equal (linked-a6 s2) (linked-a6 s))
                     (equal (read-rn oplen 14 (mc-rfile s2))
                            (read-rn oplen 14 (mc-rfile s)))
                     (equal (movem-saved s2 4 12 3)
                            (movem-saved s 4 12 3))))))

(prove-lemma memmove-s1-s2-rfile (rewrite)
     (implies (and (memmove-s1p s i* i str1 n lst1 str2 lst2 nt n_)
                   (equal (sub1 nt) 0)
                   (not (equal (remainder n 4) 0))
                   (d5-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 6)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memmove-s1-s2-mem (rewrite)
     (implies (and (memmove-s1p s i* i str1 n lst1 str2 lst2 nt n_)
                   (equal (sub1 nt) 0)
                   (not (equal (remainder n 4) 0))
                   (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                   (disjoint x k str1 n_))
              (equal (read-mem x (mc-mem (stepn s 6)) k)
                     (read-mem x (mc-mem s) k)))))

; s1 --> s1.
(prove-lemma memmove-s1-s1 (rewrite)
     (let ((s1 (stepn s 3)))
       (implies (and (memmove-s1p s i* i str1 n lst1 str2 lst2 nt n_)
                     (not (equal (sub1 nt) 0)))
                (and (memmove-s1p s1 (add 32 i* 4) (plus 4 i) str1 n
                                  (movn-lst 4 lst1 lst2 i) str2 lst2
                                  (sub1 nt) n_)
                     (equal (linked-rts-addr s1) (linked-rts-addr s))
```

```
                    (equal (linked-a6 s1) (linked-a6 s))
                    (equal (read-rn oplen 14 (mc-rfile s1))
                           (read-rn oplen 14 (mc-rfile s)))
                    (equal (movem-saved s1 4 12 3)
                           (movem-saved s 4 12 3)))))
     ((disable mcdr plus)))

(prove-lemma memmove-s1-s1-rfile (rewrite)
     (implies (and (memmove-s1p s i* i str1 n lst1 str2 lst2 nt n_)
                   (not (equal (sub1 nt) 0))
                   (d5-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 3)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memmove-s1-s1-mem (rewrite)
     (implies (and (memmove-s1p s i* i str1 n lst1 str2 lst2 nt n_)
                   (not (equal (sub1 nt) 0))
                   (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                   (disjoint x k str1 n_))
              (equal (read-mem x (mc-mem (stepn s 3)) k)
                     (read-mem x (mc-mem s) k)))))

; from s2 to exit:  s2 --> sn.
; base case:  s2 --> sn.
(prove-lemma memmove-s2-sn-base (rewrite)
     (let ((sn (stepn s 8)))
       (implies (and (memmove-s2p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0))
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) str1)
                     (mem-lst 1 str1 (mc-mem sn) n_
                              (put-nth (get-nth i lst2) i lst1))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))))))

(prove-lemma memmove-s2-sn-rfile-base (rewrite)
     (implies (and (memmove-s2p s i* i str1 n lst1 str2 lst2 nt n_)
                   (equal (sub1 nt) 0)
                   (d2-7a2-5p rn)
                   (leq oplen 32))
              (equal (read-rn oplen rn (mc-rfile (stepn s 8)))
                     (if (d5-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (get-vlst oplen 0 rn '(2 3 4)
                                 (movem-saved s 4 12 3))))))

(prove-lemma memmove-s2-sn-mem-base (rewrite)
     (implies (and (memmove-s2p s i* i str1 n lst1 str2 lst2 nt n_)
                   (equal (sub1 nt) 0)
                   (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                   (disjoint x k str1 n_))
```

```
              (equal (read-mem x (mc-mem (stepn s 8)) k)
                     (read-mem x (mc-mem s) k))))

; induction case:  s2 --> s2.
(prove-lemma memmove-s2-s2 (rewrite)
     (let ((s2 (stepn s 3)))
       (implies (and (memmove-s2p s i* i str1 n lst1 str2 lst2 nt n_)
                     (not (equal (sub1 nt) 0)))
                (and (memmove-s2p s2 (add 32 i* 1) (add1 i) str1 n
                                  (put-nth (get-nth i lst2) i lst1) str2
                                  lst2 (sub1 nt) n_)
                     (equal (linked-rts-addr s2) (linked-rts-addr s))
                     (equal (linked-a6 s2) (linked-a6 s))
                     (equal (read-rn oplen 14 (mc-rfile s2))
                            (read-rn oplen 14 (mc-rfile s)))
                     (equal (movem-saved s2 4 12 3)
                            (movem-saved s 4 12 3)))))
     ((disable mcdr)))

(prove-lemma memmove-s2-s2-rfile (rewrite)
     (implies (and (memmove-s2p s i* i str1 n lst1 str2 lst2 nt n_)
                   (not (equal (sub1 nt) 0))
                   (d5-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 3)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memmove-s2-s2-mem (rewrite)
     (implies (and (memmove-s2p s i* i str1 n lst1 str2 lst2 nt n_)
                   (not (equal (sub1 nt) 0))
                   (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                   (disjoint x k str1 n_))
              (equal (read-mem x (mc-mem (stepn s 3)) k)
                     (read-mem x (mc-mem s) k))))

; put together: s2 --> sn.
(defn memmove-s2-sn-t (i* i str1 n lst1 str2 lst2 nt n_)
  (if (equal (sub1 nt) 0)
      8
    (splus 3 (memmove-s2-sn-t (add 32 i* 1) (add1 i) str1 n
                              (put-nth (get-nth i lst2) i lst1) str2
                              lst2 (sub1 nt) n_))))

(defn memmove-s2-sn-induct (s i* i lst1 lst2 nt)
  (if (equal (sub1 nt) 0)
      t
    (memmove-s2-sn-induct (stepn s 3) (add 32 i* 1) (add1 i)
                          (put-nth (get-nth i lst2) i lst1) lst2 (sub1 nt))))

(prove-lemma memmove-s2p-info (rewrite)
     (implies (memmove-s2p s i* i str1 n lst1 str2 lst2 nt n_)
              (and (numberp nt) (not (equal nt 0)))))

(prove-lemma memmove-s2-sn (rewrite)
     (let ((sn (stepn s (memmove-s2-sn-t i* i str1 n lst1 str2 lst2 nt n_))))
```

```
       (implies (memmove-s2p s i* i str1 n lst1 str2 lst2 nt n_)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) str1)
                     (mem-lst 1 str1 (mc-mem sn) n_ (mmov1-lst i lst1 lst2 nt))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8)))))
       ((induct (memmove-s2-sn-induct s i* i lst1 lst2 nt))
        (disable memmove-s2p read-dn)))

(prove-lemma memmove-s2-sn-rfile (rewrite)
     (let ((sn (stepn s (memmove-s2-sn-t i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s2p s i* i str1 n lst1 str2 lst2 nt n_)
                     (d2-7a2-5p rn)
                     (leq oplen 32))
                (equal (read-rn oplen rn (mc-rfile sn))
                       (if (d5-7a2-5p rn)
                           (read-rn oplen rn (mc-rfile s))
                         (get-vlst oplen 0 rn '(2 3 4)
                                   (movem-saved s 4 12 3))))))
     ((induct (memmove-s2-sn-induct s i* i lst1 lst2 nt))
      (disable memmove-s2p)))

(prove-lemma memmove-s2-sn-mem (rewrite)
     (let ((sn (stepn s (memmove-s2-sn-t i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s2p s i* i str1 n lst1 str2 lst2 nt n_)
                     (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                     (disjoint x k str1 n_))
              (equal (read-mem x (mc-mem sn) k)
                     (read-mem x (mc-mem s) k))))
     ((induct (memmove-s2-sn-induct s i* i lst1 lst2 nt))
      (disable memmove-s2p)))

(disable memmove-s2p-info)

; from s1 to sn: s1 --> sn.
; base case 1: s1 --> sn.
(prove-lemma memmove-s1-sn-base-1 (rewrite)
     (let ((sn (stepn s 10)))
       (implies (and (memmove-s1p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (equal (remainder n 4) 0))
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) str1)
                     (mem-lst 1 str1 (mc-mem sn) n_ (movn-lst 4 lst1 lst2 i))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8)))))
     ((disable mcdr plus)))
```

```
(prove-lemma memmove-s1-sn-rfile-base-1 (rewrite)
     (implies (and (memmove-s1p s i* i str1 n lst1 str2 lst2 nt n_)
                   (equal (sub1 nt) 0)
                   (equal (remainder n 4) 0)
                   (d2-7a2-5p rn)
                   (leq oplen 32))
              (equal (read-rn oplen rn (mc-rfile (stepn s 10)))
                     (if (d5-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (get-vlst oplen 0 rn '(2 3 4)
                                 (movem-saved s 4 12 3))))))

(prove-lemma memmove-s1-sn-mem-base-1 (rewrite)
     (implies (and (memmove-s1p s i* i str1 n lst1 str2 lst2 nt n_)
                   (equal (sub1 nt) 0)
                   (equal (remainder n 4) 0)
                   (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                   (disjoint x k str1 n_))
              (equal (read-mem x (mc-mem (stepn s 10)) k)
                     (read-mem x (mc-mem s) k))))

; base case 2: s1 --> s2 --> sn.
(defn memmove-s1-sn-t0 (i* i str1 n lst1 str2 lst2 nt n_)
  (splus 6 (memmove-s2-sn-t (add 32 i* 4) (plus 4 i) str1 n
                            (movn-lst 4 lst1 lst2 i) str2 lst2
                            (remainder n 4) n_)))

(prove-lemma memmove-s1-sn-base-2 (rewrite)
     (let ((sn (stepn s (memmove-s1-sn-t0 i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s1p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (not (equal (remainder n 4) 0)))
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) str1)
                     (mem-lst 1 str1 (mc-mem sn) n_
                              (mmov1-lst (plus 4 i) (movn-lst 4 lst1 lst2 i)
                                         lst2 (remainder n 4)))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))))))
     ((use (memmove-s1-s2))
      (disable memmove-s1p memmove-s2p read-dn movn-lst)))

(prove-lemma memmove-s1-sn-rfile-base-2 (rewrite)
     (let ((sn (stepn s (memmove-s1-sn-t0 i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s1p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (not (equal (remainder n 4) 0))
                     (d2-7a2-5p rn)
                     (leq oplen 32))
              (equal (read-rn oplen rn (mc-rfile sn))
                     (if (d5-7a2-5p rn)
```

```
                      (read-rn oplen rn (mc-rfile s))
                     (get-vlst oplen 0 rn '(2 3 4)
                               (movem-saved s 4 12 3))))))
     ((use (memmove-s1-s2))
      (disable memmove-s1p memmove-s2p)))

(prove-lemma memmove-s1-sn-mem-base-2 (rewrite)
     (let ((sn (stepn s (memmove-s1-sn-t0 i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s1p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (not (equal (remainder n 4) 0))
                     (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                     (disjoint x k str1 n_))
                (equal (read-mem x (mc-mem sn) k)
                       (read-mem x (mc-mem s) k))))
     ((use (memmove-s1-s2))
      (disable memmove-s1p memmove-s2p)))

; put together:  s1 --> s2.
(defn memmove-s1-sn-t (i* i str1 n lst1 str2 lst2 nt n_)
  (if (equal (sub1 nt) 0)
      (if (equal (remainder n 4) 0)
          10
        (memmove-s1-sn-t0 i* i str1 n lst1 str2 lst2 nt n_))
    (splus 3 (memmove-s1-sn-t (add 32 i* 4) (plus 4 i) str1 n
                              (movn-lst 4 lst1 lst2 i) str2 lst2
                              (sub1 nt) n_))))

(defn memmove-s1-sn-induct (s i* i lst1 lst2 nt)
  (if (equal (sub1 nt) 0)
      t
    (memmove-s1-sn-induct (stepn s 3) (add 32 i* 4) (plus 4 i)
                          (movn-lst 4 lst1 lst2 i) lst2 (sub1 nt))))

(prove-lemma memmove-s1p-info (rewrite)
     (implies (memmove-s1p s i* i str1 n lst1 str2 lst2 nt n_)
              (and (numberp nt) (not (equal nt 0)))))

(prove-lemma memmove-s1-sn (rewrite)
     (let ((sn (stepn s (memmove-s1-sn-t i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (memmove-s1p s i* i str1 n lst1 str2 lst2 nt n_)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) str1)
                     (mem-lst 1 str1 (mc-mem sn) n_
                              (memmove-1 lst1 lst2 i nt n))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))))))
     ((induct (memmove-s1-sn-induct s i* i lst1 lst2 nt))
      (disable memmove-s1p memmove-s1-sn-t0 movn-lst plus read-dn)))

(prove-lemma memmove-s1-sn-rfile (rewrite)
```

```
     (let ((sn (stepn s (memmove-s1-sn-t i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s1p s i* i str1 n lst1 str2 lst2 nt n_)
                     (d2-7a2-5p rn)
                     (leq oplen 32))
                (equal (read-rn oplen rn (mc-rfile sn))
                       (if (d5-7a2-5p rn)
                           (read-rn oplen rn (mc-rfile s))
                         (get-vlst oplen 0 rn '(2 3 4)
                                   (movem-saved s 4 12 3))))))
     ((induct (memmove-s1-sn-induct s i* i lst1 lst2 nt))
      (disable memmove-s1p memmove-s1-sn-t0 movn-lst plus)))

(prove-lemma memmove-s1-sn-mem (rewrite)
     (let ((sn (stepn s (memmove-s1-sn-t i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s1p s i* i str1 n lst1 str2 lst2 nt n_)
                     (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                     (disjoint x k str1 n_))
                (equal (read-mem x (mc-mem sn) k)
                       (read-mem x (mc-mem s) k))))
     ((induct (memmove-s1-sn-induct s i* i lst1 lst2 nt))
      (disable memmove-s1p memmove-s1-sn-t0 movn-lst plus)))

(disable memmove-s1p-info)

; from s0 to sn: s0 --> sn.
; base case 1: s0 --> sn.
(prove-lemma memmove-s0-sn-base-1 (rewrite)
     (let ((sn (stepn s 13)))
       (implies (and (memmove-s0p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (zerop n))
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) str1)
                     (mem-lst 1 str1 (mc-mem sn) n_
                              (put-nth (get-nth i lst2) i lst1))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))))))

(prove-lemma memmove-s0-sn-rfile-base-1 (rewrite)
     (let ((sn (stepn s 13)))
       (implies (and (memmove-s0p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (zerop n)
                     (d2-7a2-5p rn)
                     (leq oplen 32))
                (equal (read-rn oplen rn (mc-rfile sn))
                       (if (d5-7a2-5p rn)
                           (read-rn oplen rn (mc-rfile s))
                         (get-vlst oplen 0 rn '(2 3 4)
                                   (movem-saved s 4 12 3)))))))
```

```
(prove-lemma memmove-s0-sn-mem-base-1 (rewrite)
     (let ((sn (stepn s 13)))
       (implies (and (memmove-s0p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (zerop n)
                     (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                     (disjoint x k str1 n_))
                (equal (read-mem x (mc-mem sn) k)
                       (read-mem x (mc-mem s) k)))))

; base case 2: s0 --> s1 --> sn.
(defn memmove-s0-sn-t0 (i* i str1 n lst1 str2 lst2 nt n_)
  (splus 6 (memmove-s1-sn-t (add 32 i* 1) (add1 i) str1 n
                            (put-nth (get-nth i lst2) i lst1)
                             str2 lst2 (quotient n 4) n_)))

(prove-lemma memmove-s0-sn-base-2 (rewrite)
     (let ((sn (stepn s (memmove-s0-sn-t0 i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s0p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (not (lessp n 4)))
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) str1)
                     (mem-lst 1 str1 (mc-mem sn) n_
                              (memmove-1 (put-nth (get-nth i lst2) i lst1)
                                         lst2 (add1 i) (quotient n 4) n))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8)))))
     ((use (memmove-s0-s1))
      (disable memmove-s0p memmove-s1p read-dn memmove-1)))

(prove-lemma memmove-s0-sn-rfile-base-2 (rewrite)
     (let ((sn (stepn s (memmove-s0-sn-t0 i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s0p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (not (lessp n 4))
                     (d2-7a2-5p rn)
                     (leq oplen 32))
              (equal (read-rn oplen rn (mc-rfile sn))
                     (if (d5-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (get-vlst oplen 0 rn '(2 3 4)
                                 (movem-saved s 4 12 3)))))))
     ((use (memmove-s0-s1))
      (disable memmove-s0p memmove-s1p)))

(prove-lemma memmove-s0-sn-mem-base-2 (rewrite)
     (let ((sn (stepn s (memmove-s0-sn-t0 i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s0p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (not (lessp n 4))
```

```
                  (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                  (disjoint x k str1 n_))
             (equal (read-mem x (mc-mem sn) k)
                    (read-mem x (mc-mem s) k))))
     ((use (memmove-s0-s1))
      (disable memmove-s0p memmove-s1p)))

; base case 3: s0 --> s2 --> sn.
(defn memmove-s0-sn-t1 (i* i str1 n lst1 str2 lst2 nt n_)
  (splus 9 (memmove-s2-sn-t (add 32 i* 1) (add1 i) str1 n
                            (put-nth (get-nth i lst2) i lst1) str2
                            lst2 n n_)))

(prove-lemma memmove-s0-sn-base-3 (rewrite)
     (let ((sn (stepn s (memmove-s0-sn-t1 i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s0p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (lessp n 4)
                     (not (zerop n)))
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) str1)
                     (mem-lst 1 str1 (mc-mem sn) n_
                              (mmov1-lst (add1 i)
                                         (put-nth (get-nth i lst2) i lst1)
                                         lst2 n))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))))))
     ((use (memmove-s0-s2))
      (disable memmove-s0p memmove-s2p read-dn)))

(prove-lemma memmove-s0-sn-rfile-base-3 (rewrite)
     (let ((sn (stepn s (memmove-s0-sn-t1 i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s0p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (lessp n 4)
                     (not (zerop n))
                     (d2-7a2-5p rn)
                     (leq oplen 32))
              (equal (read-rn oplen rn (mc-rfile sn))
                     (if (d5-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (get-vlst oplen 0 rn '(2 3 4)
                                 (movem-saved s 4 12 3)))))))
     ((use (memmove-s0-s2))
      (disable memmove-s0p memmove-s2p)))

(prove-lemma memmove-s0-sn-mem-base-3 (rewrite)
     (let ((sn (stepn s (memmove-s0-sn-t1 i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s0p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (lessp n 4)
```

```
                    (not (zerop n))
                    (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                    (disjoint x k str1 n_))
               (equal (read-mem x (mc-mem sn) k)
                      (read-mem x (mc-mem s) k))))
     ((use (memmove-s0-s2))
      (disable memmove-s0p memmove-s2p)))

; put together: s0 --> sn.
(defn memmove-s0-sn-t (i* i str1 n lst1 str2 lst2 nt n_)
  (if (equal (sub1 nt) 0)
      (if (lessp n 4)
          (if (zerop n)
              13
            (memmove-s0-sn-t1 i* i str1 n lst1 str2 lst2 nt n_))
        (memmove-s0-sn-t0 i* i str1 n lst1 str2 lst2 nt n_))
    (splus 3 (memmove-s0-sn-t (add 32 i* 1) (add1 i) str1 n
                              (put-nth (get-nth i lst2) i lst1) str2
                              lst2 (sub1 nt) n_))))

(prove-lemma memmove-s0p-info (rewrite)
     (implies (memmove-s0p s i* i str1 n lst1 str2 lst2 nt n_)
              (and (numberp nt) (not (equal nt 0)))))

(prove-lemma memmove-s0-sn (rewrite)
     (let ((sn (stepn s (memmove-s0-sn-t i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (memmove-s0p s i* i str1 n lst1 str2 lst2 nt n_)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) str1)
                     (mem-lst 1 str1 (mc-mem sn) n_
                              (memmove-0 lst1 lst2 i nt n))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))))))
     ((induct (memmove-s2-sn-induct s i* i lst1 lst2 nt))
      (disable memmove-s0p read-dn memmove-s0-sn-t0 memmove-s0-sn-t1 memmove-1)
      (enable plus-add1-sub1 plus-add1-1)))

(prove-lemma memmove-s0-sn-rfile (rewrite)
     (let ((sn (stepn s (memmove-s0-sn-t i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s0p s i* i str1 n lst1 str2 lst2 nt n_)
                     (d2-7a2-5p rn)
                     (leq oplen 32))
                (equal (read-rn oplen rn (mc-rfile sn))
                       (if (d5-7a2-5p rn)
                           (read-rn oplen rn (mc-rfile s))
                         (get-vlst oplen 0 rn '(2 3 4)
                                   (movem-saved s 4 12 3)))))))
     ((induct (memmove-s2-sn-induct s i* i lst1 lst2 nt))
      (disable memmove-s0p memmove-s0-sn-t0 memmove-s0-sn-t1)))

(prove-lemma memmove-s0-sn-mem (rewrite)
```

```
     (let ((sn (stepn s (memmove-s0-sn-t i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s0p s i* i str1 n lst1 str2 lst2 nt n_)
                     (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                     (disjoint x k str1 n_))
                (equal (read-mem x (mc-mem sn) k)
                       (read-mem x (mc-mem s) k))))
     ((induct (memmove-s2-sn-induct s i* i lst1 lst2 nt))
      (disable memmove-s0p memmove-s0-sn-t0 memmove-s0-sn-t1)))

(disable memmove-s0p-info)

; from the initial state s to s3:  s --> s3.
(prove-lemma memmove-s-s3-1 (rewrite)
     (let ((x1 (plus (nat-to-uint str1) n))
           (x2 (plus (nat-to-uint str2) n)))
       (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                     (not (zerop n))
                     (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                     (not (lessp (nat-to-uint str1) (nat-to-uint str2)))
                     (not (equal (remainder x1 4) (remainder x2 4)))
                     (or (not (equal (remainder x1 4) 0))
                         (not (equal (remainder x2 4) 0))))
                (memmove-s3p (stepn s 26) (uint-to-nat n) n str1 0 lst1 str2
                             lst2 n n)))
     ((disable plus remainder quotient disjoint-leq-uint disjoint-leq1-uint)))

(prove-lemma memmove-s-s3-else-1 (rewrite)
     (let ((x1 (plus (nat-to-uint str1) n))
           (x2 (plus (nat-to-uint str2) n))
           (s3 (stepn s 26)))
       (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                     (not (zerop n))
                     (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                     (not (lessp (nat-to-uint str1) (nat-to-uint str2)))
                     (not (equal (remainder x1 4) (remainder x2 4)))
                     (or (not (equal (remainder x1 4) 0))
                         (not (equal (remainder x2 4) 0))))
                (and (equal (linked-rts-addr s3) (rts-addr s))
                     (equal (linked-a6 s3) (read-an 32 6 s))
                     (equal (read-rn 32 14 (mc-rfile s3))
                            (sub 32 4 (read-sp s)))
                     (equal (movem-saved s3 4 12 3)
                            (readm-rn 32 '(2 3 4) (mc-rfile s))))))
     ((disable plus remainder quotient disjoint-leq-uint disjoint-leq1-uint)))

(prove-lemma memmove-s-s3-rfile-1 (rewrite)
     (let ((x1 (plus (nat-to-uint str1) n))
           (x2 (plus (nat-to-uint str2) n)))
       (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                     (not (zerop n))
                     (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                     (not (lessp (nat-to-uint str1) (nat-to-uint str2)))
                     (not (equal (remainder x1 4) (remainder x2 4)))
                     (or (not (equal (remainder x1 4) 0))
```

```
                        (not (equal (remainder x2 4) 0)))
                   (d5-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 26)))
                     (read-rn oplen rn (mc-rfile s)))))
     ((disable plus remainder quotient disjoint-leq-uint disjoint-leq1-uint)))

(prove-lemma memmove-s-s3-mem-1 (rewrite)
     (let ((x1 (plus (nat-to-uint str1) n))
           (x2 (plus (nat-to-uint str2) n)))
       (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                     (not (zerop n))
                     (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                     (not (lessp (nat-to-uint str1) (nat-to-uint str2)))
                     (not (equal (remainder x1 4) (remainder x2 4)))
                     (or (not (equal (remainder x1 4) 0))
                         (not (equal (remainder x2 4) 0)))
                     (disjoint x k (sub 32 16 (read-sp s)) 32))
              (equal (read-mem x (mc-mem (stepn s 26)) k)
                     (read-mem x (mc-mem s) k)))))
     ((disable plus remainder quotient disjoint-leq-uint disjoint-leq1-uint)))

(prove-lemma memmove-s-s3-2 (rewrite)
     (let ((x1 (plus (nat-to-uint str1) n))
           (x2 (plus (nat-to-uint str2) n)))
       (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                     (not (zerop n))
                     (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                     (not (lessp (nat-to-uint str1) (nat-to-uint str2)))
                     (equal (remainder x1 4) (remainder x2 4))
                     (leq n 4)
                     (or (not (equal (remainder x1 4) 0))
                         (not (equal (remainder x2 4) 0))))
              (memmove-s3p (stepn s 29) (uint-to-nat n) n str1 0 lst1
                           str2 lst2 n n)))
     ((disable plus remainder quotient disjoint-leq-uint disjoint-leq1-uint)))

(prove-lemma memmove-s-s3-else-2 (rewrite)
     (let ((x1 (plus (nat-to-uint str1) n))
           (x2 (plus (nat-to-uint str2) n))
           (s3 (stepn s 29)))
       (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                     (not (zerop n))
                     (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                     (not (lessp (nat-to-uint str1) (nat-to-uint str2)))
                     (equal (remainder x1 4) (remainder x2 4))
                     (leq n 4)
                     (or (not (equal (remainder x1 4) 0))
                         (not (equal (remainder x2 4) 0))))
                (and (equal (linked-rts-addr s3) (rts-addr s))
                     (equal (linked-a6 s3) (read-an 32 6 s))
                     (equal (read-rn 32 14 (mc-rfile s3))
                            (sub 32 4 (read-sp s)))
                     (equal (movem-saved s3 4 12 3)
                            (readm-rn 32 '(2 3 4) (mc-rfile s)))))))
```

```
     ((disable plus remainder quotient disjoint-leq-uint disjoint-leq1-uint)))

(prove-lemma memmove-s-s3-rfile-2 (rewrite)
     (let ((x1 (plus (nat-to-uint str1) n))
           (x2 (plus (nat-to-uint str2) n)))
       (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                     (not (zerop n))
                     (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                     (not (lessp (nat-to-uint str1) (nat-to-uint str2)))
                     (equal (remainder x1 4) (remainder x2 4))
                     (leq n 4)
                     (or (not (equal (remainder x1 4) 0))
                         (not (equal (remainder x2 4) 0)))
                     (d5-7a2-5p rn))
                (equal (read-rn oplen rn (mc-rfile (stepn s 29)))
                       (read-rn oplen rn (mc-rfile s))))))
     ((disable plus remainder quotient disjoint-leq-uint disjoint-leq1-uint)))

(prove-lemma memmove-s-s3-mem-2 (rewrite)
     (let ((x1 (plus (nat-to-uint str1) n))
           (x2 (plus (nat-to-uint str2) n)))
       (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                     (not (zerop n))
                     (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                     (not (lessp (nat-to-uint str1) (nat-to-uint str2)))
                     (equal (remainder x1 4) (remainder x2 4))
                     (leq n 4)
                     (or (not (equal (remainder x1 4) 0))
                         (not (equal (remainder x2 4) 0)))
                     (disjoint x k (sub 32 16 (read-sp s)) 32))
                (equal (read-mem x (mc-mem (stepn s 29)) k)
                       (read-mem x (mc-mem s) k)))))
     ((disable plus remainder quotient disjoint-leq-uint disjoint-leq1-uint)))

(prove-lemma memmove-s-s3-3 (rewrite)
     (let ((x1 (plus (nat-to-uint str1) n))
           (x2 (plus (nat-to-uint str2) n))
           (nt (remainder (plus n (nat-to-uint str1)) 4)))
       (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                     (not (zerop n))
                     (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                     (not (lessp (nat-to-uint str1) (nat-to-uint str2)))
                     (equal (remainder x1 4) (remainder x2 4))
                     (lessp 4 n)
                     (or (not (equal (remainder x1 4) 0))
                         (not (equal (remainder x2 4) 0))))
                (memmove-s3p (stepn s 29) (uint-to-nat n) n str1
                             (difference n nt) lst1 str2 lst2 nt n)))
     ((disable plus remainder quotient disjoint-leq-uint disjoint-leq1-uint)))

(prove-lemma memmove-s-s3-else-3 (rewrite)
     (let ((x1 (plus (nat-to-uint str1) n))
           (x2 (plus (nat-to-uint str2) n))
           (s3 (stepn s 29)))
```

```
       (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                     (not (zerop n))
                     (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                     (not (lessp (nat-to-uint str1) (nat-to-uint str2)))
                     (equal (remainder x1 4) (remainder x2 4))
                     (lessp 4 n)
                     (or (not (equal (remainder x1 4) 0))
                         (not (equal (remainder x2 4) 0))))
                (and (equal (linked-rts-addr s3) (rts-addr s))
                     (equal (linked-a6 s3) (read-an 32 6 s))
                     (equal (read-rn 32 14 (mc-rfile s3))
                            (sub 32 4 (read-sp s)))
                     (equal (movem-saved s3 4 12 3)
                            (readm-rn 32 '(2 3 4) (mc-rfile s))))))
     ((disable plus remainder quotient disjoint-leq-uint disjoint-leq1-uint)))

(prove-lemma memmove-s-s3-rfile-3 (rewrite)
     (let ((x1 (plus (nat-to-uint str1) n))
           (x2 (plus (nat-to-uint str2) n)))
       (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                     (not (zerop n))
                     (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                     (not (lessp (nat-to-uint str1) (nat-to-uint str2)))
                     (equal (remainder x1 4) (remainder x2 4))
                     (lessp 4 n)
                     (or (not (equal (remainder x1 4) 0))
                         (not (equal (remainder x2 4) 0)))
                     (d5-7a2-5p rn))
                (equal (read-rn oplen rn (mc-rfile (stepn s 29)))
                       (read-rn oplen rn (mc-rfile s))))))
     ((disable plus remainder quotient disjoint-leq-uint disjoint-leq1-uint)))

(prove-lemma memmove-s-s3-mem-3 (rewrite)
     (let ((x1 (plus (nat-to-uint str1) n))
           (x2 (plus (nat-to-uint str2) n)))
       (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                     (not (zerop n))
                     (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                     (not (lessp (nat-to-uint str1) (nat-to-uint str2)))
                     (equal (remainder x1 4) (remainder x2 4))
                     (lessp 4 n)
                     (or (not (equal (remainder x1 4) 0))
                         (not (equal (remainder x2 4) 0)))
                     (disjoint x k (sub 32 16 (read-sp s)) 32))
                (equal (read-mem x (mc-mem (stepn s 29)) k)
                       (read-mem x (mc-mem s) k)))))
     ((disable plus remainder quotient disjoint-leq-uint disjoint-leq1-uint)))

; from the initial state s to s4:  s --> s4.
(prove-lemma memmove-s-s4 (rewrite)
     (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                   (not (zerop n))
                   (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                   (not (lessp (nat-to-uint str1) (nat-to-uint str2)))
```

```
                (equal (remainder (plus n (nat-to-uint str1)) 4) 0)
                (equal (remainder (plus n (nat-to-uint str2)) 4) 0)
                (not (lessp n 4)))
           (memmove-s4p (stepn s 21) (uint-to-nat n) n str1 n lst1
                        str2 lst2 (quotient n 4) n)))

(prove-lemma memmove-s-s4-else (rewrite)
     (let ((s4 (stepn s 21)))
       (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                     (not (zerop n))
                     (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                     (not (lessp (nat-to-uint str1) (nat-to-uint str2)))
                     (equal (remainder (plus n (nat-to-uint str1)) 4) 0)
                     (equal (remainder (plus n (nat-to-uint str2)) 4) 0)
                     (not (lessp n 4)))
                (and (equal (linked-rts-addr s4) (rts-addr s))
                     (equal (linked-a6 s4) (read-an 32 6 s))
                     (equal (read-rn 32 14 (mc-rfile s4))
                            (sub 32 4 (read-sp s)))
                     (equal (movem-saved s4 4 12 3)
                            (readm-rn 32 '(2 3 4) (mc-rfile s)))))))

(prove-lemma memmove-s-s4-rfile (rewrite)
     (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                   (not (zerop n))
                   (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                   (not (lessp (nat-to-uint str1) (nat-to-uint str2)))
                   (equal (remainder (plus n (nat-to-uint str1)) 4) 0)
                   (equal (remainder (plus n (nat-to-uint str2)) 4) 0)
                   (not (lessp n 4))
                   (d5-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 21)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memmove-s-s4-mem (rewrite)
     (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                   (not (zerop n))
                   (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                   (not (lessp (nat-to-uint str1) (nat-to-uint str2)))
                   (equal (remainder (plus n (nat-to-uint str1)) 4) 0)
                   (equal (remainder (plus n (nat-to-uint str2)) 4) 0)
                   (not (lessp n 4))
                   (disjoint x k (sub 32 16 (read-sp s)) 32))
              (equal (read-mem x (mc-mem (stepn s 21)) k)
                     (read-mem x (mc-mem s) k))))

; from s to s5.  s --> s5.
(prove-lemma memmove-s-s5 (rewrite)
     (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                   (not (zerop n))
                   (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                   (not (lessp (nat-to-uint str1) (nat-to-uint str2)))
                   (equal (remainder (plus n (nat-to-uint str1)) 4) 0)
                   (equal (remainder (plus n (nat-to-uint str2)) 4) 0)
```

```
                (lessp n 4))
           (memmove-s5p (stepn s 24) (uint-to-nat n) n str1 n lst1
                        str2 lst2 n n)))

(prove-lemma memmove-s-s5-else (rewrite)
     (let ((s0 (stepn s 24)))
       (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                     (not (zerop n))
                     (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                     (not (lessp (nat-to-uint str1) (nat-to-uint str2)))
                     (equal (remainder (plus n (nat-to-uint str1)) 4) 0)
                     (equal (remainder (plus n (nat-to-uint str2)) 4) 0)
                     (lessp n 4))
                (and (equal (linked-rts-addr s0) (rts-addr s))
                     (equal (linked-a6 s0) (read-an 32 6 s))
                     (equal (read-rn 32 14 (mc-rfile s0))
                            (sub 32 4 (read-sp s)))
                     (equal (movem-saved s0 4 12 3)
                            (readm-rn 32 '(2 3 4) (mc-rfile s)))))))

(prove-lemma memmove-s-s5-rfile (rewrite)
     (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                   (not (zerop n))
                   (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                   (not (lessp (nat-to-uint str1) (nat-to-uint str2)))
                   (equal (remainder (plus n (nat-to-uint str1)) 4) 0)
                   (equal (remainder (plus n (nat-to-uint str2)) 4) 0)
                   (lessp n 4)
                   (d5-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 24)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memmove-s-s5-mem (rewrite)
     (implies (and (memmove-statep s str1 n lst1 str2 lst2)
                   (not (zerop n))
                   (not (equal (nat-to-uint str1) (nat-to-uint str2)))
                   (not (lessp (nat-to-uint str1) (nat-to-uint str2)))
                   (equal (remainder (plus n (nat-to-uint str1)) 4) 0)
                   (equal (remainder (plus n (nat-to-uint str2)) 4) 0)
                   (lessp n 4)
                   (disjoint x k (sub 32 16 (read-sp s)) 32))
              (equal (read-mem x (mc-mem (stepn s 24)) k)
                     (read-mem x (mc-mem s) k))))

; s3 --> s4.
(prove-lemma memmove-s3-s4 ()
     (let ((s4 (stepn s 6)))
       (implies (and (memmove-s3p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (not (lessp n 4)))
                (memmove-s4p s4 (sub 32 1 i*) (sub1 i) str1 n
                             (put-nth (get-nth (sub1 i) lst2) (sub1 i) lst1)
                             str2 lst2 (quotient n 4) n_))))
```

```
(prove-lemma memmove-s3-s4-else (rewrite)
     (let ((s4 (stepn s 6)))
       (implies (and (memmove-s3p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (not (lessp n 4)))
                (and (equal (linked-rts-addr s4) (linked-rts-addr s))
                     (equal (linked-a6 s4) (linked-a6 s))
                     (equal (read-rn oplen 14 (mc-rfile s4))
                            (read-rn oplen 14 (mc-rfile s)))
                     (equal (movem-saved s4 4 12 3)
                            (movem-saved s 4 12 3))))))

(prove-lemma memmove-s3-s4-rfile-base (rewrite)
     (implies (and (memmove-s3p s i* i str1 n lst1 str2 lst2 nt n_)
                   (equal (sub1 nt) 0)
                   (not (lessp n 4))
                   (d5-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 6)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memmove-s3-s4-mem-base (rewrite)
     (implies (and (memmove-s3p s i* i str1 n lst1 str2 lst2 nt n_)
                   (equal (sub1 nt) 0)
                   (not (lessp n 4))
                   (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                   (disjoint x k str1 n_))
              (equal (read-mem x (mc-mem (stepn s 6)) k)
                     (read-mem x (mc-mem s) k))))

; s3 --> s5.
(prove-lemma memmove-s3-s5 ()
     (let ((s5 (stepn s 9)))
       (implies (and (memmove-s3p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (lessp n 4)
                     (not (zerop n)))
                (memmove-s5p s5 (sub 32 1 i*) (sub1 i) str1 n
                             (put-nth (get-nth (sub1 i) lst2) (sub1 i) lst1)
                             str2 lst2 n n_)))))

(prove-lemma memmove-s3-s5-else (rewrite)
     (let ((s5 (stepn s 9)))
       (implies (and (memmove-s3p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (lessp n 4)
                     (not (zerop n)))
                (and (equal (linked-rts-addr s5) (linked-rts-addr s))
                     (equal (linked-a6 s5) (linked-a6 s))
                     (equal (read-rn oplen 14 (mc-rfile s5))
                            (read-rn oplen 14 (mc-rfile s)))
                     (equal (movem-saved s5 4 12 3)
                            (movem-saved s 4 12 3))))))

(prove-lemma memmove-s3-s5-rfile (rewrite)
```

```
     (implies (and (memmove-s3p s i* i str1 n lst1 str2 lst2 nt n_)
                   (equal (sub1 nt) 0)
                   (lessp n 4)
                   (not (zerop n))
                   (d5-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 9)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memmove-s3-s5-mem (rewrite)
     (implies (and (memmove-s3p s i* i str1 n lst1 str2 lst2 nt n_)
                   (equal (sub1 nt) 0)
                   (lessp n 4)
                   (not (zerop n))
                   (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                   (disjoint x k str1 n_))
              (equal (read-mem x (mc-mem (stepn s 9)) k)
                     (read-mem x (mc-mem s) k))))

; s3 --> s3.
(prove-lemma memmove-s3-s3 (rewrite)
     (let ((s3 (stepn s 3)))
       (implies (and (memmove-s3p s i* i str1 n lst1 str2 lst2 nt n_)
                     (not (equal (sub1 nt) 0)))
                (and (memmove-s3p s3 (sub 32 1 i*) (sub1 i) str1 n
                                  (put-nth (get-nth (sub1 i) lst2)
                                           (sub1 i) lst1)
                                  str2 lst2 (sub1 nt) n_)
                     (equal (linked-rts-addr s3) (linked-rts-addr s))
                     (equal (linked-a6 s3) (linked-a6 s))
                     (equal (read-rn oplen 14 (mc-rfile s3))
                            (read-rn oplen 14 (mc-rfile s)))
                     (equal (movem-saved s3 4 12 3)
                            (movem-saved s 4 12 3))))))

(prove-lemma memmove-s3-s3-rfile (rewrite)
     (implies (and (memmove-s3p s i* i str1 n lst1 str2 lst2 nt n_)
                   (not (equal (sub1 nt) 0))
                   (d5-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 3)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memmove-s3-s3-mem (rewrite)
     (implies (and (memmove-s3p s i* i str1 n lst1 str2 lst2 nt n_)
                   (not (equal (sub1 nt) 0))
                   (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                   (disjoint x k str1 n_))
              (equal (read-mem x (mc-mem (stepn s 3)) k)
                     (read-mem x (mc-mem s) k))))

; s4 --> s5.
(prove-lemma memmove-s4-s5 ()
     (let ((s5 (stepn s 6)))
       (implies (and (memmove-s4p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
```

```
                    (not (equal (remainder n 4) 0)))
               (memmove-s5p s5 (sub 32 4 i*) (difference i 4) str1 n
                            (movn-lst 4 lst1 lst2 (difference i 4)) str2
                            lst2 (remainder n 4) n_)))
     ((disable put-get-vals-is-cpy)))

(prove-lemma memmove-s4-s5-else (rewrite)
     (let ((s5 (stepn s 6)))
       (implies (and (memmove-s4p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (not (equal (remainder n 4) 0)))
                (and (equal (linked-rts-addr s5) (linked-rts-addr s))
                     (equal (linked-a6 s5) (linked-a6 s))
                     (equal (read-rn oplen 14 (mc-rfile s5))
                            (read-rn oplen 14 (mc-rfile s)))
                     (equal (movem-saved s5 4 12 3)
                            (movem-saved s 4 12 3))))))

(prove-lemma memmove-s4-s5-rfile (rewrite)
     (implies (and (memmove-s4p s i* i str1 n lst1 str2 lst2 nt n_)
                   (equal (sub1 nt) 0)
                   (not (equal (remainder n 4) 0))
                   (d5-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 6)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memmove-s4-s5-mem (rewrite)
     (implies (and (memmove-s4p s i* i str1 n lst1 str2 lst2 nt n_)
                   (equal (sub1 nt) 0)
                   (not (equal (remainder n 4) 0))
                   (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                   (disjoint x k str1 n_))
              (equal (read-mem x (mc-mem (stepn s 6)) k)
                     (read-mem x (mc-mem s) k))))

; s4 --> s4.
(prove-lemma memmove-s4-s4 (rewrite)
     (let ((s4 (stepn s 3)))
       (implies (and (memmove-s4p s i* i str1 n lst1 str2 lst2 nt n_)
                     (not (equal (sub1 nt) 0)))
                (and (memmove-s4p s4 (sub 32 4 i*) (difference i 4) str1 n
                                  (movn-lst 4 lst1 lst2 (difference i 4))
                                  str2 lst2 (sub1 nt) n_)
                     (equal (linked-rts-addr s4) (linked-rts-addr s))
                     (equal (linked-a6 s4) (linked-a6 s))
                     (equal (read-rn oplen 14 (mc-rfile s4))
                            (read-rn oplen 14 (mc-rfile s)))
                     (equal (movem-saved s4 4 12 3)
                            (movem-saved s 4 12 3)))))
     ((disable put-get-vals-is-cpy)))

(prove-lemma memmove-s4-s4-rfile (rewrite)
     (implies (and (memmove-s4p s i* i str1 n lst1 str2 lst2 nt n_)
                   (not (equal (sub1 nt) 0))
```

```
                (d5-7a2-5p rn))
           (equal (read-rn oplen rn (mc-rfile (stepn s 3)))
                  (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memmove-s4-s4-mem (rewrite)
     (implies (and (memmove-s4p s i* i str1 n lst1 str2 lst2 nt n_)
                   (not (equal (sub1 nt) 0))
                   (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                   (disjoint x k str1 n_))
              (equal (read-mem x (mc-mem (stepn s 3)) k)
                     (read-mem x (mc-mem s) k)))))

; from s5 to exit:  s5 --> sn.
; base case: s5 --> sn.
(prove-lemma memmove-s5-sn-base (rewrite)
     (let ((sn (stepn s 7)))
       (implies (and (memmove-s5p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0))
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) str1)
                     (mem-lst 1 str1 (mc-mem sn) n_
                              (put-nth (get-nth (sub1 i) lst2) (sub1 i) lst1))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))))))

(prove-lemma memmove-s5-sn-rfile-base (rewrite)
     (implies (and (memmove-s5p s i* i str1 n lst1 str2 lst2 nt n_)
                   (equal (sub1 nt) 0)
                   (d2-7a2-5p rn)
                   (leq oplen 32))
              (equal (read-rn oplen rn (mc-rfile (stepn s 7)))
                     (if (d5-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (get-vlst oplen 0 rn '(2 3 4)
                                 (movem-saved s 4 12 3))))))

(prove-lemma memmove-s5-sn-mem-base (rewrite)
     (implies (and (memmove-s5p s i* i str1 n lst1 str2 lst2 nt n_)
                   (equal (sub1 nt) 0)
                   (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                   (disjoint x k str1 n_))
              (equal (read-mem x (mc-mem (stepn s 7)) k)
                     (read-mem x (mc-mem s) k))))

; induction case:  s5 --> s5.
(prove-lemma memmove-s5-s5 (rewrite)
     (let ((s5 (stepn s 3)))
       (implies (and (memmove-s5p s i* i str1 n lst1 str2 lst2 nt n_)
                     (not (equal (sub1 nt) 0)))
                (and (memmove-s5p s5 (sub 32 1 i*) (sub1 i) str1 n
                                  (put-nth (get-nth (sub1 i) lst2) (sub1 i)
```

```
                                        lst1)
                                   str2 lst2 (sub1 nt) n_)
                     (equal (linked-rts-addr s5) (linked-rts-addr s))
                     (equal (linked-a6 s5) (linked-a6 s))
                     (equal (read-rn oplen 14 (mc-rfile s5))
                            (read-rn oplen 14 (mc-rfile s)))
                     (equal (movem-saved s5 4 12 3)
                            (movem-saved s 4 12 3))))))

(prove-lemma memmove-s5-s5-rfile (rewrite)
     (implies (and (memmove-s5p s i* i str1 n lst1 str2 lst2 nt n_)
                   (not (equal (sub1 nt) 0))
                   (d5-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 3)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memmove-s5-s5-mem (rewrite)
     (implies (and (memmove-s5p s i* i str1 n lst1 str2 lst2 nt n_)
                   (not (equal (sub1 nt) 0))
                   (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                   (disjoint x k str1 n_))
              (equal (read-mem x (mc-mem (stepn s 3)) k)
                     (read-mem x (mc-mem s) k)))))

; put together: s5 --> sn.
(defn memmove-s5-sn-t (i* i str1 n lst1 str2 lst2 nt n_)
  (if (equal (sub1 nt) 0)
      7
    (splus 3 (memmove-s5-sn-t (sub 32 1 i*) (sub1 i) str1 n
                              (put-nth (get-nth (sub1 i) lst2) (sub1 i)
                                       lst1)
                              str2 lst2 (sub1 nt) n_))))

(defn memmove-s5-sn-induct (s i* i lst1 lst2 nt)
  (if (equal (sub1 nt) 0)
      t
    (memmove-s5-sn-induct (stepn s 3) (sub 32 1 i*) (sub1 i)
                          (put-nth (get-nth (sub1 i) lst2) (sub1 i) lst1)
                          lst2 (sub1 nt))))

(prove-lemma memmove-s5p-info (rewrite)
     (implies (memmove-s5p s i* i str1 n lst1 str2 lst2 nt n_)
              (and (numberp nt) (not (equal nt 0)))))

(prove-lemma memmove-s5-sn (rewrite)
     (let ((sn (stepn s (memmove-s5-sn-t i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (memmove-s5p s i* i str1 n lst1 str2 lst2 nt n_)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) str1)
                     (mem-lst 1 str1 (mc-mem sn) n_
                              (mmov1-lst1 i lst1 lst2 nt))
                     (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
```

```
                          (add 32 (read-an 32 6 s) 8)))))
       ((induct (memmove-s5-sn-induct s i* i lst1 lst2 nt))
        (disable memmove-s5p read-dn sub-neg)))

(prove-lemma memmove-s5-sn-rfile (rewrite)
     (let ((sn (stepn s (memmove-s5-sn-t i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s5p s i* i str1 n lst1 str2 lst2 nt n_)
                     (d2-7a2-5p rn)
                     (leq oplen 32))
                (equal (read-rn oplen rn (mc-rfile sn))
                       (if (d5-7a2-5p rn)
                           (read-rn oplen rn (mc-rfile s))
                         (get-vlst oplen 0 rn '(2 3 4)
                                   (movem-saved s 4 12 3)))))))
     ((induct (memmove-s5-sn-induct s i* i lst1 lst2 nt))
      (disable memmove-s5p sub-neg)))

(prove-lemma memmove-s5-sn-mem (rewrite)
     (let ((sn (stepn s (memmove-s5-sn-t i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s5p s i* i str1 n lst1 str2 lst2 nt n_)
                     (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                     (disjoint x k str1 n_))
              (equal (read-mem x (mc-mem sn) k)
                     (read-mem x (mc-mem s) k)))))
     ((induct (memmove-s5-sn-induct s i* i lst1 lst2 nt))
      (disable memmove-s5p sub-neg)))

(disable memmove-s5p-info)

; from s4 to sn: s4 --> sn.
; base case 1: s4 --> sn.
(prove-lemma memmove-s4-sn-base-1 (rewrite)
     (let ((sn (stepn s 10)))
       (implies (and (memmove-s4p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (equal (remainder n 4) 0))
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) str1)
                     (mem-lst 1 str1 (mc-mem sn) n_
                              (movn-lst 4 lst1 lst2 (difference i 4)))
                     (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))))))
     ((disable put-get-vals-is-cpy)))

(prove-lemma memmove-s4-sn-rfile-base-1 (rewrite)
     (implies (and (memmove-s4p s i* i str1 n lst1 str2 lst2 nt n_)
                   (equal (sub1 nt) 0)
                   (equal (remainder n 4) 0)
                   (d2-7a2-5p rn)
                   (leq oplen 32))
              (equal (read-rn oplen rn (mc-rfile (stepn s 10)))
                     (if (d5-7a2-5p rn)
```

```
                      (read-rn oplen rn (mc-rfile s))
                     (get-vlst oplen 0 rn '(2 3 4)
                               (movem-saved s 4 12 3))))))

(prove-lemma memmove-s4-sn-mem-base-1 (rewrite)
     (implies (and (memmove-s4p s i* i str1 n lst1 str2 lst2 nt n_)
                   (equal (sub1 nt) 0)
                   (equal (remainder n 4) 0)
                   (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                   (disjoint x k str1 n_))
              (equal (read-mem x (mc-mem (stepn s 10)) k)
                     (read-mem x (mc-mem s) k)))))

; base case 2: s4 --> s5 --> sn.
(defn memmove-s4-sn-t0 (i* i str1 n lst1 str2 lst2 nt n_)
  (splus 6 (memmove-s5-sn-t (sub 32 4 i*) (difference i 4) str1 n
                            (movn-lst 4 lst1 lst2 (difference i 4)) str2
                            lst2 (remainder n 4) n_)))

(prove-lemma memmove-s4-sn-base-2 (rewrite)
     (let ((sn (stepn s (memmove-s4-sn-t0 i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s4p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (not (equal (remainder n 4) 0)))
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) str1)
                     (mem-lst 1 str1 (mc-mem sn) n_
                              (mmov1-lst1 (difference i 4)
                                          (movn-lst 4 lst1 lst2
                                                    (difference i 4))
                                          lst2 (remainder n 4)))
                     (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))))))
     ((use (memmove-s4-s5))
      (disable memmove-s4p memmove-s5p read-dn movn-lst sub-neg)))

(prove-lemma memmove-s4-sn-rfile-base-2 (rewrite)
     (let ((sn (stepn s (memmove-s4-sn-t0 i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s4p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (not (equal (remainder n 4) 0))
                     (d2-7a2-5p rn)
                     (leq oplen 32))
              (equal (read-rn oplen rn (mc-rfile sn))
                     (if (d5-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (get-vlst oplen 0 rn '(2 3 4)
                                 (movem-saved s 4 12 3))))))
     ((use (memmove-s4-s5))
      (disable memmove-s4p memmove-s5p sub-neg)))

(prove-lemma memmove-s4-sn-mem-base-2 (rewrite)
```

```
     (let ((sn (stepn s (memmove-s4-sn-t0 i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s4p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (not (equal (remainder n 4) 0))
                     (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                     (disjoint x k str1 n_))
                (equal (read-mem x (mc-mem sn) k)
                       (read-mem x (mc-mem s) k))))
     ((use (memmove-s4-s5))
      (disable memmove-s4p memmove-s5p sub-neg)))

; put together:  s4 --> sn.
(defn memmove-s4-sn-t (i* i str1 n lst1 str2 lst2 nt n_)
  (if (equal (sub1 nt) 0)
      (if (equal (remainder n 4) 0)
          10
        (memmove-s4-sn-t0 i* i str1 n lst1 str2 lst2 nt n_))
    (splus 3 (memmove-s4-sn-t (sub 32 4 i*) (difference i 4) str1 n
                              (movn-lst 4 lst1 lst2 (difference i 4)) str2
                              lst2 (sub1 nt) n_))))

(defn memmove-s4-sn-induct (s i* i lst1 lst2 nt)
  (if (equal (sub1 nt) 0)
      t
    (memmove-s4-sn-induct (stepn s 3) (sub 32 4 i*) (difference i 4)
                          (movn-lst 4 lst1 lst2 (difference i 4)) lst2
                          (sub1 nt))))

(prove-lemma memmove-s4p-info (rewrite)
     (implies (memmove-s4p s i* i str1 n lst1 str2 lst2 nt n_)
              (and (numberp nt) (not (equal nt 0)))))

(prove-lemma memmove-s4-sn (rewrite)
     (let ((sn (stepn s (memmove-s4-sn-t i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (memmove-s4p s i* i str1 n lst1 str2 lst2 nt n_)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) str1)
                     (mem-lst 1 str1 (mc-mem sn) n_
                              (memmove-4 lst1 lst2 i nt n))
                     (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8)))))
     ((induct (memmove-s4-sn-induct s i* i lst1 lst2 nt))
      (disable memmove-s4p memmove-s4-sn-t0 read-dn movn-lst sub-neg)))

(prove-lemma memmove-s4-sn-rfile (rewrite)
     (let ((sn (stepn s (memmove-s4-sn-t i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s4p s i* i str1 n lst1 str2 lst2 nt n_)
                     (d2-7a2-5p rn)
                     (leq oplen 32))
                (equal (read-rn oplen rn (mc-rfile sn))
                       (if (d5-7a2-5p rn)
                           (read-rn oplen rn (mc-rfile s))
```

```
                        (get-vlst oplen 0 rn '(2 3 4)
                                 (movem-saved s 4 12 3))))))
     ((induct (memmove-s4-sn-induct s i* i lst1 lst2 nt))
      (disable memmove-s4p memmove-s4-sn-t0 movn-lst sub-neg)))

(prove-lemma memmove-s4-sn-mem (rewrite)
     (let ((sn (stepn s (memmove-s4-sn-t i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s4p s i* i str1 n lst1 str2 lst2 nt n_)
                     (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                     (disjoint x k str1 n_))
                (equal (read-mem x (mc-mem sn) k)
                       (read-mem x (mc-mem s) k))))
     ((induct (memmove-s4-sn-induct s i* i lst1 lst2 nt))
      (disable memmove-s4p memmove-s4-sn-t0 movn-lst sub-neg)))

(disable memmove-s4p-info)

; from s3 to sn: s3 --> sn.
; base case 1: s3 --> sn.
(prove-lemma memmove-s3-sn-base-1 (rewrite)
     (let ((sn (stepn s 13)))
       (implies (and (memmove-s3p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (zerop n))
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) str1)
                     (mem-lst 1 str1 (mc-mem sn) n_
                              (put-nth (get-nth (sub1 i) lst2) (sub1 i) lst1))
                     (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))))))

(prove-lemma memmove-s3-sn-rfile-base-1 (rewrite)
     (let ((sn (stepn s 13)))
       (implies (and (memmove-s3p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (zerop n)
                     (d2-7a2-5p rn)
                     (leq oplen 32))
                (equal (read-rn oplen rn (mc-rfile sn))
                       (if (d5-7a2-5p rn)
                           (read-rn oplen rn (mc-rfile s))
                         (get-vlst oplen 0 rn '(2 3 4)
                                   (movem-saved s 4 12 3)))))))

(prove-lemma memmove-s3-sn-mem-base-1 (rewrite)
     (let ((sn (stepn s 13)))
       (implies (and (memmove-s3p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (zerop n)
                     (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                     (disjoint x k str1 n_))
                (equal (read-mem x (mc-mem sn) k)
```

```
                    (read-mem x (mc-mem s) k)))))

; base case 2: s3 --> s4 --> sn.
(defn memmove-s3-sn-t0 (i* i str1 n lst1 str2 lst2 nt n_)
  (splus 6 (memmove-s4-sn-t (sub 32 1 i*) (sub1 i) str1 n
                            (put-nth (get-nth (sub1 i) lst2) (sub1 i) lst1)
                            str2 lst2 (quotient n 4) n_)))

(prove-lemma memmove-s3-sn-base-2 (rewrite)
     (let ((sn (stepn s (memmove-s3-sn-t0 i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s3p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (not (lessp n 4)))
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) str1)
                     (mem-lst 1 str1 (mc-mem sn) n_
                              (memmove-4 (put-nth (get-nth (sub1 i) lst2)
                                                  (sub1 i) lst1)
                                         lst2 (sub1 i) (quotient n 4) n))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))))))
     ((use (memmove-s3-s4))
      (disable memmove-s3p memmove-s4p read-dn memmove-4 sub-neg)))

(prove-lemma memmove-s3-sn-rfile-base-2 (rewrite)
     (let ((sn (stepn s (memmove-s3-sn-t0 i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s3p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (not (lessp n 4))
                     (d2-7a2-5p rn)
                     (leq oplen 32))
                (equal (read-rn oplen rn (mc-rfile sn))
                       (if (d5-7a2-5p rn)
                           (read-rn oplen rn (mc-rfile s))
                         (get-vlst oplen 0 rn '(2 3 4)
                                   (movem-saved s 4 12 3)))))))
     ((use (memmove-s3-s4))
      (disable memmove-s3p memmove-s4p sub-neg)))

(prove-lemma memmove-s3-sn-mem-base-2 (rewrite)
     (let ((sn (stepn s (memmove-s3-sn-t0 i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s3p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (not (lessp n 4))
                     (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                     (disjoint x k str1 n_))
                (equal (read-mem x (mc-mem sn) k)
                       (read-mem x (mc-mem s) k)))))
     ((use (memmove-s3-s4))
      (disable memmove-s3p memmove-s4p sub-neg)))
```

```
; base case 3: s3 --> s5 --> sn.
(defn memmove-s3-sn-t1 (i* i str1 n lst1 str2 lst2 nt n_)
  (splus 9 (memmove-s5-sn-t (sub 32 1 i*) (sub1 i) str1 n
                            (put-nth (get-nth (sub1 i) lst2) (sub1 i) lst1)
                            str2 lst2 n n_)))

(prove-lemma memmove-s3-sn-base-3 (rewrite)
     (let ((sn (stepn s (memmove-s3-sn-t1 i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s3p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (lessp n 4)
                     (not (zerop n)))
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) str1)
                     (mem-lst 1 str1 (mc-mem sn) n_
                              (mmov1-lst1 (sub1 i)
                                          (put-nth (get-nth (sub1 i) lst2)
                                                   (sub1 i) lst1)
                                          lst2 n))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))))))
     ((use (memmove-s3-s5))
      (disable memmove-s3p memmove-s5p read-dn sub-neg)))

(prove-lemma memmove-s3-sn-rfile-base-3 (rewrite)
     (let ((sn (stepn s (memmove-s3-sn-t1 i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s3p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (lessp n 4)
                     (not (zerop n))
                     (d2-7a2-5p rn)
                     (leq oplen 32))
                (equal (read-rn oplen rn (mc-rfile sn))
                       (if (d5-7a2-5p rn)
                           (read-rn oplen rn (mc-rfile s))
                         (get-vlst oplen 0 rn '(2 3 4)
                                   (movem-saved s 4 12 3)))))))
     ((use (memmove-s3-s5))
      (disable memmove-s3p memmove-s5p sub-neg)))

(prove-lemma memmove-s3-sn-mem-base-3 (rewrite)
     (let ((sn (stepn s (memmove-s3-sn-t1 i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s3p s i* i str1 n lst1 str2 lst2 nt n_)
                     (equal (sub1 nt) 0)
                     (lessp n 4)
                     (not (zerop n))
                     (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                     (disjoint x k str1 n_))
                (equal (read-mem x (mc-mem sn) k)
                       (read-mem x (mc-mem s) k)))))
     ((use (memmove-s3-s5))
```

```
      (disable memmove-s3p memmove-s5p sub-neg)))

; put together: s3 --> sn.
(defn memmove-s3-sn-t (i* i str1 n lst1 str2 lst2 nt n_)
  (if (equal (sub1 nt) 0)
      (if (lessp n 4)
          (if (zerop n)
              13
            (memmove-s3-sn-t1 i* i str1 n lst1 str2 lst2 nt n_))
        (memmove-s3-sn-t0 i* i str1 n lst1 str2 lst2 nt n_))
    (splus 3 (memmove-s3-sn-t (sub 32 1 i*) (sub1 i) str1 n
                              (put-nth (get-nth (sub1 i) lst2) (sub1 i) lst1)
                              str2 lst2 (sub1 nt) n_))))

(prove-lemma memmove-s3p-info (rewrite)
     (implies (memmove-s3p s i* i str1 n lst1 str2 lst2 nt n_)
              (and (numberp nt) (not (equal nt 0)))))

(prove-lemma memmove-s3-sn (rewrite)
     (let ((sn (stepn s (memmove-s3-sn-t i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (memmove-s3p s i* i str1 n lst1 str2 lst2 nt n_)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) str1)
                     (mem-lst 1 str1 (mc-mem sn) n_
                              (memmove-3 lst1 lst2 i nt n))
                     (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8)))))
     ((induct (memmove-s5-sn-induct s i* i lst1 lst2 nt))
      (disable memmove-s3p memmove-s3-sn-t0 memmove-s3-sn-t1 read-dn
               memmove-4 sub-neg)))

(prove-lemma memmove-s3-sn-rfile (rewrite)
     (let ((sn (stepn s (memmove-s3-sn-t i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s3p s i* i str1 n lst1 str2 lst2 nt n_)
                     (d2-7a2-5p rn)
                     (leq oplen 32))
                (equal (read-rn oplen rn (mc-rfile sn))
                       (if (d5-7a2-5p rn)
                           (read-rn oplen rn (mc-rfile s))
                         (get-vlst oplen 0 rn '(2 3 4)
                                   (movem-saved s 4 12 3)))))))
     ((induct (memmove-s5-sn-induct s i* i lst1 lst2 nt))
      (disable memmove-s3p memmove-s3-sn-t0 memmove-s3-sn-t1 sub-neg)))

(prove-lemma memmove-s3-sn-mem (rewrite)
     (let ((sn (stepn s (memmove-s3-sn-t i* i str1 n lst1 str2 lst2 nt n_))))
       (implies (and (memmove-s3p s i* i str1 n lst1 str2 lst2 nt n_)
                     (disjoint x k (sub 32 12 (read-an 32 6 s)) 32)
                     (disjoint x k str1 n_))
                (equal (read-mem x (mc-mem sn) k)
                       (read-mem x (mc-mem s) k)))))
     ((induct (memmove-s5-sn-induct s i* i lst1 lst2 nt))
```

```
      (disable memmove-s3p memmove-s3-sn-t0 memmove-s3-sn-t1 sub-neg)))

(disable memmove-s3p-info)

; the correctness of memmove.
(defn memmove-t (str1 n lst1 str2 lst2)
  (if (zerop n)
      11
    (let ((x1 (nat-to-uint str1))
          (x2 (nat-to-uint str2)))
      (if (equal x1 x2)
          13
        (if (lessp x1 x2)
            (if (and (equal (remainder x1 4) 0)
                     (equal (remainder x2 4) 0))
                (if (lessp n 4)
                    (splus 22 (memmove-s2-sn-t 0 0 str1 n lst1 str2 lst2 n n))
                  (splus 19 (memmove-s1-sn-t 0 0 str1 n lst1 str2 lst2
                                             (quotient n 4) n)))
              (if (equal (remainder x1 4) (remainder x2 4))
                  (if (lessp 3 n)
                      (splus 29 (memmove-s0-sn-t
                                 0 0 str1
                                 (difference (plus n (remainder x1 4)) 4)
                                 lst1 str2 lst2
                                 (difference 4 (remainder x1 4)) n))
                    (splus 27 (memmove-s0-sn-t 0 0 str1 0 lst1 str2 lst2 n n)))
                (splus 24 (memmove-s0-sn-t 0 0 str1 0 lst1 str2 lst2 n n))))
          (let ((y1 (plus n x1))
                (y2 (plus n x2)))
            (if (and (equal (remainder y1 4) 0)
                     (equal (remainder y2 4) 0))
                (if (lessp n 4)
                    (splus 24 (memmove-s5-sn-t (uint-to-nat n) n str1 n lst1
                                               str2 lst2 n n))
                  (splus 21 (memmove-s4-sn-t (uint-to-nat n) n str1 n lst1
                                             str2 lst2 (quotient n 4) n)))
              (if (equal (remainder y1 4) (remainder y2 4))
                  (if (lessp 4 n)
                      (splus 29 (memmove-s3-sn-t
                                 (uint-to-nat n) n str1
                                 (difference n (remainder y1 4))
                                 lst1 str2 lst2 (remainder y1 4) n))
                    (splus 29 (memmove-s3-sn-t (uint-to-nat n) n str1 0 lst1
                                               str2 lst2 n n)))
                (splus 26 (memmove-s3-sn-t (uint-to-nat n) n str1 0 lst1 str2
                                           lst2 n n)))))))))))

(prove-lemma memmove-correctness (rewrite)
     (let ((sn (stepn s (memmove-t str1 n lst1 str2 lst2))))
       (implies (memmove-statep s str1 n lst1 str2 lst2)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile sn))
```

```
                        (read-rn 32 14 (mc-rfile s)))
                  (equal (read-rn 32 15 (mc-rfile sn))
                         (add 32 (read-sp s) 4))
                  (implies (and (d2-7a2-5p rn)
                                (leq oplen 32))
                           (equal (read-rn oplen rn (mc-rfile sn))
                                  (read-rn oplen rn (mc-rfile s))))
                  (implies (and (disjoint x k (sub 32 16 (read-sp s)) 32)
                                (disjoint x k str1 n))
                           (equal (read-mem x (mc-mem sn) k)
                                  (read-mem x (mc-mem s) k)))
                  (equal (read-dn 32 0 sn) str1)
                  (mem-lst 1 str1 (mc-mem sn) n
                           (memmove str1 str2 n lst1 lst2)))))
     ((disable memmove-statep memmove-s0p memmove-s1p memmove-s2p
               memmove-s3p memmove-s4p memmove-s5p memmove-s2-sn-t
               memmove-s1-sn-t memmove-s0-sn-t memmove-s5-sn-t
               memmove-s4-sn-t memmove-s3-sn-t memmove-0 memmove-1
               memmove-3 memmove-4 read-dn rts-addr linked-rts-addr
               linked-a6 remainder plus)))

(disable memmove-t)

; some properties of memmove.
; see file cstring.events.
```

C.14 The memset Function

```
;          Proof of the Correctness of the MEMSET Function
#|
This is part of our effort to verify the Berkeley string library.  The
Berkeley string library is widely used as part of the Berkeley Unix OS.

This is the source code of memset function in the Berkeley string library.

/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 */
void *
memset(dst, c, n)
        void *dst;
        register int c;
        register size_t n;
{

        if (n != 0) {
                register char *d = dst;

                do
                        *d++ = c;
                while (--n != 0);
```

```
        }
        return (dst);
}
```

The MC68020 assembly code of the C function memset on SUN-3 is given as follows. This binary is generated by "gcc -O".

```
0x29d0 <memset>:        linkw fp,#0
0x29d4 <memset+4>:      movel d2,sp@-
0x29d6 <memset+6>:      movel fp@(8),d2
0x29da <memset+10>:     movel fp@(16),d0
0x29de <memset+14>:     beq 0x29ec <memset+28>
0x29e0 <memset+16>:     moveal d2,a0
0x29e2 <memset+18>:     moveb fp@(15),d1
0x29e6 <memset+22>:     moveb d1,a0@+
0x29e8 <memset+24>:     subl #1,d0
0x29ea <memset+26>:     bne 0x29e6 <memset+22>
0x29ec <memset+28>:     movel d2,d0
0x29ee <memset+30>:     movel fp@(-4),d2
0x29f2 <memset+34>:     unlk fp
0x29f4 <memset+36>:     rts
```

The machine code of the above program is:

```
<memset>:     0x4e56  0x0000  0x2f02  0x242e  0x0008  0x202e  0x0010  0x670c
<memset+16>:  0x2042  0x122e  0x000f  0x10c1  0x5380  0x66fa  0x2002  0x242e
<memset+32>:  0xfffc  0x4e5e  0x4e75
```

```
'(78    86      0       0       47      2       36      46
  0     8       32      46      0       16      103     12
  32    66      18      46      0       15      16      193
  83    128     102     250     32      2       36      46
  255   252     78      94      78      117)
|#

; in the logic, the above program is defined by (memset-code).
(defn memset-code ()
  '(78    86      0       0       47      2       36      46
    0     8       32      46      0       16      103     12
    32    66      18      46      0       15      16      193
    83    128     102     250     32      2       36      46
    255   252     78      94      78      117))

; the computation time of the program.
(defn memset-t1 (n)
  (if (equal (sub1 n) 0)
      7
    (splus 3 (memset-t1 (sub1 n)))))

(defn memset-t (n)
  (if (equal n 0)
      9
    (splus 7 (memset-t1 n))))
```

```
; an induction hint.
(defn memset-induct (s i* i n lst ch)
  (if (equal (sub1 n) 0)
      t
    (memset-induct (stepn s 3) (add 32 i* 1) (add1 i) (sub1 n)
                   (put-nth ch i lst) ch)))

; the preconditions of the initial state.
(defn memset-statep (s str n lst ch)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (mc-pc s) (mc-mem s) 38)
       (mcode-addrp (mc-pc s) (mc-mem s) (memset-code))
       (ram-addrp (sub 32 8 (read-sp s)) (mc-mem s) 24)
       (ram-addrp str (mc-mem s) n)
       (mem-lst 1 str (mc-mem s) n lst)
       (disjoint (sub 32 8 (read-sp s)) 24 str n)
       (equal str (read-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal ch (uread-mem (add 32 (read-sp s) 11) (mc-mem s) 1))
       (equal n (uread-mem (add 32 (read-sp s) 12) (mc-mem s) 4))))

; an intermediate state.
(defn memset-s0p (s i* i str n lst ch n_)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 22 (mc-pc s)) (mc-mem s) 38)
       (mcode-addrp (sub 32 22 (mc-pc s)) (mc-mem s) (memset-code))
       (ram-addrp (sub 32 4 (read-an 32 6 s)) (mc-mem s) 24)
       (ram-addrp str (mc-mem s) n_)
       (mem-lst 1 str (mc-mem s) n_ lst)
       (disjoint (sub 32 4 (read-an 32 6 s)) 24 str n_)
       (equal* (read-an 32 0 s) (add 32 str i*))
       (equal str (read-dn 32 2 s))
       (equal ch (nat-to-uint (read-dn 8 1 s)))
       (equal i (nat-to-uint i*))
       (equal n (nat-to-uint (read-dn 32 0 s)))
       (leq (plus i n) n_)
       (not (equal n 0))
       (numberp i*)
       (numberp n_)
       (nat-rangep i* 32)
       (uint-rangep n_ 32)))

; from the intial state s to exit: s --> sn.
(prove-lemma memset-s-sn (rewrite)
     (implies (and (memset-statep s str n lst ch)
                   (equal n 0))
              (and (equal (mc-status (stepn s 9)) 'running)
                   (equal (mc-pc (stepn s 9)) (rts-addr s))
                   (equal (read-dn 32 0 (stepn s 9)) str)
                   (mem-lst 1 str (mc-mem (stepn s 9)) n lst)
                   (equal (read-rn 32 15 (mc-rfile (stepn s 9)))
                          (add 32 (read-an 32 7 s) 4))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 9)))
```

```
                        (read-an 32 6 s)))))

(prove-lemma memset-s-sn-rfile (rewrite)
     (implies (and (memset-statep s str n lst ch)
                   (equal n 0)
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 9)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memset-s-sn-mem (rewrite)
     (implies (and (memset-statep s str n lst ch)
                   (equal n 0)
                   (disjoint x k (sub 32 8 (read-sp s)) 24))
              (equal (read-mem x (mc-mem (stepn s 9)) k)
                     (read-mem x (mc-mem s) k))))

; from the initial state s to s0: s --> s0.
(prove-lemma memset-s-s0 ()
     (implies (and (memset-statep s str n lst ch)
                   (not (equal n 0)))
              (memset-s0p (stepn s 7) 0 0 str n lst ch n)))

(prove-lemma memset-s-s0-else (rewrite)
     (implies (and (memset-statep s str n lst ch)
                   (not (equal n 0)))
              (and (equal (linked-rts-addr (stepn s 7)) (rts-addr s))
                   (equal (linked-a6 (stepn s 7)) (read-an 32 6 s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 7)))
                          (sub 32 4 (read-sp s)))
                   (equal (rn-saved (stepn s 7))
                          (read-rn 32 2 (mc-rfile s))))))

(prove-lemma memset-s-s0-rfile (rewrite)
     (implies (and (memset-statep s str n lst ch)
                   (not (equal n 0))
                   (d3-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 7)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memset-s-s0-mem (rewrite)
     (implies (and (memset-statep s str n lst ch)
                   (not (equal n 0))
                   (disjoint x k (sub 32 8 (read-sp s)) 24))
              (equal (read-mem x (mc-mem (stepn s 7)) k)
                     (read-mem x (mc-mem s) k))))

; from s0 to exit (base case), from s0 to s0 (induction case).
; base case: s0 --> exit.
(prove-lemma memset-s0-sn-base (rewrite)
     (implies (and (memset-s0p s i* i str n lst ch n_)
                   (equal (sub1 n) 0))
              (and (equal (mc-status (stepn s 7)) 'running)
                   (equal (mc-pc (stepn s 7)) (linked-rts-addr s))
```

```
                (equal (read-dn 32 0 (stepn s 7)) str)
                (mem-lst 1 str (mc-mem (stepn s 7)) n_ (put-nth ch i lst))
                (equal (read-rn 32 14 (mc-rfile (stepn s 7)))
                       (linked-a6 s))
                (equal (read-rn 32 15 (mc-rfile (stepn s 7)))
                       (add 32 (read-an 32 6 s) 8)))))

(prove-lemma memset-s0-sn-rfile-base (rewrite)
     (implies (and (memset-s0p s i* i str n lst ch n_)
                   (equal (sub1 n) 0)
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 7)))
                     (if (d3-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (head (rn-saved s) oplen)))))

(prove-lemma memset-s0-sn-mem-base (rewrite)
     (implies (and (memset-s0p s i* i str n lst ch n_)
                   (equal (sub1 n) 0)
                   (disjoint x k str n_))
              (equal (read-mem x (mc-mem (stepn s 7)) k)
                     (read-mem x (mc-mem s) k))))

; induction case: s0 --> s0.
(prove-lemma memset-s0-s0 (rewrite)
     (implies (and (memset-s0p s i* i str n lst ch n_)
                   (not (equal (sub1 n) 0)))
              (and (memset-s0p (stepn s 3) (add 32 i* 1) (add1 i) str (sub1 n)
                               (put-nth ch i lst) ch n_)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 3)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (linked-a6 (stepn s 3)) (linked-a6 s))
                   (equal (linked-rts-addr (stepn s 3)) (linked-rts-addr s))
                   (equal (rn-saved (stepn s 3)) (rn-saved s)))))

(prove-lemma memset-s0-s0-rfile (rewrite)
     (implies (and (memset-s0p s i* i str n lst ch n_)
                   (not (equal (sub1 n) 0))
                   (d3-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 3)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma memset-s0-s0-mem (rewrite)
     (implies (and (memset-s0p s i* i str n lst ch n_)
                   (not (equal (sub1 n) 0))
                   (disjoint x k str n_))
              (equal (read-mem x (mc-mem (stepn s 3)) k)
                     (read-mem x (mc-mem s) k))))

; put together (s0 --> exit).
(prove-lemma memset-s0-sn (rewrite)
     (let ((sn (stepn s (memset-t1 n))))
       (implies (memset-s0p s i* i str n lst ch n_)
```

```
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) str)
                     (mem-lst 1 str (mc-mem sn) n_ (memset1 i n lst ch))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8)))))
     ((induct (memset-induct s i* i n lst ch))
      (disable memset-s0p read-dn)))

(prove-lemma memset-s0-sn-rfile (rewrite)
     (implies (and (memset-s0p s i* i str n lst ch n_)
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s (memset-t1 n))))
                     (if (d3-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (head (rn-saved s) oplen))))
     ((induct (memset-induct s i* i n lst ch))
      (disable memset-s0p)))

(prove-lemma memset-s0-sn-mem (rewrite)
     (implies (and (memset-s0p s i* i str n lst ch n_)
                   (disjoint x k str n_))
              (equal (read-mem x (mc-mem (stepn s (memset-t1 n))) k)
                     (read-mem x (mc-mem s) k)))
     ((induct (memset-induct s i* i n lst ch))
      (disable memset-s0p)))

; the correctness of the MEMSET program.
(prove-lemma memset-correctness (rewrite)
     (let ((sn (stepn s (memset-t n))))
       (implies (memset-statep s str n lst ch)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-sp s) 4))
                     (implies (and (leq oplen 32)
                                   (d2-7a2-5p rn))
                              (equal (read-rn oplen rn (mc-rfile sn))
                                     (read-rn oplen rn (mc-rfile s))))
                     (implies (and (disjoint x k (sub 32 8 (read-sp s)) 24)
                                   (disjoint x k str n))
                              (equal (read-mem x (mc-mem sn) k)
                                     (read-mem x (mc-mem s) k)))
                     (equal (read-dn 32 0 sn) str)
                     (mem-lst 1 str (mc-mem sn) n (memset n lst ch))))))
     ((use (memset-s-s0))
      (disable memset-statep memset-s0p rts-addr linked-rts-addr linked-a6
               read-dn)))
```

```
(disable memset-t)

; some properties of memset.
; see file cstring.events.
```

C.15 The strcat Function

```
;          Proof of the Correctness of the STRCAT Function
#|
This is part of our effort to verify the Berkeley string library.  The
Berkeley string library is widely used as part of the Berkeley Unix OS.

This is the source code of strcat function in the Berkeley string library.

/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 */
/* concatenate char append[] to the end of s[] */
char *
strcat(s, append)
        register char *s, *append;
{
        char *save = s;

        for (; *s; ++s);
        while (*s++ = *append++);
        return(save);
}

The MC68020 assembly code of the C function strcat on SUN-3 is given as
follows.  This binary is generated by "gcc -O".

0x24d8 <strcat>:        linkw fp,#0
0x24dc <strcat+4>:      moveal fp@(8),a0
0x24e0 <strcat+8>:      moveal fp@(12),a1
0x24e4 <strcat+12>:     movel a0,d1
0x24e6 <strcat+14>:     tstb a0@
0x24e8 <strcat+16>:     beq 0x24f0 <strcat+24>
0x24ea <strcat+18>:     addqw #1,a0
0x24ec <strcat+20>:     tstb a0@
0x24ee <strcat+22>:     bne 0x24ea <strcat+18>
0x24f0 <strcat+24>:     moveb a1@+,d0
0x24f2 <strcat+26>:     moveb d0,a0@+
0x24f4 <strcat+28>:     bne 0x24f0 <strcat+24>
0x24f6 <strcat+30>:     movel d1,d0
0x24f8 <strcat+32>:     unlk fp
0x24fa <strcat+34>:     rts

The machine code of the above program is:

<strcat>:      0x4e56  0x0000  0x206e  0x0008  0x226e  0x000c  0x2208  0x4a10
```

```
<strcat+16>:   0x6706  0x5248  0x4a10  0x66fa  0x1019  0x10c0  0x66fa  0x2001
<strcat+32>:   0x4e5e  0x4e75

'(78      86      0       0       32      110     0       8
  34      110     0       12      34      8       74      16
  103     6       82      72      74      16      102     250
  16      25      16      192     102     250     32      1
  78      94      78      117)

Bird-eye view of the control flow of the program:


   s -------> s0* -------> s1* --------> exit
    \_____________________/
|#

; in the logic, the above program is defined by (strcat-code).
(defn strcat-code ()
  '(78      86      0       0       32      110     0       8
    34      110     0       12      34      8       74      16
    103     6       82      72      74      16      102     250
    16      25      16      192     102     250     32      1
    78      94      78      117))

; the computation time of the program.
(defn strcat-t0 (i n1 lst1)
  (if (lessp i n1)
      (if (equal (get-nth i lst1) 0)
          2
        (splus 3 (strcat-t0 (add1 i) n1 lst1)))
    0)
  ((lessp (difference n1 i))))

(defn strcat-t1 (n1 lst1)
  (splus 7 (strcat-t0 1 n1 lst1)))

(defn strcat-t2 (j n2 lst2)
  (if (lessp j n2)
      (if (equal (get-nth j lst2) 0)
          6
        (splus 3 (strcat-t2 (add1 j) n2 lst2)))
    0)
  ((lessp (difference n2 j))))

(defn strcat-t (n1 lst1 n2 lst2)
  (if (equal (get-nth 0 lst1) 0)
      (splus 6 (strcat-t2 0 n2 lst2))
    (splus (strcat-t1 n1 lst1) (strcat-t2 0 n2 lst2))))

; two induction hints for the two loops in the program.
(defn strcat-induct0 (s i* i lst1 n1)
  (if (lessp i n1)
      (if (equal (get-nth i lst1) 0)
          t
```

```
        (strcat-induct0 (stepn s 3) (add 32 i* 1) (add1 i) lst1 n1))
    t)
  ((lessp (difference n1 i))))

(defn strcat-induct1 (s i* i lst1 j* j n2 lst2)
  (if (lessp j n2)
      (if (equal (get-nth j lst2) 0)
          t
          (strcat-induct1 (stepn s 3) (add 32 i* 1) (add1 i)
                          (put-nth (get-nth j lst2) i lst1) (add 32 j* 1)
                          (add1 j) n2 lst2))
    t)
  ((lessp (difference n2 j))))

; the preconditions of the initial state.
(defn strcat-statep (s str1 n1 lst1 str2 n2 lst2)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (mc-pc s) (mc-mem s) 36)
       (mcode-addrp (mc-pc s) (mc-mem s) (strcat-code))
       (ram-addrp (sub 32 4 (read-sp s)) (mc-mem s) 16)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (sub 32 4 (read-sp s)) 16 str1 n1)
       (disjoint (sub 32 4 (read-sp s)) 16 str2 n2)
       (disjoint str1 n1 str2 n2)
       (equal str1 (read-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal str2 (read-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
       (lessp (plus (slen 0 n1 lst1) (slen 0 n2 lst2)) n1)
       (lessp (slen 0 n2 lst2) n2)
       (numberp n1)
       (numberp n2)
       (uint-rangep n1 32)))

; an intermediate state.
(defn strcat-s0p (s i* i str1 n1 lst1 str2 n2 lst2)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 20 (mc-pc s)) (mc-mem s) 36)
       (mcode-addrp (sub 32 20 (mc-pc s)) (mc-mem s) (strcat-code))
       (ram-addrp (read-an 32 6 s) (mc-mem s) 16)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (read-an 32 6 s) 16 str1 n1)
       (disjoint (read-an 32 6 s) 16 str2 n2)
       (disjoint str1 n1 str2 n2)
       (equal* (read-an 32 0 s) (add 32 str1 i*))
       (equal str1 (read-dn 32 1 s))
       (equal str2 (read-an 32 1 s))
       (lessp (plus (slen i n1 lst1) (slen 0 n2 lst2)) n1)
```

```
       (lessp (slen 0 n2 lst2) n2)
       (numberp i*)
       (nat-rangep i* 32)
       (equal i (nat-to-uint i*))
       (numberp n1)
       (numberp n2)
       (uint-rangep n1 32)))

; an intermediate state.
(defn strcat-s1p (s i* i str1 n1 lst1 j* j str2 n2 lst2 i_)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 24 (mc-pc s)) (mc-mem s) 36)
       (mcode-addrp (sub 32 24 (mc-pc s)) (mc-mem s) (strcat-code))
       (ram-addrp (read-an 32 6 s) (mc-mem s) 16)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (read-an 32 6 s) 16 str1 n1)
       (disjoint (read-an 32 6 s) 16 str2 n2)
       (disjoint str1 n1 str2 n2)
       (equal* (read-an 32 0 s) (add 32 str1 i*))
       (equal* (read-an 32 1 s) (add 32 str2 j*))
       (equal str1 (read-dn 32 1 s))
       (lessp (plus i_ (slen j n2 lst2)) n1)
       (lessp (slen j n2 lst2) n2)
       (leq i (plus i_ j))
       (numberp i_)
       (numberp i*)
       (nat-rangep i* 32)
       (equal i (nat-to-uint i*))
       (numberp j*)
       (nat-rangep j* 32)
       (equal j (nat-to-uint j*))
       (numberp n1)
       (numberp n2)
       (uint-rangep n1 32)))

; from the initial state to s1:  s --> s1, if lst1[0] == 0.
(prove-lemma strcat-s-s1-1 ()
     (implies (and (strcat-statep s str1 n1 lst1 str2 n2 lst2)
                   (equal (get-nth 0 lst1) 0))
              (and (strcat-s1p (stepn s 6) 0 0 str1 n1 lst1 0 0 str2 n2 lst2 0)
                   (equal (linked-rts-addr (stepn s 6)) (rts-addr s))
                   (equal (linked-a6 (stepn s 6)) (read-an 32 6 s))
                   (equal* (read-rn 32 14 (mc-rfile (stepn s 6)))
                           (sub 32 4 (read-sp s))))))

(prove-lemma strcat-s-s1-1-rfile (rewrite)
     (implies (and (strcat-statep s str1 n1 lst1 str2 n2 lst2)
                   (equal (get-nth 0 lst1) 0)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 6)))
```

```
                  (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strcat-s-s1-1-mem (rewrite)
     (implies (and (strcat-statep s str1 n1 lst1 str2 n2 lst2)
                   (equal (get-nth 0 lst1) 0)
                   (disjoint x k (sub 32 4 (read-sp s)) 16))
              (equal (read-mem x (mc-mem (stepn s 6)) k)
                     (read-mem x (mc-mem s) k))))

; from the initial state to s0:  s --> s0, if lst1[0] =~ 0.
(prove-lemma strcat-s-s0 ()
     (implies (and (strcat-statep s str1 n1 lst1 str2 n2 lst2)
                   (not (equal (get-nth 0 lst1) 0)))
              (and (strcat-s0p (stepn s 7) 1 1 str1 n1 lst1 str2 n2 lst2)
                   (equal (linked-rts-addr (stepn s 7)) (rts-addr s))
                   (equal (linked-a6 (stepn s 7)) (read-an 32 6 s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 7)))
                          (sub 32 4 (read-sp s))))))

(prove-lemma strcat-s-s0-rfile (rewrite)
     (implies (and (strcat-statep s str1 n1 lst1 str2 n2 lst2)
                   (not (equal (get-nth 0 lst1) 0))
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 7)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strcat-s-s0-mem (rewrite)
     (implies (and (strcat-statep s str1 n1 lst1 str2 n2 lst2)
                   (not (equal (get-nth 0 lst1) 0))
                   (disjoint x k (sub 32 4 (read-sp s)) 16))
              (equal (read-mem x (mc-mem (stepn s 7)) k)
                     (read-mem x (mc-mem s) k))))

; from s0 to s1:  s0 --> s1.
; base case: s0 --> s1, when lst1[i] == 0.
(prove-lemma strcat-s0-s1-base (rewrite)
     (implies (and (strcat-s0p s i* i str1 n1 lst1 str2 n2 lst2)
                   (equal (get-nth i lst1) 0))
              (and (strcat-s1p (stepn s 2) i* i str1 n1 lst1 0 0 str2 n2 lst2
                               i)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 2)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (linked-a6 (stepn s 2)) (linked-a6 s))
                   (equal (linked-rts-addr (stepn s 2))
                          (linked-rts-addr s))
                   (equal (read-mem x (mc-mem (stepn s 2)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma strcat-s0-s1-rfile-base (rewrite)
     (implies (and (strcat-s0p s i* i str1 n1 lst1 str2 n2 lst2)
                   (equal (get-nth i lst1) 0)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 2)))
                     (read-rn oplen rn (mc-rfile s)))))
```

```
; induction case: s0 --> s0, when lst[i] =~ 0.
(prove-lemma strcat-s0-s0 (rewrite)
     (implies (and (strcat-s0p s i* i str1 n1 lst1 str2 n2 lst2)
                   (not (equal (get-nth i lst1) 0)))
              (and (strcat-s0p (stepn s 3) (add 32 i* 1) (add1 i)
                               str1 n1 lst1 str2 n2 lst2)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 3)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (linked-a6 (stepn s 3)) (linked-a6 s))
                   (equal (linked-rts-addr (stepn s 3))
                          (linked-rts-addr s))
                   (equal (read-mem x (mc-mem (stepn s 3)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma strcat-s0-s0-rfile (rewrite)
     (implies (and (strcat-s0p s i* i str1 n1 lst1 str2 n2 lst2)
                   (not (equal (get-nth i lst1) 0))
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 3)))
                     (read-rn oplen rn (mc-rfile s)))))

; put together:  s0 --> s1.
(prove-lemma strcat-s0p-info (rewrite)
     (implies (strcat-s0p s i* i str1 n1 lst1 str2 n2 lst2)
              (equal (lessp i n1) t)))

(prove-lemma strcat-s0-s1 (rewrite)
     (let ((s1 (stepn s (strcat-t0 i n1 lst1))))
       (implies (strcat-s0p s i* i str1 n1 lst1 str2 n2 lst2)
                (and (strcat-s1p s1 (strlen* i* i n1 lst1) (strlen i n1 lst1)
                                 str1 n1 lst1 0 0 str2 n2 lst2
                                 (strlen i n1 lst1))
                     (equal (read-rn 32 14 (mc-rfile s1))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (linked-a6 s1) (linked-a6 s))
                     (equal (linked-rts-addr s1) (linked-rts-addr s))
                     (equal (read-mem x (mc-mem s1) k)
                            (read-mem x (mc-mem s) k)))))
     ((induct (strcat-induct0 s i* i lst1 n1))
      (disable strcat-s0p strcat-s1p)))

(disable strcat-s0p-info)

(prove-lemma strcat-s0-s1-rfile (rewrite)
     (implies
      (and (strcat-s0p s i* i str1 n1 lst1 str2 n2 lst2)
           (d2-7a2-5p rn))
      (equal (read-rn oplen rn (mc-rfile (stepn s (strcat-t0 i n1 lst1))))
             (read-rn oplen rn (mc-rfile s))))
     ((induct (strcat-induct0 s i* i lst1 n1))
      (disable strcat-s0p)))

; from s1 to exit:  s1 --> sn.
```

```
; base case:  s1 --> sn, when lst2[j] == 0.
(prove-lemma strcat-s1-sn-base (rewrite)
     (implies (and (strcat-s1p s i* i str1 n1 lst1 j* j str2 n2 lst2 i_)
                   (equal (get-nth j lst2) 0))
              (and (equal (mc-status (stepn s 6)) 'running)
                   (equal (mc-pc (stepn s 6)) (linked-rts-addr s))
                   (equal (read-rn 32 0 (mc-rfile (stepn s 6))) str1)
                   (mem-lst 1 str1 (mc-mem (stepn s 6)) n1 (put-nth 0 i lst1))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 6))) (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 6)))
                          (add 32 (read-an 32 6 s) 8)))))

(prove-lemma strcat-s1-sn-rfile-base (rewrite)
     (implies (and (strcat-s1p s i* i str1 n1 lst1 j* j str2 n2 lst2 i_)
                   (equal (get-nth j lst2) 0)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 6)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strcat-s1-sn-mem-base (rewrite)
     (implies (and (strcat-s1p s i* i str1 n1 lst1 j* j str2 n2 lst2 i_)
                   (equal (get-nth j lst2) 0)
                   (disjoint x k str1 n1))
              (equal (read-mem x (mc-mem (stepn s 6)) k)
                     (read-mem x (mc-mem s) k))))

; induction case: s1 --> s1.
(prove-lemma strcat-s1-s1 (rewrite)
     (implies (and (strcat-s1p s i* i str1 n1 lst1 j* j str2 n2 lst2 i_)
                   (not (equal (get-nth j lst2) 0)))
              (and (strcat-s1p (stepn s 3) (add 32 i* 1) (add1 i) str1 n1
                               (put-nth (get-nth j lst2) i lst1) (add 32 j* 1)
                               (add1 j) str2 n2 lst2 i_)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 3)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (linked-a6 (stepn s 3)) (linked-a6 s))
                   (equal (linked-rts-addr (stepn s 3))
                          (linked-rts-addr s)))))

(prove-lemma strcat-s1-s1-rfile (rewrite)
     (implies (and (strcat-s1p s i* i str1 n1 lst1 j* j str2 n2 lst2 i_)
                   (not (equal (get-nth j lst2) 0))
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 3)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strcat-s1-s1-mem (rewrite)
     (implies (and (strcat-s1p s i* i str1 n1 lst1 j* j str2 n2 lst2 i_)
                   (not (equal (get-nth j lst2) 0))
                   (disjoint x k str1 n1))
              (equal (read-mem x (mc-mem (stepn s 3)) k)
                     (read-mem x (mc-mem s) k))))

; put together. s1 --> sn.
```

```
(prove-lemma strcat-s1-info (rewrite)
     (implies (strcat-s1p s i* i str1 n1 lst1 j* j str2 n2 lst2 i_)
              (equal (lessp j n2) t)))

(prove-lemma strcat-s1-sn (rewrite)
     (let ((sn (stepn s (strcat-t2 j n2 lst2))))
       (implies (strcat-s1p s i* i str1 n1 lst1 j* j str2 n2 lst2 i_)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) str1)
                     (mem-lst 1 str1 (mc-mem sn) n1 (strcpy1 i lst1 j n2 lst2))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8)))))
     ((induct (strcat-induct1 s i* i lst1 j* j n2 lst2))
      (disable strcat-s1p)))

(prove-lemma strcat-s1-sn-rfile (rewrite)
     (implies
      (and (strcat-s1p s i* i str1 n1 lst1 j* j str2 n2 lst2 i_)
           (d2-7a2-5p rn))
      (equal (read-rn oplen rn (mc-rfile (stepn s (strcat-t2 j n2 lst2))))
             (read-rn oplen rn (mc-rfile s))))
     ((induct (strcat-induct1  s i* i lst1 j* j n2 lst2))
      (disable strcat-s1p)))

(prove-lemma strcat-s1-sn-mem (rewrite)
     (implies (and (strcat-s1p s i* i str1 n1 lst1 j* j str2 n2 lst2 i_)
                   (disjoint x k str1 n1))
              (equal (read-mem x (mc-mem (stepn s (strcat-t2 j n2 lst2))) k)
                     (read-mem x (mc-mem s) k)))
     ((induct (strcat-induct1 s i* i lst1 j* j n2 lst2))
      (disable strcat-s1p)))

; put together: s --> s1, if lst[0] =~ 0.
(prove-lemma strcat-s-s1-2 ()
     (let ((s1 (stepn s (strcat-t1 n1 lst1))))
       (implies (and (strcat-statep s str1 n1 lst1 str2 n2 lst2)
                     (not (equal (get-nth 0 lst1) 0)))
                (and (strcat-s1p s1 (strlen* 1 1 n1 lst1) (strlen 1 n1 lst1)
                                 str1 n1 lst1 0 0 str2 n2 lst2
                                 (strlen 1 n1 lst1))
                     (equal (read-rn 32 14 (mc-rfile s1))
                            (sub 32 4 (read-sp s)))
                     (equal (linked-a6 s1) (read-an 32 6 s))
                     (equal (linked-rts-addr s1) (rts-addr s))))))
     ((use (strcat-s-s0))
      (disable strcat-statep strcat-s0p strcat-t0)))

(prove-lemma strcat-s-s1-2-rfile (rewrite)
     (implies
      (and (strcat-statep s str1 n1 lst1 str2 n2 lst2)
           (not (equal (get-nth 0 lst1) 0))
```

```
           (d2-7a2-5p rn))
      (equal (read-rn oplen rn (mc-rfile (stepn s (strcat-t1 n1 lst1))))
             (read-rn oplen rn (mc-rfile s))))
     ((use (strcat-s-s0))
      (disable strcat-statep strcat-s0p strcat-t0)))

(prove-lemma strcat-s-s1-2-mem (rewrite)
     (implies
      (and (strcat-statep s str1 n1 lst1 str2 n2 lst2)
           (not (equal (get-nth 0 lst1) 0))
           (disjoint x k (sub 32 4 (read-sp s)) 16))
      (equal (read-mem x (mc-mem (stepn s (strcat-t1 n1 lst1))) k)
             (read-mem x (mc-mem s) k)))
     ((use (strcat-s-s0))
      (disable strcat-statep strcat-s0p strcat-t0)))

; the correctness of the strcat program.
(prove-lemma strcat-correctness (rewrite)
     (let ((sn (stepn s (strcat-t n1 lst1 n2 lst2))))
       (implies (strcat-statep s str1 n1 lst1 str2 n2 lst2)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-an 32 6 sn) (read-an 32 6 s))
                     (equal (read-an 32 7 sn)
                            (add 32 (read-an 32 7 s) 4))
                     (implies (d2-7a2-5p rn)
                              (equal (read-rn oplen rn (mc-rfile sn))
                                     (read-rn oplen rn (mc-rfile s))))
                     (implies (and (disjoint x k str1 n1)
                                   (disjoint x k (sub 32 4 (read-sp s)) 16))
                              (equal (read-mem x (mc-mem sn) k)
                                     (read-mem x (mc-mem s) k)))
                     (equal (read-dn 32 0 sn) str1)
                     (mem-lst 1 str1 (mc-mem sn) n1
                              (strcat n1 lst1 n2 lst2))))))
     ((use (strcat-s-s1-1) (strcat-s-s1-2))
      (disable strcat-statep strcat-s1p strcat-t1 strcat-t2 read-dn
               linked-rts-addr linked-a6)))

(disable strcat-t)

; some properties of strcat.
; see file cstring.events.
```

C.16 The strchr Function

```
;          Proof of the Correctness of the STRCHR Function
#|
This is part of our effort to verify the Berkeley string library.  The
Berkeley string library is widely used as part of the Berkeley Unix OS.

This is the source code of strchr function in the Berkeley string library.
```

```
/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 */
char *
strchr(p, ch)
        register char *p, ch;
{
        for (;; ++p) {
                if (*p == ch)
                        return(p);
                if (!*p)
                        return((char *)NULL);
        }
        /* NOTREACHED */
}
```

The MC68020 assembly code of the C function strchr on SUN-3 is given as follows. This binary is generated by "gcc -O".

```
0x2500 <strchr>:        linkw fp,#0
0x2504 <strchr+4>:      moveal fp@(8),a0
0x2508 <strchr+8>:      moveb fp@(15),d0
0x250c <strchr+12>:     cmpb a0@,d0
0x250e <strchr+14>:     bne 0x2514 <strchr+20>
0x2510 <strchr+16>:     movel a0,d0
0x2512 <strchr+18>:     bra 0x2520 <strchr+32>
0x2514 <strchr+20>:     tstb a0@
0x2516 <strchr+22>:     bne 0x251c <strchr+28>
0x2518 <strchr+24>:     clrl d0
0x251a <strchr+26>:     bra 0x2520 <strchr+32>
0x251c <strchr+28>:     addqw #1,a0
0x251e <strchr+30>:     bra 0x250c <strchr+12>
0x2520 <strchr+32>:     unlk fp
0x2522 <strchr+34>:     rts
```

The machine code of the above program is:

```
<strchr>:      0x4e56  0x0000  0x206e  0x0008  0x102e  0x000f  0xb010  0x6604
<strchr+16>:   0x2008  0x600c  0x4a10  0x6604  0x4280  0x6004  0x5248  0x60ec
<strchr+32>:   0x4e5e  0x4e75

'(78      86      0       0       32      110     0       8
  16      46      0       15      176     16      102     4
  32      8       96      12      74      16      102     4
  66      128     96      4       82      72      96      236
  78      94      78      117)
|#

; in the logic, the above program is defined by (strchr-code).
(defn strchr-code ()
  '(78      86      0       0       32      110     0       8
    16      46      0       15      176     16      102     4
```

```
    32      8       96      12      74      16      102     4
    66      128     96      4       82      72      96      236
    78      94      78      117))

; the computation time of the program.
(defn strchr1-t (i n lst ch)
  (if (lessp i n)
      (if (equal (get-nth i lst) ch)
          6
        (if (equal (get-nth i lst) 0)
            8
          (splus 6 (strchr1-t (add1 i) n lst ch))))
    0)
  ((lessp (difference n i))))

(defn strchr-t (n lst ch)
  (splus 3 (strchr1-t 0 n lst ch)))

; an induction hint.
(defn strchr-induct (s i* i n lst ch)
  (if (lessp i n)
      (if (equal (get-nth i lst) ch)
          t
        (if (equal (get-nth i lst) 0)
            8
          (strchr-induct (stepn s 6) (add 32 i* 1) (add1 i) n lst ch)))
    t)
  ((lessp (difference n i))))

; the preconditions of the initial state.
(defn strchr-statep (s str n lst ch)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (mc-pc s) (mc-mem s) 38)
       (mcode-addrp (mc-pc s) (mc-mem s) (strchr-code))
       (ram-addrp (sub 32 4 (read-sp s)) (mc-mem s) 16)
       (ram-addrp str (mc-mem s) n)
       (mem-lst 1 str (mc-mem s) n lst)
       (disjoint (sub 32 4 (read-sp s)) 16 str n)
       (equal str (read-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal ch (uread-mem (add 32 (read-sp s) 11) (mc-mem s) 1))
       (lessp (slen 0 n lst) n)
       (numberp n)
       (not (equal (nat-to-uint str) 0))
       (uint-rangep (plus (nat-to-uint str) n) 32)))

; an intermediate state.
(defn strchr-s0p (s i* i str n lst ch)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 12 (mc-pc s)) (mc-mem s) 38)
       (mcode-addrp (sub 32 12 (mc-pc s)) (mc-mem s) (strchr-code))
       (ram-addrp (read-an 32 6 s) (mc-mem s) 16)
       (ram-addrp str (mc-mem s) n)
```

```
       (mem-lst 1 str (mc-mem s) n lst)
       (disjoint (read-an 32 6 s) 16 str n)
       (equal* (read-an 32 0 s) (add 32 str i*))
       (equal ch (nat-to-uint (read-dn 8 0 s)))
       (lessp (slen i n lst) n)
       (lessp i n)
       (numberp i*)
       (nat-rangep i* 32)
       (equal i (nat-to-uint i*))
       (numberp n)
       (uint-rangep n 32)))

; from the initial state s to s0: s --> s0;
(prove-lemma strchr-s-s0 ()
     (implies (strchr-statep s str n lst ch)
              (strchr-s0p (stepn s 3) 0 0 str n lst ch)))

(prove-lemma strchr-s-s0-else (rewrite)
     (implies (strchr-statep s str n lst ch)
              (and (equal (linked-rts-addr (stepn s 3)) (rts-addr s))
                   (equal (linked-a6 (stepn s 3)) (read-an 32 6 s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 3)))
                          (sub 32 4 (read-sp s))))))

(prove-lemma strchr-s-s0-rfile (rewrite)
     (implies (and (strchr-statep s str n lst ch)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 3)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strchr-s-s0-mem (rewrite)
     (implies (and (strchr-statep s str n lst ch)
                   (disjoint x k (sub 32 4 (read-sp s)) 16))
              (equal (read-mem x (mc-mem (stepn s 3)) k)
                     (read-mem x (mc-mem s) k))))

; from s0 to exit: s0 --> sn.
; base case 1. s0 --> sn, when lst[i] = ch.
(prove-lemma strchr-s0-sn-base1 (rewrite)
     (implies (and (strchr-s0p s i* i str n lst ch)
                   (equal (get-nth i lst) ch))
              (and (equal (mc-status (stepn s 6)) 'running)
                   (equal (mc-pc (stepn s 6)) (linked-rts-addr s))
                   (equal (read-dn 32 0 (stepn s 6)) (add 32 str i*))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 6)))
                          (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 6)))
                          (add 32 (read-an 32 6 s) 8))
                   (equal (read-mem x (mc-mem (stepn s 6)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma strchr-s0-sn-rfile-base1 (rewrite)
     (implies (and (strchr-s0p s i* i str n lst ch)
                   (equal (get-nth i lst) ch)
```

```
                (d2-7a2-5p rn))
           (equal (read-rn oplen rn (mc-rfile (stepn s 6)))
                  (read-rn oplen rn (mc-rfile s)))))

; base case 2: s0 --> sn, when lst[i] =- ch and lst[i] = 0.
(prove-lemma strchr-s0-sn-base2 (rewrite)
     (implies (and (strchr-s0p s i* i str n lst ch)
                   (not (equal (get-nth i lst) ch))
                   (equal (get-nth i lst) 0))
              (and (equal (mc-status (stepn s 8)) 'running)
                   (equal (mc-pc (stepn s 8)) (linked-rts-addr s))
                   (equal (read-dn 32 0 (stepn s 8)) 0)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 8)))
                          (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 8)))
                          (add 32 (read-an 32 6 s) 8))
                   (equal (read-mem x (mc-mem (stepn s 8)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma strchr-s0-sn-rfile-base2 (rewrite)
     (implies (and (strchr-s0p s i* i str n lst ch)
                   (not (equal (get-nth i lst) ch))
                   (equal (get-nth i lst) 0)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 8)))
                     (read-rn oplen rn (mc-rfile s)))))

; induction case: s0 --> s0, when lst[i] =- ch and lst[i] =- 0.
(prove-lemma strchr-s0-s0 (rewrite)
     (implies (and (strchr-s0p s i* i str n lst ch)
                   (not (equal (get-nth i lst) ch))
                   (not (equal (get-nth i lst) 0)))
              (and (strchr-s0p (stepn s 6) (add 32 i* 1) (add1 i)
                               str n lst ch)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 6)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (linked-a6 (stepn s 6)) (linked-a6 s))
                   (equal (linked-rts-addr (stepn s 6)) (linked-rts-addr s))
                   (equal (read-mem x (mc-mem (stepn s 6)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma strchr-s0-s0-rfile (rewrite)
     (implies (and (strchr-s0p s i* i str n lst ch)
                   (not (equal (get-nth i lst) ch))
                   (not (equal (get-nth i lst) 0))
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 6)))
                     (read-rn oplen rn (mc-rfile s)))))

; put together. s0 --> exit.
(prove-lemma strchr-s0p-info (rewrite)
     (implies (strchr-s0p s i* i str n lst ch)
              (equal (lessp i n) t)))
```

```
(prove-lemma strchr-s0-sn (rewrite)
     (let ((sn (stepn s (strchr1-t i n lst ch))))
       (implies (strchr-s0p s i* i str n lst ch)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn)
                            (if (strchr i n lst ch)
                                (add 32 str (strchr* i* i n lst ch))
                              0))
                     (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))
                     (equal (read-mem x (mc-mem sn) k)
                            (read-mem x (mc-mem s) k))))))
     ((induct (strchr-induct s i* i n lst ch))
      (disable strchr-s0p read-dn)))

(prove-lemma strchr-s0-sn-rfile (rewrite)
     (let ((sn (stepn s (strchr1-t i n lst ch))))
       (implies (and (strchr-s0p s i* i str n lst ch)
                     (d2-7a2-5p rn))
                (equal (read-rn oplen rn (mc-rfile sn))
                       (read-rn oplen rn (mc-rfile s))))))
     ((induct (strchr-induct s i* i n lst ch))
      (disable strchr-s0p)))

(disable strchr-s0p-info)

; the correctness of strchr.
(prove-lemma strchr-correctness (rewrite)
     (let ((sn (stepn s (strchr-t n lst ch))))
       (implies (strchr-statep s str n lst ch)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-sp s) 4))
                     (implies (d2-7a2-5p rn)
                              (equal (read-rn oplen rn (mc-rfile sn))
                                     (read-rn oplen rn (mc-rfile s))))
                     (implies (disjoint x k (sub 32 4 (read-sp s)) 16)
                              (equal (read-mem x (mc-mem sn) k)
                                     (read-mem x (mc-mem s) k)))
                     (equal (read-dn 32 0 sn)
                            (if (strchr 0 n lst ch)
                                (add 32 str (strchr* 0 0 n lst ch))
                              0)))))
     ((use (strchr-s-s0))
      (disable strchr-statep strchr-s0p read-dn linked-rts-addr linked-a6)))

(disable strchr-t)

; strchr* --> strchr.
```

```
(prove-lemma strchr*-strchr (rewrite)
     (implies (and (strchr i n lst ch)
                   (equal i (nat-to-uint i*))
                   (nat-rangep i* 32)
                   (uint-rangep n 32))
              (equal (nat-to-uint (strchr* i* i n lst ch))
                     (strchr i n lst ch)))
     ((induct (strchr* i* i n lst ch))))

(prove-lemma strchr-non-zerop-la ()
     (let ((sn (stepn s (strchr-t n lst ch))))
       (implies (and (strchr-statep s str n lst ch)
                     (nat-rangep str 32)
                     (not (equal (nat-to-uint str) 0))
                     (uint-rangep (plus (nat-to-uint str) n) 32)
                     (strchr 0 n lst ch))
              (not (equal (nat-to-uint (read-dn 32 0 sn)) 0))))
     ((enable nat-rangep-la)
      (disable strchr-statep read-dn)))

(prove-lemma strchr-non-zerop (rewrite)
     (let ((sn (stepn s (strchr-t n lst ch))))
       (implies (and (strchr-statep s str n lst ch)
                     (strchr 0 n lst ch))
              (not (equal (nat-to-uint (read-dn 32 0 sn)) 0))))
     ((use (strchr-non-zerop-la))))

(disable strchr*)

; some properties of strchr.
; see file cstring.events.
```

C.17 The strcmp Function

```
;          Proof of the Correctness of the STRCMP Function
#|
This is part of our effort to verify the Berkeley string library.  The
Berkeley string library is widely used as part of the Berkeley Unix OS.

This is the source code of strcmp function in the Berkeley string library.

/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 */
/* compare (unsigned) char s1[] to s2[] */
int
strcmp(s1, s2)
        register const char *s1, *s2;
{
        while (*s1 == *s2++)
                if (*s1++ == 0)
```

```
                        return (0);
        return (*(unsigned char *)s1 - *(unsigned char *)--s2);
}
```

The MC68020 assembly code of the C function strcmp on SUN-3 is given as follows. This binary is generated by "gcc -O".

```
0x2528 <strcmp>:        linkw fp,#0
0x252c <strcmp+4>:      movel d2,sp@-
0x252e <strcmp+6>:      moveal fp@(8),a0
0x2532 <strcmp+10>:     moveal fp@(12),a1
0x2536 <strcmp+14>:     bra 0x2540 <strcmp+24>
0x2538 <strcmp+16>:     tstb a0@+
0x253a <strcmp+18>:     bne 0x2540 <strcmp+24>
0x253c <strcmp+20>:     clrl d0
0x253e <strcmp+22>:     bra 0x2550 <strcmp+40>
0x2540 <strcmp+24>:     moveb a0@,d2
0x2542 <strcmp+26>:     cmpb a1@+,d2
0x2544 <strcmp+28>:     beq 0x2538 <strcmp+16>
0x2546 <strcmp+30>:     clrl d0
0x2548 <strcmp+32>:     moveb a0@,d0
0x254a <strcmp+34>:     clrl d1
0x254c <strcmp+36>:     moveb a1@-,d1
0x254e <strcmp+38>:     subl d1,d0
0x2550 <strcmp+40>:     movel fp@(-4),d2
0x2554 <strcmp+44>:     unlk fp
0x2556 <strcmp+46>:     rts
```

The machine code of the above program is:

```
<strcmp>:      0x4e56  0x0000  0x2f02  0x206e  0x0008  0x226e  0x000c  0x6008
<strcmp+16>:   0x4a18  0x6604  0x4280  0x6010  0x1410  0xb419  0x67f2  0x4280
<strcmp+32>:   0x1010  0x4281  0x1221  0x9081  0x242e  0xfffc  0x4e5e  0x4e75

'(78      86      0       0       47      2       32      110
  0       8       34      110     0       12      96      8
  74      24      102     4       66      128     96      16
  20      16      180     25      103     242     66      128
  16      16      66      129     18      33      144     129
  36      46      255     252     78      94      78      117)
|#

; in the logic, the above program is defined by (strcmp-code).
(defn strcmp-code ()
  '(78      86      0       0       47      2       32      110
    0       8       34      110     0       12      96      8
    74      24      102     4       66      128     96      16
    20      16      180     25      103     242     66      128
    16      16      66      129     18      33      144     129
    36      46      255     252     78      94      78      117))

(constrain strcmp-load (rewrite)
   (equal (strcmp-loadp s)
          (and (evenp (strcmp-addr))
```

```
              (numberp (strcmp-addr))
              (nat-rangep (strcmp-addr) 32)
              (rom-addrp (strcmp-addr) (mc-mem s) 48)
              (mcode-addrp (strcmp-addr) (mc-mem s) (strcmp-code))))
    ((strcmp-loadp (lambda (s) f))
     (strcmp-addr (lambda () 1))))

(prove-lemma stepn-strcmp-loadp (rewrite)
     (equal (strcmp-loadp (stepn s n))
            (strcmp-loadp s)))

; the computation time of the program.
(defn strcmp1-t (i n1 lst1 lst2)
  (if (lessp i n1)
      (if (equal (get-nth i lst1) (get-nth i lst2))
          (if (equal (get-nth i lst1) (null))
              10
            (splus 5 (strcmp1-t (add1 i) n1 lst1 lst2)))
        11)
    0)
  ((lessp (difference n1 i))))

(defn strcmp-t (n1 lst1 lst2)
  (splus 5 (strcmp1-t 0 n1 lst1 lst2)))

; an induction hint.
(defn strcmp-induct (s i* i n1 lst1 lst2)
  (if (lessp i n1)
      (if (equal (get-nth i lst1) (get-nth i lst2))
          (if (equal (get-nth i lst1) (null))
              t
            (strcmp-induct (stepn s 5) (add 32 i* 1) (add1 i) n1 lst1 lst2))
        t)
    t)
  ((lessp (difference n1 i))))

; the preconditions on the initial state.
(defn strcmp-statep (s str1 n1 lst1 str2 n2 lst2)
  (and (equal (mc-status s) 'running)
       (strcmp-loadp s)
       (equal (mc-pc s) (strcmp-addr))
       (ram-addrp (sub 32 8 (read-sp s)) (mc-mem s) 20)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (sub 32 8 (read-sp s)) 20 str1 n1)
       (disjoint (sub 32 8 (read-sp s)) 20 str2 n2)
       (equal str1 (read-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal str2 (read-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
       (stringp 0 n1 lst1)
       (leq n1 n2)
       (numberp n1)
       (numberp n2)
```

```
      (uint-rangep n2 32)))

; an intermediate state.
(defn strcmp-s0p (s i* i str1 n1 lst1 str2 n2 lst2)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 24 (mc-pc s)) (mc-mem s) 48)
       (mcode-addrp (sub 32 24 (mc-pc s)) (mc-mem s) (strcmp-code))
       (ram-addrp (sub 32 4 (read-an 32 6 s)) (mc-mem s) 20)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (sub 32 4 (read-an 32 6 s)) 20 str1 n1)
       (disjoint (sub 32 4 (read-an 32 6 s)) 20 str2 n2)
       (equal* (read-an 32 0 s) (add 32 str1 i*))
       (equal* (read-an 32 1 s) (add 32 str2 i*))
       (equal i (nat-to-uint i*))
       (stringp i n1 lst1)
       (lessp i n1)
       (leq n1 n2)
       (numberp i*)
       (nat-rangep i* 32)
       (numberp n1)
       (numberp n2)
       (uint-rangep n2 32)))

; from the initial state s to s0: s --> s0.
(prove-lemma strcmp-s-s0 ()
     (implies (strcmp-statep s str1 n1 lst1 str2 n2 lst2)
              (strcmp-s0p (stepn s 5) 0 0 str1 n1 lst1 str2 n2 lst2)))

(prove-lemma strcmp-s-s0-else (rewrite)
     (implies (strcmp-statep s str1 n1 lst1 str2 n2 lst2)
              (and (equal (linked-rts-addr (stepn s 5)) (rts-addr s))
                   (equal (linked-a6 (stepn s 5)) (read-an 32 6 s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 5)))
                          (sub 32 4 (read-sp s)))
                   (equal (rn-saved (stepn s 5)) (read-dn 32 2 s)))))

(prove-lemma strcmp-s-s0-rfile (rewrite)
     (implies (and (strcmp-statep s str1 n1 lst1 str2 n2 lst2)
                   (d3-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 5)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strcmp-s-s0-mem (rewrite)
     (implies (and (strcmp-statep s str1 n1 lst1 str2 n2 lst2)
                   (disjoint x k (sub 32 8 (read-sp s)) 20))
              (equal (read-mem x (mc-mem (stepn s 5)) k)
                     (read-mem x (mc-mem s) k))))

; from s0 to exit: s0 --> sn.
; base case 1: s0 --> sn.  lst1[i] =- lst2[i].
```

```
(prove-lemma strcmp-s0-sn-base1 (rewrite)
     (implies (and (strcmp-s0p s i* i str1 n1 lst1 str2 n2 lst2)
                   (not (equal (get-nth i lst1) (get-nth i lst2))))
              (and (equal (mc-status (stepn s 11)) 'running)
                   (equal (mc-pc (stepn s 11)) (linked-rts-addr s))
                   (equal (iread-dn 32 0 (stepn s 11))
                          (idifference (get-nth i lst1) (get-nth i lst2)))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 11)))
                          (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 11)))
                          (add 32 (read-an 32 6 s) 8))
                   (equal (read-mem x (mc-mem (stepn s 11)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma strcmp-s0-sn-rfile-base1 (rewrite)
     (implies (and (strcmp-s0p s i* i str1 n1 lst1 str2 n2 lst2)
                   (not (equal (get-nth i lst1) (get-nth i lst2)))
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 11)))
                     (if (d3-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (head (rn-saved s) oplen)))))

; base case 2: s0 --> sn.  lst[i] = lst2[i] and lst[i] = 0.
(prove-lemma strcmp-s0-sn-base2 (rewrite)
     (implies (and (strcmp-s0p s i* i str1 n1 lst1 str2 n2 lst2)
                   (equal (get-nth i lst1) (get-nth i lst2))
                   (equal (get-nth i lst1) 0))
              (and (equal (mc-status (stepn s 10)) 'running)
                   (equal (mc-pc (stepn s 10)) (linked-rts-addr s))
                   (equal (iread-dn 32 0 (stepn s 10)) 0)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 10)))
                          (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 10)))
                          (add 32 (read-an 32 6 s) 8))
                   (equal (read-mem x (mc-mem (stepn s 10)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma strcmp-s0-sn-rfile-base2 (rewrite)
     (implies (and (strcmp-s0p s i* i str1 n1 lst1 str2 n2 lst2)
                   (equal (get-nth i lst1) (get-nth i lst2))
                   (equal (get-nth i lst1) 0)
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 10)))
                     (if (d3-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (head (rn-saved s) oplen)))))

; induction case: s0 --> s0. lst[i] = lst2[i] and lst[i] =- 0.
(prove-lemma strcmp-s0-s0 (rewrite)
     (implies (and (strcmp-s0p s i* i str1 n1 lst1 str2 n2 lst2)
                   (equal (get-nth i lst1) (get-nth i lst2))
```

```
                (not (equal (get-nth i lst1) 0)))
           (and (strcmp-s0p (stepn s 5) (add 32 i* 1) (add1 i)
                            str1 n1 lst1 str2 n2 lst2)
                (equal (read-rn 32 14 (mc-rfile (stepn s 5)))
                       (read-rn 32 14 (mc-rfile s)))
                (equal (linked-a6 (stepn s 5)) (linked-a6 s))
                (equal (linked-rts-addr (stepn s 5)) (linked-rts-addr s))
                (equal (rn-saved (stepn s 5)) (rn-saved s))
                (equal (read-mem x (mc-mem (stepn s 5)) k)
                       (read-mem x (mc-mem s) k)))))

(prove-lemma strcmp-s0-s0-rfile (rewrite)
     (implies (and (strcmp-s0p s i* i str1 n1 lst1 str2 n2 lst2)
                   (equal (get-nth i lst1) (get-nth i lst2))
                   (not (equal (get-nth i lst1) 0))
                   (d3-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 5)))
                     (read-rn oplen rn (mc-rfile s)))))

; put together.  s0 --> sn.
(prove-lemma strcmp-s0p-info (rewrite)
     (implies (strcmp-s0p s i* i str1 n1 lst1 str2 n2 lst2)
              (equal (lessp i n1) t)))

(prove-lemma strcmp-s0-sn (rewrite)
     (let ((sn (stepn s (strcmp1-t i n1 lst1 lst2))))
       (implies (strcmp-s0p s i* i str1 n1 lst1 str2 n2 lst2)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (iread-dn 32 0 sn) (strcmp i n1 lst1 lst2))
                     (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))
                     (equal (read-mem x (mc-mem sn) k)
                            (read-mem x (mc-mem s) k))))))
     ((induct (strcmp-induct s i* i n1 lst1 lst2))
      (disable strcmp-s0p iread-dn)))

(prove-lemma strcmp-s0-sn-rfile (rewrite)
     (implies
      (and (strcmp-s0p s i* i str1 n1 lst1 str2 n2 lst2)
           (leq oplen 32)
           (d2-7a2-5p rn))
      (equal (read-rn oplen rn (mc-rfile (stepn s (strcmp1-t i n1 lst1 lst2))))
             (if (d3-7a2-5p rn)
                 (read-rn oplen rn (mc-rfile s))
               (head (rn-saved s) oplen))))
     ((induct (strcmp-induct s i* i n1 lst1 lst2))
      (disable strcmp-s0p)))

; the correctness of strcmp.
(prove-lemma strcmp-correctness (rewrite)
     (let ((sn (stepn s (strcmp-t n1 lst1 lst2))))
       (implies (strcmp-statep s str1 n1 lst1 str2 n2 lst2)
```

```
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 7 s) 4))
                     (implies (and (leq oplen 32)
                                   (d2-7a2-5p rn))
                              (equal (read-rn oplen rn (mc-rfile sn))
                                     (read-rn oplen rn (mc-rfile s))))
                     (implies (disjoint x k (sub 32 8 (read-sp s)) 20)
                              (equal (read-mem x (mc-mem sn) k)
                                     (read-mem x (mc-mem s) k)))
                     (equal (iread-dn 32 0 sn) (strcmp 0 n1 lst1 lst2)))))
     ((use (strcmp-s-s0))
      (disable strcmp-statep strcmp-s0p iread-dn)))

(disable strcmp-t)

; some properties of strcmp.
; see file cstring.events.
```

C.18 The strcoll Function

```
;          Proof of the Correctness of the STRCOLL Function
#|
This is part of our effort to verify the Berkeley string library.  The
Berkeley string library is widely used as part of the Berkeley Unix OS.

This is the source code of strcoll function in the Berkeley string library.

/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 */
/*
 * Compare strings according to LC_COLLATE category of current locale.
 */
strcoll(s1, s2)
        const char *s1, *s2;
{
        /* LC_COLLATE is unimplemented, hence always "C" */
        return (strcmp(s1, s2));
}

The MC68020 assembly code of the C function strcoll on SUN-3 is given as
follows.  This binary is generated by "gcc -O".

0x2388 <strcoll>:       linkw fp,#0
0x238c <strcoll+4>:     movel fp@(12),sp@-
0x2390 <strcoll+8>:     movel fp@(8),sp@-
0x2394 <strcoll+12>:    jsr @#0x2358 <strcmp>
```

```
0x239a <strcoll+18>:    unlk fp
0x239c <strcoll+20>:    rts

The machine code of the above program is:

<strcoll>:    0x4e56  0x0000  0x2f2e  0x000c  0x2f2e  0x0008  0x4eb9  0x0000
<strcoll+16>: 0x2358  0x4e5e  0x4e75

'(78     86      0       0       47      46      0       12
  47     46      0       8       78      185     0       0
  35     88      78      94      78      117)
|#

; in the logic, the above program is defined by (strcoll-code).
(defn strcoll-code ()
  '(78     86      0       0       47      46      0       12
    47     46      0       8       78      185     -1      -1
    -1     -1      78      94      78      117))

(constrain strcoll-load (rewrite)
   (equal (strcoll-loadp s)
          (and (evenp (strcoll-addr))
               (numberp (strcoll-addr))
               (nat-rangep (strcoll-addr) 32)
               (rom-addrp (strcoll-addr) (mc-mem s) 22)
               (mcode-addrp (strcoll-addr) (mc-mem s) (strcoll-code))
               (strcmp-loadp s)
               (equal (pc-read-mem (add 32 (strcoll-addr) 14) (mc-mem s) 4)
                      (strcmp-addr))))
   ((strcoll-loadp (lambda (s) f))
    (strcoll-addr (lambda () 1))))

(prove-lemma stepn-strcoll-loadp (rewrite)
     (equal (strcoll-loadp (stepn s n))
            (strcoll-loadp s)))

; the computation time of the program.
(defn strcoll-t (n1 lst1 lst2)
  (splus 4 (splus (strcmp-t n1 lst1 lst2) 2)))

; the preconditions of the initial state.
(defn strcoll-statep (s str1 n1 lst1 str2 n2 lst2)
  (and (equal (mc-status s) 'running)
       (strcoll-loadp s)
       (equal (mc-pc s) (strcoll-addr))
       (ram-addrp (sub 32 24 (read-sp s)) (mc-mem s) 36)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (sub 32 24 (read-sp s)) 36 str1 n1)
       (disjoint (sub 32 24 (read-sp s)) 36 str2 n2)
       (equal str1 (read-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal str2 (read-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
```

```
       (stringp 0 n1 lst1)
       (leq n1 n2)
       (numberp n1)
       (numberp n2)
       (uint-rangep n2 32)))

; the intermediate state right before the execution of the subroutine strcmp.
(defn strcoll-s0p (s str1 n1 lst1 str2 n2 lst2)
  (and (equal (mc-status s) 'running)
       (strcoll-loadp s)
       (equal (mc-pc s) (strcmp-addr))
       (equal (rts-addr s) (add 32 (strcoll-addr) 18))
       (ram-addrp (sub 32 20 (read-an 32 6 s)) (mc-mem s) 36)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (sub 32 20 (read-an 32 6 s)) 36 str1 n1)
       (disjoint (sub 32 20 (read-an 32 6 s)) 36 str2 n2)
       (equal str1 (read-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal str2 (read-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
       (equal* (read-sp s) (sub 32 12 (read-an 32 6 s)))
       (stringp 0 n1 lst1)
       (leq n1 n2)
       (numberp n1)
       (numberp n2)
       (uint-rangep n2 32)))

; the intermediate state right after the execution of the subroutine strcmp.
(defn strcoll-s1p (s str1 n1 lst1 str2 n2 lst2)
  (and (equal (mc-status s) 'running)
       (strcoll-loadp s)
       (equal (mc-pc s) (add 32 (strcoll-addr) 18))
       (ram-addrp (sub 32 20 (read-an 32 6 s)) (mc-mem s) 36)
       (equal (iread-dn 32 0 s) (strcmp 0 n1 lst1 lst2))))

; from the initial state s to s0: s --> s0.
(prove-lemma strcoll-s-s0 ()
     (implies (strcoll-statep s str1 n1 lst1 str2 n2 lst2)
              (strcoll-s0p (stepn s 4) str1 n1 lst1 str2 n2 lst2)))

(prove-lemma strcoll-s-s0-else (rewrite)
     (implies (strcoll-statep s str1 n1 lst1 str2 n2 lst2)
              (and (equal (linked-rts-addr (stepn s 4)) (rts-addr s))
                   (equal (linked-a6 (stepn s 4)) (read-an 32 6 s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 4)))
                          (sub 32 4 (read-sp s))))))

(prove-lemma strcoll-s-s0-rfile (rewrite)
     (implies (and (strcoll-statep s str1 n1 lst1 str2 n2 lst2)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 4)))
                     (read-rn oplen rn (mc-rfile s)))))
```

```
(prove-lemma strcoll-s-s0-mem (rewrite)
     (implies (and (strcoll-statep s str1 n1 lst1 str2 n2 lst2)
                   (disjoint x k (sub 32 24 (read-sp s)) 36))
              (equal (read-mem x (mc-mem (stepn s 4)) k)
                     (read-mem x (mc-mem s) k))))

; from s0 to s1: s0 --> s1. by strcmp.
(prove-lemma strcoll-s0p-strcmp-statep ()
     (implies (strcoll-s0p s str1 n1 lst1 str2 n2 lst2)
              (strcmp-statep s str1 n1 lst1 str2 n2 lst2)))

(prove-lemma strcoll-s0-s1 ()
     (let ((s1 (stepn s (strcmp-t n1 lst1 lst2))))
       (implies (strcoll-s0p s str1 n1 lst1 str2 n2 lst2)
                (strcoll-s1p s1 str1 n1 lst1 str2 n2 lst2)))
     ((use (strcoll-s0p-strcmp-statep))
      (disable strcmp-statep strcmp-load)))

(prove-lemma strcoll-s0-s1-else (rewrite)
     (let ((s1 (stepn s (strcmp-t n1 lst1 lst2))))
       (implies (strcoll-s0p s str1 n1 lst1 str2 n2 lst2)
                (and (equal (read-rn 32 14 (mc-rfile s1))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (linked-rts-addr s1) (linked-rts-addr s))
                     (equal (linked-a6 s1) (linked-a6 s))))))
     ((use (strcoll-s0p-strcmp-statep))
      (disable strcmp-statep)))

(prove-lemma strcoll-s0-s1-rfile (rewrite)
     (let ((s1 (stepn s (strcmp-t n1 lst1 lst2))))
       (implies (and (strcoll-s0p s str1 n1 lst1 str2 n2 lst2)
                     (d2-7a2-5p rn)
                     (leq oplen 32))
                (equal (read-rn oplen rn (mc-rfile s1))
                       (read-rn oplen rn (mc-rfile s))))))
     ((use (strcoll-s0p-strcmp-statep))
      (disable strcmp-statep)))

(prove-lemma strcoll-s0-s1-mem (rewrite)
     (let ((s1 (stepn s (strcmp-t n1 lst1 lst2))))
       (implies (and (strcoll-s0p s str1 n1 lst1 str2 n2 lst2)
                     (disjoint x k (sub 32 20 (read-an 32 6 s)) 36))
                (equal (read-mem x (mc-mem s1) k)
                       (read-mem x (mc-mem s) k))))
     ((use (strcoll-s0p-strcmp-statep))
      (disable strcmp-statep)))

; from s1 to exit: s1 --> sn.
(prove-lemma strcoll-s1-sn (rewrite)
     (implies (strcoll-s1p s str1 n1 lst1 str2 n2 lst2)
              (and (equal (mc-status (stepn s 2)) 'running)
                   (equal (mc-pc (stepn s 2)) (linked-rts-addr s))
                   (equal (iread-dn 32 0 (stepn s 2))
                          (strcmp 0 n1 lst1 lst2))
```

```
                  (equal (read-rn 32 14 (mc-rfile (stepn s 2)))
                         (linked-a6 s))
                  (equal (read-rn 32 15 (mc-rfile (stepn s 2)))
                         (add 32 (read-an 32 6 s) 8))
                  (equal (read-mem x (mc-mem (stepn s 2)) k)
                         (read-mem x (mc-mem s) k)))))

(prove-lemma strcoll-s1-sn-rfile (rewrite)
     (implies (and (strcoll-s1p s str1 n1 lst1 str2 n2 lst2)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 2)))
                     (read-rn oplen rn (mc-rfile s)))))

; the correctness of strcoll.
(prove-lemma strcoll-correctness (rewrite)
     (let ((sn (stepn s (strcoll-t n1 lst1 lst2))))
       (implies (strcoll-statep s str1 n1 lst1 str2 n2 lst2)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 7 s) 4))
                     (implies (and (d2-7a2-5p rn)
                                   (leq oplen 32))
                              (equal (read-rn oplen rn (mc-rfile sn))
                                     (read-rn oplen rn (mc-rfile s))))
                     (implies (disjoint x k (sub 32 24 (read-sp s)) 36)
                              (equal (read-mem x (mc-mem sn) k)
                                     (read-mem x (mc-mem s) k)))
                     (equal (iread-dn 32 0 sn) (strcoll n1 lst1 lst2))))))
     ((use (strcoll-s-s0) (strcoll-s0-s1 (s (stepn s 4))))
      (disable strcoll-statep strcoll-s0p strcoll-s1p)))

(disable strcoll-t)

; some properties of strcoll.
; the same as strcmp.
```

C.19 The strcpy Function

```
;          Proof of the Correctness of the STRCPY Function
#|
This is part of our effort to verify the Berkeley string library.  The
Berkeley string library is widely used as part of the Berkeley Unix OS.

This is the source code of strcpy function in the Berkeley string library.

/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 */
```

```
/* copy char from[] to to[] */
char *
strcpy(to, from)
        register char *to, *from;
{
        char *save = to;

        for (; *to = *from; ++from, ++to);
        return(save);
}
```

The MC68020 assembly code of the C function strcpy on SUN-3 is given as follows. This binary is generated by "gcc -O".

```
0x2558 <strcpy>:        linkw fp,#0
0x255c <strcpy+4>:      moveal fp@(8),a0
0x2560 <strcpy+8>:      moveal fp@(12),a1
0x2564 <strcpy+12>:     movel a0,d1
0x2566 <strcpy+14>:     bra 0x256c <strcpy+20>
0x2568 <strcpy+16>:     addqw #1,a1
0x256a <strcpy+18>:     addqw #1,a0
0x256c <strcpy+20>:     moveb a1@,d0
0x256e <strcpy+22>:     moveb d0,a0@
0x2570 <strcpy+24>:     bne 0x2568 <strcpy+16>
0x2572 <strcpy+26>:     movel d1,d0
0x2574 <strcpy+28>:     unlk fp
0x2576 <strcpy+30>:     rts
```

The machine code of the above program is:

```
<strcpy>:      0x4e56  0x0000  0x206e  0x0008  0x226e  0x000c  0x2208  0x6004
<strcpy+16>:   0x5249  0x5248  0x1011  0x1080  0x66f6  0x2001  0x4e5e  0x4e75

'(78      86      0       0       32      110     0       8
  34      110     0       12      34      8       96      4
  82      73      82      72      16      17      16      128
  102     246     32      1       78      94      78      117)
|#

; in the logic, the above program is defined by (strcpy-code).
(defn strcpy-code ()
  '(78      86      0       0       32      110     0       8
    34      110     0       12      34      8       96      4
    82      73      82      72      16      17      16      128
    102     246     32      1       78      94      78      117))

; the computation time of the program.
(defn strcpy1-t (i n2 lst2)
  (if (lessp i n2)
      (if (equal (get-nth i lst2) (null))
          6
        (splus 5 (strcpy1-t (add1 i) n2 lst2)))
    0)
  ((lessp (difference n2 i))))
```

```
; the computation time for the program (strcpy-code).
(defn strcpy-t (n2 lst2)
  (splus 5 (strcpy1-t 0 n2 lst2)))

; an induction hint for the loop.
(defn strcpy-induct (s i* i lst1 n2 lst2)
  (if (lessp i n2)
      (if (equal (get-nth i lst2) (null))
          t
        (strcpy-induct (stepn s 5) (add 32 i* 1) (add1 i)
                       (put-nth (get-nth i lst2) i lst1) n2 lst2))
    t)
  ((lessp (difference n2 i))))

; the pre-conditions of the initial state.
(defn strcpy-statep (s str1 n1 lst1 str2 n2 lst2)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (mc-pc s) (mc-mem s) 32)
       (mcode-addrp (mc-pc s) (mc-mem s) (strcpy-code))
       (ram-addrp (sub 32 4 (read-sp s)) (mc-mem s) 16)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (sub 32 4 (read-sp s)) 16 str1 n1)
       (disjoint (sub 32 4 (read-sp s)) 16 str2 n2)
       (disjoint str1 n1 str2 n2)
       (equal str1 (read-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal str2 (read-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
       (lessp (slen 0 n2 lst2) n2)
       (leq n2 n1)
       (numberp n1)
       (numberp n2)
       (uint-rangep n1 32)))

; an intermediate state.
(defn strcpy-s0p (s i* i str1 n1 lst1 str2 n2 lst2)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 20 (mc-pc s)) (mc-mem s) 32)
       (mcode-addrp (sub 32 20 (mc-pc s)) (mc-mem s) (strcpy-code))
       (ram-addrp (read-an 32 6 s) (mc-mem s) 16)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (read-an 32 6 s) 16 str1 n1)
       (disjoint (read-an 32 6 s) 16 str2 n2)
       (disjoint str1 n1 str2 n2)
       (equal str1 (read-dn 32 1 s))
       (equal* (read-an 32 0 s) (add 32 str1 i*))
       (equal* (read-an 32 1 s) (add 32 str2 i*))
```

```
       (lessp (slen i n2 lst2) n2)
       (leq n2 n1)
       (lessp i n2)
       (numberp i*)
       (nat-rangep i* 32)
       (equal i (nat-to-uint i*))
       (numberp n1)
       (numberp n2)
       (uint-rangep n1 32)))

; from the initial state s to s0: s --> s0.
(prove-lemma strcpy-s-s0 ()
     (implies (strcpy-statep s str1 n1 lst1 str2 n2 lst2)
              (strcpy-s0p (stepn s 5) 0 0 str1 n1 lst1 str2 n2 lst2)))

(prove-lemma strcpy-s-s0-else (rewrite)
     (implies (strcpy-statep s str1 n1 lst1 str2 n2 lst2)
              (and (equal (linked-rts-addr (stepn s 5)) (rts-addr s))
                   (equal (linked-a6 (stepn s 5)) (read-an 32 6 s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 5)))
                          (sub 32 4 (read-sp s))))))

(prove-lemma strcpy-s-s0-rfile (rewrite)
     (implies (and (strcpy-statep s str1 n1 lst1 str2 n2 lst2)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 5)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strcpy-s-s0-mem (rewrite)
     (implies (and (strcpy-statep s str1 n1 lst1 str2 n2 lst2)
                   (disjoint x k (sub 32 4 (read-sp s)) 16))
              (equal (read-mem x (mc-mem (stepn s 5)) k)
                     (read-mem x (mc-mem s) k))))

; from s0 to exit (base case), from s0 to s0 (induction case).
; base case: s0 --> exit.
(prove-lemma strcpy-s0-sn-base (rewrite)
     (implies
      (and (strcpy-s0p s i* i str1 n1 lst1 str2 n2 lst2)
           (equal (get-nth i lst2) 0))
      (and (equal (mc-status (stepn s 6)) 'running)
           (equal (mc-pc (stepn s 6)) (linked-rts-addr s))
           (equal (read-rn 32 0 (mc-rfile (stepn s 6))) str1)
           (mem-lst 1 str1 (mc-mem (stepn s 6)) n1 (put-nth 0 i lst1))
           (equal (read-rn 32 14 (mc-rfile (stepn s 6))) (linked-a6 s))
           (equal (read-rn 32 15 (mc-rfile (stepn s 6)))
                  (add 32 (read-an 32 6 s) 8)))))

(prove-lemma strcpy-s0-sn-rfile-base (rewrite)
     (implies (and (strcpy-s0p s i* i str1 n1 lst1 str2 n2 lst2)
                   (d2-7a2-5p rn)
                   (equal (get-nth i lst2) 0))
              (equal (read-rn oplen rn (mc-rfile (stepn s 6)))
                     (read-rn oplen rn (mc-rfile s)))))
```

```
(prove-lemma strcpy-s0-sn-mem-base (rewrite)
     (implies (and (strcpy-s0p s i* i str1 n1 lst1 str2 n2 lst2)
                   (disjoint x k str1 n1)
                   (equal (get-nth i lst2) 0))
              (equal (read-mem x (mc-mem (stepn s 6)) k)
                     (read-mem x (mc-mem s) k))))

; induction case: s0 --> s0.
(prove-lemma strcpy-s0-s0 (rewrite)
     (implies (and (strcpy-s0p s i* i str1 n1 lst1 str2 n2 lst2)
                   (not (equal (get-nth i lst2) 0)))
              (and (strcpy-s0p (stepn s 5) (add 32 i* 1) (add1 i) str1 n1
                               (put-nth (get-nth i lst2) i lst1) str2 n2 lst2)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 5)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (linked-a6 (stepn s 5)) (linked-a6 s))
                   (equal (linked-rts-addr (stepn s 5))
                          (linked-rts-addr s)))))

(prove-lemma strcpy-s0-s0-rfile (rewrite)
     (implies (and (strcpy-s0p s i* i str1 n1 lst1 str2 n2 lst2)
                   (d2-7a2-5p rn)
                   (not (equal (get-nth i lst2) 0)))
              (equal (read-rn oplen rn (mc-rfile (stepn s 5)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strcpy-s0-s0-mem (rewrite)
     (implies (and (strcpy-s0p s i* i str1 n1 lst1 str2 n2 lst2)
                   (disjoint x k str1 n1)
                   (not (equal (get-nth i lst2) 0)))
              (equal (read-mem x (mc-mem (stepn s 5)) k)
                     (read-mem x (mc-mem s) k))))

; put together (s0 --> exit).
(prove-lemma strcpy-s0p-info (rewrite)
     (implies (strcpy-s0p s i* i str1 n1 lst1 str2 n2 lst2)
              (equal (lessp i n2) t)))

(prove-lemma strcpy-s0-sn (rewrite)
     (let ((sn (stepn s (strcpy1-t i n2 lst2))))
       (implies (strcpy-s0p s i* i str1 n1 lst1 str2 n2 lst2)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) str1)
                     (mem-lst 1 str1 (mc-mem sn) n1 (strcpy i lst1 n2 lst2))
                     (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))))))
     ((induct (strcpy-induct s i* i lst1 n2 lst2))
      (disable strcpy-s0p)))

(prove-lemma strcpy-s0-sn-rfile (rewrite)
     (implies
```

```
      (and (strcpy-s0p s i* i str1 n1 lst1 str2 n2 lst2)
           (d2-7a2-5p rn))
      (equal (read-rn oplen rn (mc-rfile (stepn s (strcpy1-t i n2 lst2))))
             (read-rn oplen rn (mc-rfile s))))
     ((induct (strcpy-induct s i* i lst1 n2 lst2))
      (disable strcpy-s0p)))

(prove-lemma strcpy-s0-sn-mem (rewrite)
     (implies (and (strcpy-s0p s i* i str1 n1 lst1 str2 n2 lst2)
                   (disjoint x k str1 n1))
              (equal (read-mem x (mc-mem (stepn s (strcpy1-t i n2 lst2))) k)
                     (read-mem x (mc-mem s) k)))
     ((induct (strcpy-induct s i* i lst1 n2 lst2))
      (disable strcpy-s0p)))

(disable strcpy-s0p-info)

; the correctness of the strcpy program.
(prove-lemma strcpy-correctness (rewrite)
     (let ((sn (stepn s (strcpy-t n2 lst2))))
       (implies (strcpy-statep s str1 n1 lst1 str2 n2 lst2)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 7 s) 4))
                     (implies (d2-7a2-5p rn)
                              (equal (read-rn oplen rn (mc-rfile sn))
                                     (read-rn oplen rn (mc-rfile s))))
                     (implies (and (disjoint x k str1 n1)
                                   (disjoint x k (sub 32 4 (read-sp s)) 16))
                              (equal (read-mem x (mc-mem sn) k)
                                     (read-mem x (mc-mem s) k)))
                     (equal (read-dn 32 0 sn) str1)
                     (mem-lst 1 str1 (mc-mem sn) n1 (strcpy 0 lst1 n2 lst2))))))
     ((use (strcpy-s-s0))
      (disable strcpy-statep strcpy-s0p strcpy1-t read-dn linked-rts-addr
               linked-a6)))

(disable strcpy-t)

; some properties of strcpy.
; see file cstring.events.
```

C.20 The strcspn Function

```
;          Proof of the Correctness of the STRCSPN Function
#|
This is part of our effort to verify the Berkeley string library.  The
Berkeley string library is widely used as part of the Berkeley Unix OS.
```

This is the source code of strcspn function in the Berkeley string library.

```
/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 */
size_t
strcspn(s1, s2)
        const char *s1;
        register const char *s2;
{
        register const char *p, *spanp;
        register char c, sc;

        /*
         * Stop as soon as we find any character from s2.  Note that there
         * must be a NUL in s2; it suffices to stop when we find that, too.
         */
        for (p = s1;;) {
                c = *p++;
                spanp = s2;
                do {
                        if ((sc = *spanp++) == c)
                                return (p - 1 - s1);
                } while (sc != 0);
        }
        /* NOTREACHED */
}
```

The MC68020 assembly code of the C function strcspn on SUN-3 is given as follows. This binary is generated by "gcc -O".

```
0x2578 <strcspn>:       linkw fp,#0
0x257c <strcspn+4>:     moveml d2-d3,sp@-
0x2580 <strcspn+8>:     movel fp@(8),d0
0x2584 <strcspn+12>:    movel fp@(12),d3
0x2588 <strcspn+16>:    moveal d0,a1
0x258a <strcspn+18>:    movel d0,d2
0x258c <strcspn+20>:    addql #1,d2
0x258e <strcspn+22>:    moveb a1@+,d1
0x2590 <strcspn+24>:    moveal d3,a0
0x2592 <strcspn+26>:    moveb a0@+,d0
0x2594 <strcspn+28>:    cmpb d0,d1
0x2596 <strcspn+30>:    beq 0x259e <strcspn+38>
0x2598 <strcspn+32>:    tstb d0
0x259a <strcspn+34>:    bne 0x2592 <strcspn+26>
0x259c <strcspn+36>:    bra 0x258e <strcspn+22>
0x259e <strcspn+38>:    movel a1,d0
0x25a0 <strcspn+40>:    subl d2,d0
0x25a2 <strcspn+42>:    moveml fp@(-8),d2-d3
0x25a8 <strcspn+48>:    unlk fp
0x25aa <strcspn+50>:    rts
```

The machine code of the above program is:

```
<strcspn>:     0x4e56  0x0000  0x48e7  0x3000  0x202e  0x0008  0x262e  0x000c
<strcspn+16>:  0x2240  0x2400  0x5282  0x1219  0x2043  0x1018  0xb200  0x6706
<strcspn+32>:  0x4a00  0x66f6  0x60f0  0x2009  0x9082  0x4cee  0x000c  0xfff8
<strcspn+48>:  0x4e5e  0x4e75

'(78      86      0       0       72      231     48      0
  32      46      0       8       38      46      0       12
  34      64      36      0       82      130     18      25
  32      67      16      24      178     0       103     6
  74      0       102     246     96      240     32      9
  144     130     76      238     0       12      255     248
  78      94      78      117)
|#

; in the logic, the above program is defined by (strcspn-code).
(defn strcspn-code ()
  '(78      86      0       0       72      231     48      0
    32      46      0       8       38      46      0       12
    34      64      36      0       82      130     18      25
    32      67      16      24      178     0       103     6
    74      0       102     246     96      240     32      9
    144     130     76      238     0       12      255     248
    78      94      78      117))

; the computatin time of the program.
(defn strcspn-t0 (i2 n2 lst2 ch)
  (if (lessp i2 n2)
      (if (equal (get-nth i2 lst2) ch)
          8
        (if (equal (get-nth i2 lst2) 0)
            6
          (splus 5 (strcspn-t0 (add1 i2) n2 lst2 ch))))
    0)
  ((lessp (difference n2 i2))))

(defn strcspn-t1 (n2 lst2 ch)
  (splus 2 (strcspn-t0 0 n2 lst2 ch)))

(defn strcspn-t2 (i1 n1 lst1 n2 lst2)
  (if (lessp i1 n1)
      (if (strchr 0 n2 lst2 (get-nth i1 lst1))
          (strcspn-t1 n2 lst2 (get-nth i1 lst1))
        (splus (strcspn-t1 n2 lst2 (get-nth i1 lst1))
               (strcspn-t2 (add1 i1) n1 lst1 n2 lst2)))
    0)
  ((lessp (difference n1 i1))))

(defn strcspn-t (n1 lst1 n2 lst2)
  (splus 7 (strcspn-t2 0 n1 lst1 n2 lst2)))

; two induction hints.
(defn strcspn-induct0 (s i2* i2 n2 lst2 ch)
  (if (lessp i2 n2)
```

```
      (if (equal (get-nth i2 lst2) ch)
          t
        (if (equal (get-nth i2 lst2) 0)
            t
          (strcspn-induct0 (stepn s 5) (add 32 i2* 1) (add1 i2) n2 lst2 ch)))
    t)
  ((lessp (difference n2 i2))))

(defn strcspn-induct1 (s i1* i1 n1 lst1 n2 lst2)
  (if (lessp i1 n1)
      (if (strchr 0 n2 lst2 (get-nth i1 lst1))
          t
        (strcspn-induct1 (stepn s (strcspn-t1 n2 lst2 (get-nth i1 lst1)))
                         (add 32 i1* 1) (add1 i1) n1 lst1 n2 lst2))
    t)
  ((lessp (difference n1 i1))))

; the preconditions of the initial state.
(defn strcspn-statep (s str1 n1 lst1 str2 n2 lst2)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (mc-pc s) (mc-mem s) 52)
       (mcode-addrp (mc-pc s) (mc-mem s) (strcspn-code))
       (ram-addrp (sub 32 12 (read-sp s)) (mc-mem s) 24)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (sub 32 12 (read-sp s)) 24 str1 n1)
       (disjoint (sub 32 12 (read-sp s)) 24 str2 n2)
       (equal str1 (read-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal str2 (read-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
       (lessp (slen 0 n1 lst1) n1)
       (lessp (slen 0 n2 lst2) n2)
       (numberp n1)
       (numberp n2)
       (uint-rangep n1 32)
       (uint-rangep n2 32)))

; an intermediate state s0.
(defn strcspn-s0p (s i1* i1 str1 n1 lst1 str2 n2 lst2)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 22 (mc-pc s)) (mc-mem s) 52)
       (mcode-addrp (sub 32 22 (mc-pc s)) (mc-mem s) (strcspn-code))
       (ram-addrp (sub 32 8 (read-an 32 6 s)) (mc-mem s) 24)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (sub 32 8 (read-an 32 6 s)) 24 str1 n1)
       (disjoint (sub 32 8 (read-an 32 6 s)) 24 str2 n2)
       (equal* (read-an 32 1 s) (add 32 str1 i1*))
       (equal str2 (read-dn 32 3 s))
```

```
       (equal (read-dn 32 2 s) (add 32 str1 1))
       (numberp i1*)
       (nat-rangep i1* 32)
       (equal i1 (nat-to-uint i1*))
       (lessp (slen i1 n1 lst1) n1)
       (lessp (slen 0 n2 lst2) n2)
       (numberp n1)
       (numberp n2)
       (uint-rangep n1 32)
       (uint-rangep n2 32)))

; an intermediate state s1.
(defn strcspn-s1-1p (s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 26 (mc-pc s)) (mc-mem s) 52)
       (mcode-addrp (sub 32 26 (mc-pc s)) (mc-mem s) (strcspn-code))
       (ram-addrp (sub 32 8 (read-an 32 6 s)) (mc-mem s) 24)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (sub 32 8 (read-an 32 6 s)) 24 str2 n2)
       (equal* (read-an 32 1 s) (add 32 str1 i1*))
       (equal* (read-an 32 0 s) (add 32 str2 i2*))
       (equal str2 (read-dn 32 3 s))
       (equal ch (uread-dn 8 1 s))
       (equal (read-dn 32 2 s) (add 32 str1 1))
       (numberp i1*)
       (nat-rangep i1* 32)
       (equal i1 (nat-to-uint i1*))
       (numberp i2*)
       (nat-rangep i2* 32)
       (equal i2 (nat-to-uint i2*))
       (lessp (slen 0 n2 lst2) n2)
       (lessp (slen i2 n2 lst2) n2)
       (numberp n2)
       (uint-rangep n2 32)))

(defn strcspn-s1p (s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
  (and (strcspn-s1-1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (disjoint (sub 32 8 (read-an 32 6 s)) 24 str1 n1)
       (lessp (slen i1 n1 lst1) n1)
       (numberp n1)
       (uint-rangep n1 32)))

; from the initial state s to s0: s --> s0.
(prove-lemma strcspn-s-s0 ()
     (implies (strcspn-statep s str1 n1 lst1 str2 n2 lst2)
              (strcspn-s0p (stepn s 7) 0 0 str1 n1 lst1 str2 n2 lst2)))

(prove-lemma strcspn-s-s0-else (rewrite)
     (implies (strcspn-statep s str1 n1 lst1 str2 n2 lst2)
              (and (equal (linked-rts-addr (stepn s 7)) (rts-addr s))
```

```
                (equal (linked-a6 (stepn s 7)) (read-an 32 6 s))
                (equal (read-rn 32 14 (mc-rfile (stepn s 7)))
                       (sub 32 4 (read-sp s)))
                (equal (movem-saved (stepn s 7) 4 8 2)
                       (readm-rn 32 '(2 3) (mc-rfile s))))))

(prove-lemma strcspn-s-s0-rfile (rewrite)
     (implies (and (strcspn-statep s str1 n1 lst1 str2 n2 lst2)
                   (d4-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 7)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strcspn-s-s0-mem (rewrite)
     (implies (and (strcspn-statep s str1 n1 lst1 str2 n2 lst2)
                   (disjoint x k (sub 32 12 (read-sp s)) 24))
              (equal (read-mem x (mc-mem (stepn s 7)) k)
                     (read-mem x (mc-mem s) k))))

; loop 0.
; from s0 to s1: s0 --> s1.
(prove-lemma strcspn-s0-s1-1 ()
     (implies (strcspn-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
              (strcspn-s1-1p (stepn s 2) (add 32 i1* 1) (add1 i1) str1
                             n1 lst1 0 0 str2 n2 lst2 (get-nth i1 lst1))))

(prove-lemma strchr-la ()
     (implies (and (lessp (slen i2 n2 lst2) n2)
                   (not (strchr i2 n2 lst2 ch)))
              (not (equal ch 0)))
     ((enable slen)))

(prove-lemma strcspn-s0-s1 ()
     (implies (and (strcspn-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                   (not (strchr 0 n2 lst2 (get-nth i1 lst1))))
              (strcspn-s1p (stepn s 2) (add 32 i1* 1) (add1 i1) str1
                           n1 lst1 0 0 str2 n2 lst2 (get-nth i1 lst1)))
     ((use (strchr-la (i2 0) (ch (get-nth i1 lst1))))))

(prove-lemma strcspn-s0-s1-else (rewrite)
     (let ((s1 (stepn s 2)))
       (implies (strcspn-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                (and (equal (read-rn 32 14 (mc-rfile s1))
                            (read-rn 32 14 (mc-rfile s)))
                   (equal (linked-a6 s1) (linked-a6 s))
                   (equal (linked-rts-addr s1) (linked-rts-addr s))
                   (equal (movem-saved s1 4 8 2) (movem-saved s 4 8 2))
                   (equal (read-mem x (mc-mem s1) k)
                          (read-mem x (mc-mem s) k))))))

(prove-lemma strcspn-s0-s1-rfile (rewrite)
     (implies (and (strcspn-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                   (d4-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 2)))
                     (read-rn oplen rn (mc-rfile s)))))
```

```
; loop 1.
; base case 1. from s1 to exit: s1 --> sn, when lst1[i] = lst2[j].
(prove-lemma strcspn-s1-sn-base (rewrite)
     (implies (and (strcspn-s1-1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                   (equal (get-nth i2 lst2) ch))
              (and (equal (mc-status (stepn s 8)) 'running)
                   (equal (mc-pc (stepn s 8)) (linked-rts-addr s))
                   (equal (read-dn 32 0 (stepn s 8)) (sub 32 1 i1*))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 8)))
                          (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 8)))
                          (add 32 (read-an 32 6 s) 8))
                   (equal (read-mem x (mc-mem (stepn s 8)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma strcspn-s1-sn-rfile-base (rewrite)
     (implies (and (strcspn-s1-1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                   (equal (get-nth i2 lst2) ch)
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 8)))
                     (if (d4-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (get-vlst oplen 0 rn '(2 3) (movem-saved s 4 8 2))))))

; base case 2.  s1 --> s0, when lst1[i] =~ lst2[j] and lst2[j] = 0.
(prove-lemma strcspn-s1-s0-base (rewrite)
     (implies (and (strcspn-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                   (not (equal (get-nth i2 lst2) ch))
                   (equal (get-nth i2 lst2) 0))
              (and (strcspn-s0p (stepn s 6) i1* i1 str1 n1 lst1 str2 n2 lst2)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 6)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (linked-a6 (stepn s 6)) (linked-a6 s))
                   (equal (linked-rts-addr (stepn s 6))
                          (linked-rts-addr s))
                   (equal (movem-saved (stepn s 6) 4 8 2)
                          (movem-saved s 4 8 2))
                   (equal (read-mem x (mc-mem (stepn s 6)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma strcspn-s1-s0-rfile-base (rewrite)
     (implies (and (strcspn-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                   (not (equal (get-nth i2 lst2) ch))
                   (equal (get-nth i2 lst2) 0)
                   (d4-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 6)))
                     (read-rn oplen rn (mc-rfile s)))))

; induction case. s1 --> s1, when lst1[i] =~ lst2[j] and lst2[j] =~ 0.
(prove-lemma strcspn-s1-s1-1 (rewrite)
     (implies (and (strcspn-s1-1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                   (not (equal (get-nth i2 lst2) ch))
```

```
                (not (equal (get-nth i2 lst2) 0)))
           (and (strcspn-s1-1p (stepn s 5) i1* i1 str1 n1 lst1
                               (add 32 i2* 1) (add1 i2) str2 n2 lst2 ch)
                (equal (read-rn 32 14 (mc-rfile (stepn s 5)))
                       (read-rn 32 14 (mc-rfile s)))
                (equal (linked-a6 (stepn s 5)) (linked-a6 s))
                (equal (linked-rts-addr (stepn s 5))
                       (linked-rts-addr s))
                (equal (movem-saved (stepn s 5) 4 8 2)
                       (movem-saved s 4 8 2))
                (equal (read-mem x (mc-mem (stepn s 5)) k)
                       (read-mem x (mc-mem s) k)))))

(prove-lemma strcspn-s1-s1 (rewrite)
     (implies (and (strcspn-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                   (not (equal (get-nth i2 lst2) ch))
                   (not (equal (get-nth i2 lst2) 0)))
              (strcspn-s1p (stepn s 5) i1* i1 str1 n1 lst1
                           (add 32 i2* 1) (add1 i2) str2 n2 lst2 ch)))

(prove-lemma strcspn-s1-s1-rfile (rewrite)
     (implies (and (strcspn-s1-1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                   (not (equal (get-nth i2 lst2) ch))
                   (not (equal (get-nth i2 lst2) 0))
                   (d4-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 5)))
                     (read-rn oplen rn (mc-rfile s)))))

; put together.
; case 1. s1 --> exit, when (strchr i2 n2 lst2 ch).
(prove-lemma strcspn-s1-1p-info (rewrite)
     (implies (strcspn-s1-1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
              (equal (lessp i2 n2) t)))

(prove-lemma strcspn-s1-sn (rewrite)
     (let ((sn (stepn s (strcspn-t0 i2 n2 lst2 ch))))
       (implies
        (and (strcspn-s1-1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
             (strchr i2 n2 lst2 ch))
        (and (equal (mc-status sn) 'running)
             (equal (mc-pc sn) (linked-rts-addr s))
             (equal (read-dn 32 0 sn) (sub 32 1 i1*))
             (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
             (equal (read-rn 32 15 (mc-rfile sn)) (add 32 (read-an 32 6 s) 8))
             (equal (read-mem x (mc-mem sn) k) (read-mem x (mc-mem s) k))))))
     ((induct (strcspn-induct0 s i2* i2 n2 lst2 ch))
      (disable strcspn-s1-1p read-dn)))

(prove-lemma strcspn-s1-sn-rfile (rewrite)
     (implies
      (and (strcspn-s1-1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
           (strchr i2 n2 lst2 ch)
           (d2-7a2-5p rn)
           (leq oplen 32))
```

```
      (equal (read-rn oplen rn (mc-rfile (stepn s (strcspn-t0 i2 n2 lst2 ch))))
             (if (d4-7a2-5p rn)
                 (read-rn oplen rn (mc-rfile s))
               (get-vlst oplen 0 rn '(2 3) (movem-saved s 4 8 2)))))
     ((induct (strcspn-induct0 s i2* i2 n2 lst2 ch))
      (disable strcspn-s1-1p)))

; case 2. s1 --> s0, when (not (strchr i2 n2 lst2 ch)).
(prove-lemma strcspn-s1p-s1-1p ()
     (implies (strcspn-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
              (strcspn-s1-1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)))

(prove-lemma strcspn-s1-s0 (rewrite)
     (let ((s0 (stepn s (strcspn-t0 i2 n2 lst2 ch))))
       (implies (and (strcspn-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                     (not (strchr i2 n2 lst2 ch)))
                (and (strcspn-s0p s0 i1* i1 str1 n1 lst1 str2 n2 lst2)
                     (equal (read-rn 32 14 (mc-rfile s0))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (linked-a6 s0) (linked-a6 s))
                     (equal (linked-rts-addr s0) (linked-rts-addr s))
                     (equal (movem-saved s0 4 8 2)
                            (movem-saved s 4 8 2))
                     (equal (read-mem x (mc-mem s0) k)
                            (read-mem x (mc-mem s) k)))))
     ((induct (strcspn-induct0 s i2* i2 n2 lst2 ch))
      (use (strcspn-s1p-s1-1p))
      (disable strcspn-s0p strcspn-s1p strcspn-s1-1p movem-saved linked-a6
               linked-rts-addr)))

(disable strcspn-s1-1p-info)

(prove-lemma strcspn-s1-s0-rfile (rewrite)
     (implies
      (and (strcspn-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
           (not (strchr i2 n2 lst2 ch))
           (d4-7a2-5p rn))
      (equal (read-rn oplen rn (mc-rfile (stepn s (strcspn-t0 i2 n2 lst2 ch))))
             (read-rn oplen rn (mc-rfile s))))
     ((induct (strcspn-induct0 s i2* i2 n2 lst2 ch))
      (use (strcspn-s1p-s1-1p))
      (disable strcspn-s1p strcspn-s1-1p)))

; from s0 --> exit. s0 --> sn.
; base case:  s0 --> sn, when (strchr i2 n2 lst2 ch).
(prove-lemma strcspn-s0-sn-base (rewrite)
     (let ((sn (stepn s (strcspn-t1 n2 lst2 (get-nth i1 lst1)))))
       (implies (and (strcspn-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                     (strchr 0 n2 lst2 (get-nth i1 lst1)))
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) (head i1* 32))
                     (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
```

```
                        (add 32 (read-an 32 6 s) 8))
                  (equal (read-mem x (mc-mem sn) k)
                        (read-mem x (mc-mem s) k)))))
     ((use (strcspn-s0-s1-1))
      (disable strcspn-s0p strcspn-s1-1p strchr strcspn-t0 read-dn)))

(prove-lemma strcspn-s0-sn-rfile-base (rewrite)
     (let ((ch (get-nth i1 lst1)))
       (implies
        (and (strcspn-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
             (strchr 0 n2 lst2 ch)
             (leq oplen 32)
             (d2-7a2-5p rn))
        (equal (read-rn oplen rn (mc-rfile (stepn s (strcspn-t1 n2 lst2 ch))))
               (if (d4-7a2-5p rn)
                   (read-rn oplen rn (mc-rfile s))
                 (get-vlst oplen 0 rn '(2 3) (movem-saved s 4 8 2)))))))
     ((use (strcspn-s0-s1-1))
      (disable strcspn-s0p strcspn-s1-1p strchr strcspn-t0)))

; induction case: s0 --> s0, when (not (strchr i2 n2 lst2 ch)).
(prove-lemma strcspn-s0-s0 (rewrite)
     (let ((s0 (stepn s (strcspn-t1 n2 lst2 (get-nth i1 lst1)))))
       (implies (and (strcspn-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                     (not (strchr 0 n2 lst2 (get-nth i1 lst1))))
                (and (strcspn-s0p s0 (add 32 i1* 1) (add1 i1) str1 n1 lst1
                                  str2 n2 lst2)
                     (equal (read-rn 32 14 (mc-rfile s0))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (linked-a6 s0) (linked-a6 s))
                     (equal (linked-rts-addr s0) (linked-rts-addr s))
                     (equal (movem-saved s0 4 8 2)
                            (movem-saved s 4 8 2))
                     (equal (read-mem x (mc-mem s0) k)
                            (read-mem x (mc-mem s) k)))))
       ((use (strcspn-s0-s1))
        (disable strcspn-s0p strcspn-s1p strcspn-t0 strchr)))

(prove-lemma strcspn-s0-s0-rfile (rewrite)
     (let ((ch (get-nth i1 lst1)))
       (implies
        (and (strcspn-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
             (not (strchr 0 n2 lst2 ch))
             (d4-7a2-5p rn))
        (equal (read-rn oplen rn (mc-rfile (stepn s (strcspn-t1 n2 lst2 ch))))
               (read-rn oplen rn (mc-rfile s)))))
     ((use (strcspn-s0-s1))
      (disable strcspn-s0p strcspn-s1p strcspn-t0 strchr)))

; put together: s0 --> sn.
(prove-lemma strcspn-s0p-info (rewrite)
     (implies (strcspn-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
              (and (equal (lessp i1 n1) t)
                   (equal (nat-to-uint (head i1* 32)) (fix i1)))))
```

```
(prove-lemma strcspn-s0-sn (rewrite)
     (let ((sn (stepn s (strcspn-t2 i1 n1 lst1 n2 lst2))))
       (implies (strcspn-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (uread-dn 32 0 sn)
                            (strcspn i1 n1 lst1 n2 lst2))
                     (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))
                     (equal (read-mem x (mc-mem sn) k)
                            (read-mem x (mc-mem s) k))))))
     ((induct (strcspn-induct1 s i1* i1 n1 lst1 n2 lst2))
      (disable strcspn-s0p strchr strcspn-t1 read-dn)))

(prove-lemma strcspn-s0-sn-rfile (rewrite)
     (implies
      (and (strcspn-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
           (d2-7a2-5p rn)
           (leq oplen 32))
      (equal (read-rn oplen rn
                      (mc-rfile (stepn s (strcspn-t2 i1 n1 lst1 n2 lst2))))
             (if (d4-7a2-5p rn)
                 (read-rn oplen rn (mc-rfile s))
               (get-vlst oplen 0 rn '(2 3) (movem-saved s 4 8 2)))))
     ((induct (strcspn-induct1 s i1* i1 n1 lst1 n2 lst2))
      (disable strcspn-s0p strchr strcspn-t1)))

(disable strcspn-s0p-info)

; now, finally, the correctness of strcspn.
(prove-lemma strcspn-correctness (rewrite)
     (let ((sn (stepn s (strcspn-t n1 lst1 n2 lst2))))
       (implies (strcspn-statep s str1 n1 lst1 str2 n2 lst2)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 7 s) 4))
                     (implies (and (d2-7a2-5p rn)
                                   (leq oplen 32))
                              (equal (read-rn oplen rn (mc-rfile sn))
                                     (read-rn oplen rn (mc-rfile s))))
                     (implies (disjoint x k (sub 32 12 (read-sp s)) 24)
                              (equal (read-mem x (mc-mem sn) k)
                                     (read-mem x (mc-mem s) k)))
                     (equal (uread-dn 32 0 sn) (strcspn 0 n1 lst1 n2 lst2))))))
     ((use (strcspn-s-s0))
      (disable strcspn-statep strcspn-s0p linked-rts-addr linked-a6 uread-dn)))

(disable strcspn-t)
```

```
; some properties of strcspn.
; see file cstring.events.
```

C.21 The strlen Function

```
;          Proof of the Correctness of the STRLEN Function
#|
This is part of our effort to verify the Berkeley string library.  The
Berkeley string library is widely used as part of the Berkeley Unix OS.

This is the source code of strlen function in the Berkeley string library.

/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 */
/* find the length of str[] */
size_t
strlen(str)
        const char *str;
{
        register const char *s;

        for (s = str; *s; ++s);
        return(s - str);
}

The MC68020 assembly code of the C function strlen on SUN-3 is given as
follows.  This binary is generated by "gcc -O".

0x25b0 <strlen>:        linkw fp,#0
0x25b4 <strlen+4>:      movel fp@(8),d0
0x25b8 <strlen+8>:      moveal d0,a0
0x25ba <strlen+10>:     tstb a0@
0x25bc <strlen+12>:     beq 0x25c4 <strlen+20>
0x25be <strlen+14>:     addqw #1,a0
0x25c0 <strlen+16>:     tstb a0@
0x25c2 <strlen+18>:     bne 0x25be <strlen+14>
0x25c4 <strlen+20>:     subl a0,d0
0x25c6 <strlen+22>:     negl d0
0x25c8 <strlen+24>:     unlk fp
0x25ca <strlen+26>:     rts

The machine code of the above program is:

<strlen>:    0x4e56  0x0000  0x202e  0x0008  0x2040  0x4a10  0x6706  0x5248
<strlen+16>: 0x4a10  0x66fa  0x9088  0x4480  0x4e5e  0x4e75

'(78      86      0       0       32      46      0       8
  32      64      74      16      103     6       82      72
  74      16      102     250     144     136     68      128
  78      94      78      117)
```

```
|#

; in the logic, the above program is defined by (strlen-code).
(defn strlen-code ()
  '(78      86      0       0       32      46      0       8
    32      64      74      16      103     6       82      72
    74      16      102     250     144     136     68      128
    78      94      78      117))

(constrain strlen-load (rewrite)
   (equal (strlen-loadp s)
          (and (evenp (strlen-addr))
               (numberp (strlen-addr))
               (nat-rangep (strlen-addr) 32)
               (rom-addrp (strlen-addr) (mc-mem s) 28)
               (mcode-addrp (strlen-addr) (mc-mem s) (strlen-code))))
    ((strlen-loadp (lambda (s) f))
     (strlen-addr (lambda () 1))))

(prove-lemma stepn-strlen-loadp (rewrite)
     (equal (strlen-loadp (stepn s n))
            (strlen-loadp s)))

; the computation time of the program.
(defn strlen1-t (i n lst)
  (if (lessp i n)
      (if (equal (get-nth i lst) (null))
          6
        (splus 3 (strlen1-t (add1 i) n lst)))
    0)
  ((lessp (difference n i))))

(defn strlen-t (n lst)
  (if (equal (get-nth 0 lst) 0)
      9
    (splus 6 (strlen1-t 1 n lst))))

; an induction hint.
(defn strlen-induct (s i* i n lst)
  (if (lessp i n)
      (if (equal (get-nth i lst) (null))
          t
        (strlen-induct (stepn s 3) (add 32 i* 1) (add1 i) n lst))
    t)
  ((lessp (difference n i))))

; the preconditions of the initial state.
(defn strlen-statep (s str n lst)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (mc-pc s) (mc-mem s) 28)
       (mcode-addrp (mc-pc s) (mc-mem s) (strlen-code))
       (ram-addrp (sub 32 4 (read-sp s)) (mc-mem s) 12)
       (ram-addrp str (mc-mem s) n)
```

```
       (mem-lst 1 str (mc-mem s) n lst)
       (disjoint str n (sub 32 4 (read-sp s)) 12)
       (equal str (read-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (lessp (slen 0 n lst) n)
       (numberp n)
       (uint-rangep n 32)))

; an intermediate state.
(defn strlen-s0p (s str i* i n lst)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 16 (mc-pc s)) (mc-mem s) 28)
       (mcode-addrp (sub 32 16 (mc-pc s)) (mc-mem s) (strlen-code))
       (ram-addrp (read-an 32 6 s) (mc-mem s) 12)
       (ram-addrp str (mc-mem s) n)
       (mem-lst 1 str (mc-mem s) n lst)
       (disjoint str n (read-an 32 6 s) 12)
       (equal* (read-an 32 0 s) (add 32 str i*))
       (equal str (read-dn 32 0 s))
       (equal i (nat-to-uint i*))
       (lessp (slen i n lst) n)
       (numberp i*)
       (nat-rangep i* 32)
       (numberp n)
       (uint-rangep n 32)))

; from the intial state s to exit: s --> sn.
(prove-lemma strlen-s-sn (rewrite)
     (implies (and (strlen-statep s str n lst)
                   (equal (get-nth 0 lst) 0))
              (and (equal (mc-status (stepn s 9)) 'running)
                   (equal (mc-pc (stepn s 9)) (rts-addr s))
                   (equal (nat-to-uint (read-dn 32 0 (stepn s 9))) 0)
                   (equal (read-rn 32 15 (mc-rfile (stepn s 9)))
                          (add 32 (read-an 32 7 s) 4))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 9)))
                          (read-an 32 6 s)))))

(prove-lemma strlen-s-sn-rfile (rewrite)
     (implies (and (strlen-statep s str n lst)
                   (equal (get-nth 0 lst) 0)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 9)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strlen-s-sn-mem (rewrite)
     (implies (and (strlen-statep s str n lst)
                   (equal (get-nth 0 lst) 0)
                   (disjoint x k (sub 32 4 (read-sp s)) 12))
              (equal (read-mem x (mc-mem (stepn s 9)) k)
                     (read-mem x (mc-mem s) k))))

; from the initial state s to s0: s --> s0.
(prove-lemma strlen-s-s0 ()
```

```
     (implies (and (strlen-statep s str n lst)
                   (not (equal (get-nth 0 lst) 0)))
              (strlen-s0p (stepn s 6) str 1 1 n lst)))

(prove-lemma strlen-s-s0-else (rewrite)
     (implies (and (strlen-statep s str n lst)
                   (not (equal (get-nth 0 lst) 0)))
              (and (equal (linked-rts-addr (stepn s 6)) (rts-addr s))
                   (equal (linked-a6 (stepn s 6)) (read-an 32 6 s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 6)))
                          (sub 32 4 (read-sp s))))))

(prove-lemma strlen-s-s0-rfile (rewrite)
     (implies (and (strlen-statep s str n lst)
                   (not (equal (get-nth 0 lst) 0))
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 6)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strlen-s-s0-mem (rewrite)
     (implies (and (strlen-statep s str n lst)
                   (not (equal (get-nth 0 lst) 0))
                   (disjoint x k (sub 32 4 (read-sp s)) 12))
              (equal (read-mem x (mc-mem (stepn s 6)) k)
                     (read-mem x (mc-mem s) k))))

; from s0 to exit (base case), from s0 to s0 (induction case).
; base case: s0 --> exit.
(prove-lemma strlen-s0-sn-base (rewrite)
     (implies (and (strlen-s0p s str i* i n lst)
                   (equal (get-nth i lst) 0))
              (and (equal (mc-status (stepn s 6)) 'running)
                   (equal (mc-pc (stepn s 6)) (linked-rts-addr s))
                   (equal (nat-to-uint (read-dn 32 0 (stepn s 6))) i)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 6)))
                          (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 6)))
                          (add 32 (read-an 32 6 s) 8))
                   (equal (read-mem x (mc-mem (stepn s 6)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma strlen-s0-sn-rfile-base (rewrite)
     (implies (and (strlen-s0p s str i* i n lst)
                   (d2-7a2-5p rn)
                   (equal (get-nth i lst) 0))
              (equal (read-rn oplen rn (mc-rfile (stepn s 6)))
                     (read-rn oplen rn (mc-rfile s)))))

; induction case: s0 --> s0.
(prove-lemma strlen-s0-s0 (rewrite)
     (implies (and (strlen-s0p s str i* i n lst)
                   (not (equal (get-nth i lst) 0)))
              (and (strlen-s0p (stepn s 3) str (add 32 i* 1) (add1 i) n lst)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 3)))
```

```
                       (read-rn 32 14 (mc-rfile s)))
                (equal (linked-a6 (stepn s 3)) (linked-a6 s))
                (equal (linked-rts-addr (stepn s 3)) (linked-rts-addr s))
                (equal (read-mem x (mc-mem (stepn s 3)) k)
                       (read-mem x (mc-mem s) k)))))

(prove-lemma strlen-s0-s0-rfile (rewrite)
     (implies (and (strlen-s0p s str i* i n lst)
                   (d2-7a2-5p rn)
                   (not (equal (get-nth i lst) 0)))
              (equal (read-rn oplen rn (mc-rfile (stepn s 3)))
                     (read-rn oplen rn (mc-rfile s)))))

; put together (s0 --> exit).
(prove-lemma strlen-s0p-info (rewrite)
     (implies (strlen-s0p s str i* i n lst)
              (equal (lessp i n) t)))

(prove-lemma strlen-s0-sn (rewrite)
     (let ((sn (stepn s (strlen1-t i n lst))))
       (implies (strlen-s0p s str i* i n lst)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (nat-to-uint (read-dn 32 0 sn))
                            (strlen i n lst))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))
                     (equal (read-mem x (mc-mem sn) k)
                            (read-mem x (mc-mem s) k))))))
     ((induct (strlen-induct s i* i n lst))
      (disable strlen-s0p read-dn)))

(disable strlen-s0p-info)

(prove-lemma strlen-s0-sn-rfile (rewrite)
     (implies
      (and (strlen-s0p s str i* i n lst)
           (d2-7a2-5p rn))
      (equal (read-rn oplen rn (mc-rfile (stepn s (strlen1-t i n lst))))
             (read-rn oplen rn (mc-rfile s))))
     ((induct (strlen-induct s i* i n lst))
      (disable strlen-s0p)))

; the correctness of the STRLEN program.
(prove-lemma strlen-statep-info ()
     (implies (strlen-statep s str n lst)
              (not (zerop n))))

(prove-lemma strlen-correctness (rewrite)
     (let ((sn (stepn s (strlen-t n lst))))
       (implies (strlen-statep s str n lst)
                (and (equal (mc-status sn) 'running)
```

```
                    (equal (mc-pc sn) (rts-addr s))
                    (equal (read-rn 32 14 (mc-rfile sn))
                           (read-rn 32 14 (mc-rfile s)))
                    (equal (read-rn 32 15 (mc-rfile sn))
                           (add 32 (read-sp s) 4))
                    (implies (d2-7a2-5p rn)
                             (equal (read-rn oplen rn (mc-rfile sn))
                                    (read-rn oplen rn (mc-rfile s))))
                    (implies (disjoint x k (sub 32 4 (read-sp s)) 12)
                             (equal (read-mem x (mc-mem sn) k)
                                    (read-mem x (mc-mem s) k)))
                    (equal (uread-dn 32 0 sn) (strlen 0 n lst)))))
     ((use (strlen-s-s0) (strlen-statep-info))
      (disable strlen-statep strlen-s0p linked-rts-addr linked-a6 read-dn)))

(disable strlen-t)

; some properties of strlen.
; see the file cstring.events.
```

C.22 The strncat Function

```
;          Proof of the Correctness of the STRNCAT Function
#|
This is part of our effort to verify the Berkeley string library.  The
Berkeley string library is widely used as part of the Berkeley Unix OS.

This is the source code of strncat function in the Berkeley string library.

/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 */
char *
strncat(dst, src, n)
        char *dst;
        const char *src;
        register size_t n;
{
        if (n != 0) {
                register char *d = dst;
                register const char *s = src;

                while (*d != 0)
                        d++;
                do {
                        if ((*d = *s++) == 0)
                                break;
                        d++;
                } while (--n != 0);
                *d = 0;
        }
```

```
        return (dst);
}
```

The MC68020 assembly code of the C function strncat on SUN-3 is given as follows. This binary is generated by "gcc -O".

```
0x25d0 <strncat>:       linkw fp,#0
0x25d4 <strncat+4>:     movel d2,sp@-
0x25d6 <strncat+6>:     movel fp@(8),d2
0x25da <strncat+10>:    movel fp@(16),d1
0x25de <strncat+14>:    beq 0x25fe <strncat+46>
0x25e0 <strncat+16>:    moveal d2,a0
0x25e2 <strncat+18>:    moveal fp@(12),a1
0x25e6 <strncat+22>:    tstb a0@
0x25e8 <strncat+24>:    beq 0x25f0 <strncat+32>
0x25ea <strncat+26>:    addqw #1,a0
0x25ec <strncat+28>:    tstb a0@
0x25ee <strncat+30>:    bne 0x25ea <strncat+26>
0x25f0 <strncat+32>:    moveb a1@+,d0
0x25f2 <strncat+34>:    moveb d0,a0@
0x25f4 <strncat+36>:    beq 0x25fc <strncat+44>
0x25f6 <strncat+38>:    addqw #1,a0
0x25f8 <strncat+40>:    subl #1,d1
0x25fa <strncat+42>:    bne 0x25f0 <strncat+32>
0x25fc <strncat+44>:    clrb a0@
0x25fe <strncat+46>:    movel d2,d0
0x2600 <strncat+48>:    movel fp@(-4),d2
0x2604 <strncat+52>:    unlk fp
0x2606 <strncat+54>:    rts
```

The machine code of the above program is:

```
<strncat>:     0x4e56  0x0000  0x2f02  0x242e  0x0008  0x222e  0x0010  0x671e
<strncat+16>:  0x2042  0x226e  0x000c  0x4a10  0x6706  0x5248  0x4a10  0x66fa
<strncat+32>:  0x1019  0x1080  0x6706  0x5248  0x5381  0x66f4  0x4210  0x2002
<strncat+48>:  0x242e  0xfffc  0x4e5e  0x4e75
```

```
'(78      86      0       0       47      2       36      46
  0       8       34      46      0       16      103     30
  32      66      34      110     0       12      74      16
  103     6       82      72      74      16      102     250
  16      25      16      128     103     6       82      72
  83      129     102     244     66      16      32      2
  36      46      255     252     78      94      78      117)
|#
```

```
; in the logic, the above program is defined by (strncat-code).
(defn strncat-code ()
  '(78      86      0       0       47      2       36      46
    0       8       34      46      0       16      103     30
    32      66      34      110     0       12      74      16
    103     6       82      72      74      16      102     250
    16      25      16      128     103     6       82      72
    83      129     102     244     66      16      32      2
```

```
    36      46      255     252     78      94      78      117))

; the computation time of the program.
(defn strncat-t0 (i n1 lst1)
  (if (lessp i n1)
      (if (equal (get-nth i lst1) 0)
          2
        (splus 3 (strncat-t0 (add1 i) n1 lst1)))
    0)
  ((lessp (difference n1 i))))

(defn strncat-t1 (n1 lst1)
  (splus 10 (strncat-t0 1 n1 lst1)))

(defn strncat-t2 (j n lst2)
  (if (equal (get-nth j lst2) 0)
      8
    (if (equal (sub1 n) 0)
        11
      (splus 6 (strncat-t2 (add1 j) (sub1 n) lst2)))))

(defn strncat-t (n1 lst1 n lst2)
  (if (equal n 0)
      9
    (if (equal (get-nth 0 lst1) 0)
        (splus 9 (strncat-t2 0 n lst2))
      (splus (strncat-t1 n1 lst1) (strncat-t2 0 n lst2)))))

; two induction hints.
(defn strncat-induct0 (s i* i n1 lst1)
  (if (lessp i n1)
      (if (equal (get-nth i lst1) 0)
          t
        (strncat-induct0 (stepn s 3) (add 32 i* 1) (add1 i) n1 lst1))
    t)
  ((lessp (difference n1 i))))

(defn strncat-induct1 (s i* i lst1 j* j n lst2)
  (if (equal (get-nth j lst2) 0)
      t
    (if (equal (sub1 n) 0)
        t
      (strncat-induct1 (stepn s 6) (add 32 i* 1) (add1 i)
                       (put-nth (get-nth j lst2) i lst1) (add 32 j* 1) (add1 j)
                       (sub1 n) lst2))))

; the preconditions of the initial state.
(defn strncat-statep (s str1 n1 lst1 str2 n lst2)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (mc-pc s) (mc-mem s) 56)
       (mcode-addrp (mc-pc s) (mc-mem s) (strncat-code))
       (ram-addrp (sub 32 8 (read-sp s)) (mc-mem s) 24)
       (ram-addrp str1 (mc-mem s) n1)
```

```
      (mem-lst 1 str1 (mc-mem s) n1 lst1)
      (ram-addrp str2 (mc-mem s) n)
      (mem-lst 1 str2 (mc-mem s) n lst2)
      (disjoint (sub 32 8 (read-sp s)) 24 str1 n1)
      (disjoint (sub 32 8 (read-sp s)) 24 str2 n)
      (disjoint str1 n1 str2 n)
      (equal str1 (read-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
      (equal str2 (read-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
      (equal n (uread-mem (add 32 (read-sp s) 12) (mc-mem s) 4))
      (lessp (add1 (plus (slen 0 n1 lst1) n)) n1)
      (numberp n1)
      (uint-rangep n1 32)))

; an intermediate state s0.
(defn strncat-s0p (s i* i str1 n1 lst1 str2 n lst2)
  (and (equal (mc-status s) 'running)
      (evenp (mc-pc s))
      (rom-addrp (sub 32 28 (mc-pc s)) (mc-mem s) 56)
      (mcode-addrp (sub 32 28 (mc-pc s)) (mc-mem s) (strncat-code))
      (ram-addrp (sub 32 4 (read-an 32 6 s)) (mc-mem s) 24)
      (ram-addrp str1 (mc-mem s) n1)
      (mem-lst 1 str1 (mc-mem s) n1 lst1)
      (ram-addrp str2 (mc-mem s) n)
      (mem-lst 1 str2 (mc-mem s) n lst2)
      (disjoint (sub 32 4 (read-an 32 6 s)) 24 str1 n1)
      (disjoint (sub 32 4 (read-an 32 6 s)) 24 str2 n)
      (disjoint str1 n1 str2 n)
      (equal* (read-an 32 0 s) (add 32 str1 i*))
      (equal str1 (read-dn 32 2 s))
      (equal str2 (read-an 32 1 s))
      (equal n (nat-to-uint (read-dn 32 1 s)))
      (numberp i*)
      (nat-rangep i* 32)
      (equal i (nat-to-uint i*))
      (lessp (add1 (plus (slen i n1 lst1) n)) n1)
      (numberp n1)
      (uint-rangep n1 32)
      (not (equal n 0))))

; an intermediate state s1.
(defn strncat-s1p (s i* i str1 n1 lst1 j* j str2 n lst2 i_ n_)
  (and (equal (mc-status s) 'running)
      (evenp (mc-pc s))
      (rom-addrp (sub 32 32 (mc-pc s)) (mc-mem s) 56)
      (mcode-addrp (sub 32 32 (mc-pc s)) (mc-mem s) (strncat-code))
      (ram-addrp (sub 32 4 (read-an 32 6 s)) (mc-mem s) 24)
      (ram-addrp str1 (mc-mem s) n1)
      (mem-lst 1 str1 (mc-mem s) n1 lst1)
      (ram-addrp str2 (mc-mem s) n_)
      (mem-lst 1 str2 (mc-mem s) n_ lst2)
      (disjoint (sub 32 4 (read-an 32 6 s)) 24 str1 n1)
      (disjoint (sub 32 4 (read-an 32 6 s)) 24 str2 n_)
      (disjoint str1 n1 str2 n_)
      (equal* (read-an 32 0 s) (add 32 str1 i*))
```

```
       (equal* (read-an 32 1 s) (add 32 str2 j*))
       (equal str1 (read-dn 32 2 s))
       (equal n (nat-to-uint (read-dn 32 1 s)))
       (leq (plus j n) n_)
       (lessp (add1 (plus i_ n_)) n1)
       (leq i (plus i_ j))
       (not (equal n 0))
       (numberp i*)
       (nat-rangep i* 32)
       (equal i (nat-to-uint i*))
       (numberp j*)
       (nat-rangep j* 32)
       (equal j (nat-to-uint j*))
       (lessp n1 4294967296)
       (numberp n_)
       (lessp n_ 4294967296)))

; from the initial state s to exit: s --> sn, when n = 0.
(prove-lemma strncat-s-sn (rewrite)
     (implies (and (strncat-statep s str1 n1 lst1 str2 n lst2)
                   (equal n 0))
              (and (equal (mc-status (stepn s 9)) 'running)
                   (equal (mc-pc (stepn s 9)) (rts-addr s))
                   (equal (read-dn 32 0 (stepn s 9)) str1)
                   (mem-lst 1 str1 (mc-mem (stepn s 9)) n1 lst1)
                   (equal (read-rn 32 15 (mc-rfile (stepn s 9)))
                          (add 32 (read-an 32 7 s) 4))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 9)))
                          (read-an 32 6 s)))))

(prove-lemma strncat-s-sn-rfile (rewrite)
     (implies (and (strncat-statep s str1 n1 lst1 str2 n lst2)
                   (equal n 0)
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 9)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strncat-s-sn-mem (rewrite)
     (implies (and (strncat-statep s str1 n1 lst1 str2 n lst2)
                   (equal n 0)
                   (disjoint x k (sub 32 8 (read-sp s)) 24))
              (equal (read-mem x (mc-mem (stepn s 9)) k)
                     (read-mem x (mc-mem s) k))))

; from the initial state s to s0: s --> s0, when n =- 0, lst1[0] =- 0.
(prove-lemma strncat-s-s0 ()
     (implies (and (strncat-statep s str1 n1 lst1 str2 n lst2)
                   (not (equal n 0))
                   (not (equal (get-nth 0 lst1) 0)))
              (and (strncat-s0p (stepn s 10) 1 1 str1 n1 lst1 str2 n lst2)
                   (equal (linked-rts-addr (stepn s 10)) (rts-addr s))
                   (equal (linked-a6 (stepn s 10)) (read-an 32 6 s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 10)))
```

```
                       (sub 32 4 (read-sp s)))
                (equal (rn-saved (stepn s 10)) (read-dn 32 2 s)))))

(prove-lemma strncat-s-s0-rfile (rewrite)
     (implies (and (strncat-statep s str1 n1 lst1 str2 n lst2)
                   (not (equal n 0))
                   (not (equal (get-nth 0 lst1) 0))
                   (d3-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 10)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strncat-s-s0-mem (rewrite)
     (implies (and (strncat-statep s str1 n1 lst1 str2 n lst2)
                   (not (equal n 0))
                   (not (equal (get-nth 0 lst1) 0))
                   (disjoint x k (sub 32 8 (read-sp s)) 24))
              (equal (read-mem x (mc-mem (stepn s 10)) k)
                     (read-mem x (mc-mem s) k))))

; from initial state s to s1, when n =- 0, lst1[0] = 0.
(prove-lemma strncat-s-s1 ()
     (implies (and (strncat-statep s str1 n1 lst1 str2 n lst2)
                   (not (equal n 0))
                   (equal (get-nth 0 lst1) 0))
              (strncat-s1p (stepn s 9)
                           0 0 str1 n1 lst1 0 0 str2 n lst2 0 n)))

(prove-lemma strncat-s-s1-else (rewrite)
     (implies (and (strncat-statep s str1 n1 lst1 str2 n lst2)
                   (not (equal n 0))
                   (equal (get-nth 0 lst1) 0))
              (and (equal (linked-rts-addr (stepn s 9)) (rts-addr s))
                   (equal (linked-a6 (stepn s 9)) (read-an 32 6 s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 9)))
                          (sub 32 4 (read-sp s)))
                   (equal (rn-saved (stepn s 9))
                          (read-rn 32 2 (mc-rfile s))))))

(prove-lemma strncat-s-s1-rfile (rewrite)
     (implies (and (strncat-statep s str1 n1 lst1 str2 n lst2)
                   (not (equal n 0))
                   (equal (get-nth 0 lst1) 0)
                   (d3-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 9)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strncat-s-s1-mem (rewrite)
     (implies (and (strncat-statep s str1 n1 lst1 str2 n lst2)
                   (not (equal n 0))
                   (equal (get-nth 0 lst1) 0)
                   (disjoint x k (sub 32 8 (read-sp s)) 24))
              (equal (read-mem x (mc-mem (stepn s 9)) k)
                     (read-mem x (mc-mem s) k))))
```

```
; from s0 to s1: s0 --> s1.
; base case:  s0 --> s1, when lst1[i] = 0.
(prove-lemma strncat-s0-s1-base (rewrite)
     (implies (and (strncat-s0p s i* i str1 n1 lst1 str2 n lst2)
                   (equal (get-nth i lst1) 0))
              (and (strncat-s1p (stepn s 2) i* i str1 n1 lst1
                                0 0 str2 n lst2 i n)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 2)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (linked-a6 (stepn s 2)) (linked-a6 s))
                   (equal (linked-rts-addr (stepn s 2)) (linked-rts-addr s))
                   (equal (rn-saved (stepn s 2)) (rn-saved s))
                   (equal (read-mem x (mc-mem (stepn s 2)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma strncat-s0-s1-rfile-base (rewrite)
     (implies (and (strncat-s0p s i* i str1 n1 lst1 str2 n lst2)
                   (equal (get-nth i lst1) 0)
                   (d3-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 2)))
                     (read-rn oplen rn (mc-rfile s)))))

; induction case: s0 --> s0, when lst1[i] =~ 0.
(prove-lemma strncat-s0-s0 (rewrite)
     (implies (and (strncat-s0p s i* i str1 n1 lst1 str2 n lst2)
                   (not (equal (get-nth i lst1) 0)))
              (and (strncat-s0p (stepn s 3) (add 32 i* 1) (add1 i)
                                str1 n1 lst1 str2 n lst2)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 3)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (linked-a6 (stepn s 3)) (linked-a6 s))
                   (equal (linked-rts-addr (stepn s 3))
                          (linked-rts-addr s))
                   (equal (rn-saved (stepn s 3)) (rn-saved s))
                   (equal (read-mem x (mc-mem (stepn s 3)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma strncat-s0-s0-rfile (rewrite)
     (implies (and (strncat-s0p s i* i str1 n1 lst1 str2 n lst2)
                   (not (equal (get-nth i lst1) 0))
                   (d3-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 3)))
                     (read-rn oplen rn (mc-rfile s)))))

; put together. s0 --> s1.
(prove-lemma strncat-s0p-info (rewrite)
     (implies (strncat-s0p s i* i str1 n1 lst1 str2 n lst2)
              (equal (lessp i n1) t)))

(prove-lemma strncat-s0-s1 (rewrite)
     (implies
      (strncat-s0p s i* i str1 n1 lst1 str2 n lst2)
      (and (strncat-s1p (stepn s (strncat-t0 i n1 lst1)) (strlen* i* i n1 lst1)
                        (strlen i n1 lst1) str1 n1 lst1 0 0 str2 n lst2
```

```
                    (strlen i n1 lst1) n)
          (equal (read-rn 32 14 (mc-rfile (stepn s (strncat-t0 i n1 lst1))))
                 (read-rn 32 14 (mc-rfile s)))
          (equal (linked-a6 (stepn s (strncat-t0 i n1 lst1))) (linked-a6 s))
          (equal (linked-rts-addr (stepn s (strncat-t0 i n1 lst1)))
                 (linked-rts-addr s))
          (equal (rn-saved (stepn s (strncat-t0 i n1 lst1))) (rn-saved s))
          (equal (read-mem x (mc-mem (stepn s (strncat-t0 i n1 lst1))) k)
                 (read-mem x (mc-mem s) k))))
     ((induct (strncat-induct0 s i* i n1 lst1))
      (disable strncat-s0p strncat-s1p)))

(disable strncat-s0p-info)

(prove-lemma strncat-s0-s1-rfile (rewrite)
     (implies
      (and (strncat-s0p s i* i str1 n1 lst1 str2 n lst2)
           (d3-7a2-5p rn))
      (equal (read-rn oplen rn (mc-rfile (stepn s (strncat-t0 i n1 lst1))))
             (read-rn oplen rn (mc-rfile s))))
     ((induct (strncat-induct0 s i* i n1 lst1))
      (disable strncat-s0p)))

; put together: s --> s1.
(prove-lemma strncat-s-s1-1 ()
     (implies
      (and (strncat-statep s str1 n1 lst1 str2 n lst2)
           (not (equal n 0))
           (not (equal (get-nth 0 lst1) 0)))
      (strncat-s1p (stepn s (strncat-t1 n1 lst1)) (strlen* 1 1 n1 lst1)
                   (strlen 1 n1 lst1) str1 n1 lst1 0 0 str2 n lst2
                   (strlen 1 n1 lst1) n))
     ((use (strncat-s-s0))
      (disable strncat-statep strncat-s0p strncat-s1p strncat-t0
               strlen* strlen)))

(prove-lemma strncat-s-s1-else-1 (rewrite)
     (let ((s1 (stepn s (strncat-t1 n1 lst1))))
       (implies (and (strncat-statep s str1 n1 lst1 str2 n lst2)
                     (not (equal n 0))
                     (not (equal (get-nth 0 lst1) 0)))
                (and (equal (linked-rts-addr s1) (rts-addr s))
                     (equal (linked-a6 s1) (read-an 32 6 s))
                     (equal (read-rn 32 14 (mc-rfile s1))
                            (sub 32 4 (read-sp s)))
                     (equal (rn-saved s1) (read-rn 32 2 (mc-rfile s)))))))
     ((use (strncat-s-s0))
      (disable strncat-statep strncat-s0p strncat-s1p strncat-t0
               strlen* strlen)))

(prove-lemma strncat-s-s1-rfile-1 (rewrite)
     (implies
      (and (strncat-statep s str1 n1 lst1 str2 n lst2)
           (not (equal n 0))
```

```
          (not (equal (get-nth 0 lst1) 0))
          (d3-7a2-5p rn))
     (equal (read-rn oplen rn (mc-rfile (stepn s (strncat-t1 n1 lst1))))
            (read-rn oplen rn (mc-rfile s))))
    ((use (strncat-s-s0))
     (disable strncat-statep strncat-s0p strncat-t0)))

(prove-lemma strncat-s-s1-mem-1 (rewrite)
    (implies
     (and (strncat-statep s str1 n1 lst1 str2 n lst2)
          (not (equal n 0))
          (not (equal (get-nth 0 lst1) 0))
          (disjoint x k (sub 32 8 (read-sp s)) 24))
     (equal (read-mem x (mc-mem (stepn s (strncat-t1 n1 lst1))) k)
            (read-mem x (mc-mem s) k)))
    ((use (strncat-s-s0))
     (disable strncat-statep strncat-s0p strncat-t0)))

; from s1 to exit: s1 --> sn.  By induction.
; base case 1: s1 --> sn, when lst2[j] = 0.
(prove-lemma strncat-s1-sn-base1 (rewrite)
    (implies (and (strncat-s1p s i* i str1 n1 lst1 j* j str2 n lst2 i_ n_)
                  (equal (get-nth j lst2) 0))
             (and (equal (mc-status (stepn s 8)) 'running)
                  (equal (mc-pc (stepn s 8)) (linked-rts-addr s))
                  (equal (read-dn 32 0 (stepn s 8)) str1)
                  (mem-lst 1 str1 (mc-mem (stepn s 8)) n1 (put-nth 0 i lst1))
                  (equal (read-rn 32 14 (mc-rfile (stepn s 8)))
                         (linked-a6 s))
                  (equal (read-rn 32 15 (mc-rfile (stepn s 8)))
                         (add 32 (read-an 32 6 s) 8)))))

(prove-lemma strncat-s0-sn-rfile-base1 (rewrite)
    (implies (and (strncat-s1p s i* i str1 n1 lst1 j* j str2 n lst2 i_ n_)
                  (equal (get-nth j lst2) 0)
                  (leq oplen 32)
                  (d2-7a2-5p rn))
             (equal (read-rn oplen rn (mc-rfile (stepn s 8)))
                    (if (d3-7a2-5p rn)
                        (read-rn oplen rn (mc-rfile s))
                      (head (rn-saved s) oplen)))))

(prove-lemma strncat-s1-sn-mem-base1 (rewrite)
    (implies (and (strncat-s1p s i* i str1 n1 lst1 j* j str2 n lst2 i_ n_)
                  (equal (get-nth j lst2) 0)
                  (disjoint x k str1 n1))
             (equal (read-mem x (mc-mem (stepn s 8)) k)
                    (read-mem x (mc-mem s) k))))

; base case 2: s1 --> sn, when lst2[j] =- 0, n-1 = 0.
(prove-lemma strncat-s1-sn-base2 (rewrite)
    (implies (and (strncat-s1p s i* i str1 n1 lst1 j* j str2 n lst2 i_ n_)
                  (not (equal (get-nth j lst2) 0))
                  (equal (sub1 n) 0))
```

```
             (and (equal (mc-status (stepn s 11)) 'running)
                  (equal (mc-pc (stepn s 11)) (linked-rts-addr s))
                  (equal (read-dn 32 0 (stepn s 11)) str1)
                  (mem-lst 1 str1 (mc-mem (stepn s 11)) n1
                           (put-nth 0 (add1 i)
                                    (put-nth (get-nth j lst2) i lst1)))
                  (equal (read-rn 32 14 (mc-rfile (stepn s 11)))
                         (linked-a6 s))
                  (equal (read-rn 32 15 (mc-rfile (stepn s 11)))
                         (add 32 (read-an 32 6 s) 8)))))

(prove-lemma strncat-s1-sn-rfile-base2 (rewrite)
     (implies (and (strncat-s1p s i* i str1 n1 lst1 j* j str2 n lst2 i_ n_)
                   (not (equal (get-nth j lst2) 0))
                   (equal (sub1 n) 0)
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 11)))
                     (if (d3-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (head (rn-saved s) oplen)))))

(prove-lemma strncat-s1-sn-mem-base2 (rewrite)
     (implies (and (strncat-s1p s i* i str1 n1 lst1 j* j str2 n lst2 i_ n_)
                   (not (equal (get-nth j lst2) 0))
                   (equal (sub1 n) 0)
                   (disjoint x k str1 n1))
              (equal (read-mem x (mc-mem (stepn s 11)) k)
                     (read-mem x (mc-mem s) k))))

; induction case: s1 --> s1, when lst2[j] =- 0, n-1 =- 0.
(prove-lemma strncat-s1-s1 (rewrite)
     (implies (and (strncat-s1p s i* i str1 n1 lst1 j* j str2 n lst2 i_ n_)
                   (not (equal (get-nth j lst2) 0))
                   (not (equal (sub1 n) 0)))
              (and (strncat-s1p (stepn s 6) (add 32 i* 1) (add1 i) str1 n1
                                (put-nth (get-nth j lst2) i lst1)
                                (add 32 j* 1) (add1 j) str2 (sub1 n) lst2
                                i_ n_)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 6)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (linked-a6 (stepn s 6)) (linked-a6 s))
                   (equal (linked-rts-addr (stepn s 6))
                          (linked-rts-addr s))
                   (equal (rn-saved (stepn s 6)) (rn-saved s)))))

(prove-lemma strncat-s1-s1-rfile (rewrite)
     (implies (and (strncat-s1p s i* i str1 n1 lst1 j* j str2 n lst2 i_ n_)
                   (not (equal (get-nth j lst2) 0))
                   (not (equal (sub1 n) 0))
                   (d3-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 6)))
                     (read-rn oplen rn (mc-rfile s)))))
```

```
(prove-lemma strncat-s1-s1-mem (rewrite)
     (implies (and (strncat-s1p s i* i str1 n1 lst1 j* j str2 n lst2 i_ n_)
                   (not (equal (get-nth j lst2) 0))
                   (not (equal (sub1 n) 0))
                   (disjoint x k str1 n1))
              (equal (read-mem x (mc-mem (stepn s 6)) k)
                     (read-mem x (mc-mem s) k))))

; put together.  s1 --> sn.
(prove-lemma strncat-s1-sn (rewrite)
     (let ((sn (stepn s (strncat-t2 j n lst2))))
       (implies (strncat-s1p s i* i str1 n1 lst1 j* j str2 n lst2 i_ n_)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) str1)
                     (mem-lst 1 str1 (mc-mem sn) n1 (strcpy2 i lst1 j n lst2))
                     (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))))))
     ((induct (strncat-induct1 s i* i lst1 j* j n lst2))
      (disable strncat-s1p read-dn)))

(prove-lemma strncat-s1-sn-rfile (rewrite)
     (implies
      (and (strncat-s1p s i* i str1 n1 lst1 j* j str2 n lst2 i_ n_)
           (leq oplen 32)
           (d2-7a2-5p rn))
      (equal (read-rn oplen rn (mc-rfile (stepn s (strncat-t2 j n lst2))))
             (if (d3-7a2-5p rn)
                 (read-rn oplen rn (mc-rfile s))
               (head (rn-saved s) oplen))))
     ((induct (strncat-induct1 s i* i lst1 j* j n lst2))
      (disable strncat-s1p)))

(prove-lemma strncat-s1-sn-mem (rewrite)
     (implies (and (strncat-s1p s i* i str1 n1 lst1 j* j str2 n lst2 i_ n_)
                   (disjoint x k str1 n1))
              (equal (read-mem x (mc-mem (stepn s (strncat-t2 j n lst2))) k)
                     (read-mem x (mc-mem s) k)))
     ((induct (strncat-induct1 s i* i lst1 j* j n lst2))
      (disable strncat-s1p)))

; the correctness of strncat.
(prove-lemma strncat-correctness (rewrite)
     (let ((sn (stepn s (strncat-t n1 lst1 n lst2))))
       (implies (strncat-statep s str1 n1 lst1 str2 n lst2)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-an 32 6 sn) (read-an 32 6 s))
                     (equal (read-an 32 7 sn) (add 32 (read-an 32 7 s) 4))
                     (implies (and (d2-7a2-5p rn)
                                   (leq oplen 32))
                              (equal (read-rn oplen rn (mc-rfile sn))
                                     (read-rn oplen rn (mc-rfile s))))
```

```
                    (implies (and (disjoint x k (sub 32 8 (read-sp s)) 24)
                                  (disjoint x k str1 n1))
                             (equal (read-mem x (mc-mem sn) k)
                                    (read-mem x (mc-mem s) k)))
                    (equal (read-dn 32 0 sn) str1)
                    (mem-lst 1 str1 (mc-mem sn) n1
                             (strncat n1 lst1 n lst2)))))
     ((use (strncat-s-s1) (strncat-s-s1-1))
      (disable strncat-statep strncat-s1p linked-rts-addr linked-a6
               strncat-t1 strncat-t2 read-dn strlen strlen*)))

(disable strncat-t)

; some properties of strncat.
; see file cstring.events.
```

C.23 The strncmp Function

```
;          Proof of the Correctness of the STRNCMP Function
#|
This is part of our effort to verify the Berkeley string library.  The
Berkeley string library is widely used as part of the Berkeley Unix OS.

This is the source code of strncmp function in the Berkeley string library.

/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 */
/* compare at most char s1[] to char s2[] */
int
strncmp(s1, s2, n)
        register const char *s1, *s2;
        register size_t n;
{

        if (n == 0)
                return (0);
        do {
                if (*s1 != *s2++)
                        return (*(unsigned char *)s1 - *(unsigned char *)--s2);
                if (*s1++ == 0)
                        break;
        } while (--n != 0);
        return (0);
}

The MC68020 assembly code of the C function strncmp on SUN-3 is given as
follows.  This binary is generated by "gcc -O".

0x2608 <strncmp>:       linkw fp,#0
0x260c <strncmp+4>:     moveml d2-d3,sp@-
```

```
0x2610 <strncmp+8>:     moveal fp@(8),a0
0x2614 <strncmp+12>:    moveal fp@(12),a1
0x2618 <strncmp+16>:    movel fp@(16),d0
0x261c <strncmp+20>:    beq 0x2642 <strncmp+58>
0x261e <strncmp+22>:    andil #255,d1
0x2624 <strncmp+28>:    andil #255,d2
0x262a <strncmp+34>:    moveb a0@,d3
0x262c <strncmp+36>:    cmpb a1@+,d3
0x262e <strncmp+38>:    beq 0x263a <strncmp+50>
0x2630 <strncmp+40>:    moveb a0@,d1
0x2632 <strncmp+42>:    moveb a1@-,d2
0x2634 <strncmp+44>:    movel d1,d0
0x2636 <strncmp+46>:    subl d2,d0
0x2638 <strncmp+48>:    bra 0x2644 <strncmp+60>
0x263a <strncmp+50>:    tstb a0@+
0x263c <strncmp+52>:    beq 0x2642 <strncmp+58>
0x263e <strncmp+54>:    subl #1,d0
0x2640 <strncmp+56>:    bne 0x262a <strncmp+34>
0x2642 <strncmp+58>:    clrl d0
0x2644 <strncmp+60>:    moveml fp@(-8),d2-d3
0x264a <strncmp+66>:    unlk fp
0x264c <strncmp+68>:    rts

The machine code of the above program is:

<strncmp>:     0x4e56  0x0000  0x48e7  0x3000  0x206e  0x0008  0x226e  0x000c
<strncmp+16>:  0x202e  0x0010  0x6724  0x0281  0x0000  0x00ff  0x0282  0x0000
<strncmp+32>:  0x00ff  0x1610  0xb619  0x670a  0x1210  0x1421  0x2001  0x9082
<strncmp+48>:  0x600a  0x4a18  0x6704  0x5380  0x66e8  0x4280  0x4cee  0x000c
<strncmp+64>:  0xfff8  0x4e5e  0x4e75

'(78      86      0       0       72      231     48      0
  32      110     0       8       34      110     0       12
  32      46      0       16      103     36      2       129
  0       0       0       255     2       130     0       0
  0       255     22      16      182     25      103     10
  18      16      20      33      32      1       144     130
  96      10      74      24      103     4       83      128
  102     232     66      128     76      238     0       12
  255     248     78      94      78      117)
|#

; in the logic, the above program is defined by (strncmp-code).
(defn strncmp-code ()
  '(78      86      0       0       72      231     48      0
    32      110     0       8       34      110     0       12
    32      46      0       16      103     36      2       129
    0       0       0       255     2       130     0       0
    0       255     22      16      182     25      103     10
    18      16      20      33      32      1       144     130
    96      10      74      24      103     4       83      128
    102     232     66      128     76      238     0       12
    255     248     78      94      78      117))
```

```
(constrain strncmp-load (rewrite)
   (equal (strncmp-loadp s)
          (and (evenp (strncmp-addr))
               (numberp (strncmp-addr))
               (nat-rangep (strncmp-addr) 32)
               (rom-addrp (strncmp-addr) (mc-mem s) 70)
               (mcode-addrp (strncmp-addr) (mc-mem s) (strncmp-code))))
   ((strncmp-loadp (lambda (s) f))
    (strncmp-addr (lambda () 1))))

(prove-lemma stepn-strncmp-loadp (rewrite)
     (equal (strncmp-loadp (stepn s n))
            (strncmp-loadp s)))

; the computation time of the program.
(defn strncmp1-t (i n lst1 lst2)
  (if (equal (get-nth i lst1) (get-nth i lst2))
      (if (equal (get-nth i lst1) 0)
          9
        (if (equal (sub1 n) 0)
            11
          (splus 7 (strncmp1-t (add1 i) (sub1 n) lst1 lst2))))
    11))

(defn strncmp-t (n lst1 lst2)
  (if (zerop n)
      10
    (splus 8 (strncmp1-t 0 n lst1 lst2))))

; an induction hint.
(defn strncmp-induct (s i* i n lst1 lst2)
  (if (equal (get-nth i lst1) (get-nth i lst2))
      (if (equal (get-nth i lst1) 0)
          t
        (if (equal (sub1 n) 0)
            t
          (strncmp-induct (stepn s 7) (add 32 i* 1) (add1 i) (sub1 n)
                          lst1 lst2)))
    t))

; the preconditions of the initial state.
(defn strncmp-statep (s str1 n1 lst1 str2 n2 lst2 n)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (mc-pc s) (mc-mem s) 70)
       (mcode-addrp (mc-pc s) (mc-mem s) (strncmp-code))
       (ram-addrp (sub 32 12 (read-sp s)) (mc-mem s) 28)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (lessp (slen 0 n lst1) n1)
       (lessp (slen 0 n lst2) n2)
       (disjoint (sub 32 12 (read-sp s)) 28 str1 n1)
```

```
       (disjoint (sub 32 12 (read-sp s)) 28 str2 n2)
       (equal str1 (read-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal str2 (read-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
       (equal n (uread-mem (add 32 (read-sp s) 12) (mc-mem s) 4))
       (numberp n1)
       (numberp n2)))

; an intermediate state.
(defn strncmp-s0p (s i* i str1 n1 lst1 str2 n2 lst2 n n_)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 34 (mc-pc s)) (mc-mem s) 70)
       (mcode-addrp (sub 32 34 (mc-pc s)) (mc-mem s) (strncmp-code))
       (ram-addrp (sub 32 8 (read-an 32 6 s)) (mc-mem s) 28)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (lessp (slen i n_ lst1) n1)
       (lessp (slen i n_ lst2) n2)
       (disjoint (sub 32 8 (read-an 32 6 s)) 28 str1 n1)
       (disjoint (sub 32 8 (read-an 32 6 s)) 28 str2 n2)
       (equal* (read-an 32 0 s) (add 32 str1 i*))
       (equal* (read-an 32 1 s) (add 32 str2 i*))
       (nat-rangep (read-rn 32 1 (mc-rfile s)) 8)
       (nat-rangep (read-rn 32 2 (mc-rfile s)) 8)
       (equal n (nat-to-uint (read-dn 32 0 s)))
       (numberp i*)
       (nat-rangep i* 32)
       (equal i (nat-to-uint i*))
       (numberp n_)
       (leq (plus i n) n_)
       (not (equal n 0))
       (uint-rangep n_ 32)))

; from the initial state s to exit: s --> sn, when n = 0.
(prove-lemma strncmp-s-sn (rewrite)
     (implies (and (strncmp-statep s str1 n1 lst1 str2 n2 lst2 n)
                   (zerop n))
              (and (equal (mc-status (stepn s 10)) 'running)
                   (equal (mc-pc (stepn s 10)) (rts-addr s))
                   (equal (iread-dn 32 0 (stepn s 10)) 0)
                   (equal (read-rn 32 15 (mc-rfile (stepn s 10)))
                          (add 32 (read-an 32 7 s) 4))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 10)))
                          (read-an 32 6 s)))))

(prove-lemma strncmp-s-sn-rfile (rewrite)
     (implies (and (strncmp-statep s str1 n1 lst1 str2 n2 lst2 n)
                   (zerop n)
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 10)))
                     (read-rn oplen rn (mc-rfile s)))))
```

```
(prove-lemma strncmp-s-sn-mem (rewrite)
     (implies (and (strncmp-statep s str1 n1 lst1 str2 n2 lst2 n)
                   (zerop n)
                   (disjoint x k (sub 32 12 (read-sp s)) 28))
              (equal (read-mem x (mc-mem (stepn s 10)) k)
                     (read-mem x (mc-mem s) k))))

; from the initial state s to s0: s --> s0.
(prove-lemma strncmp-s-s0 ()
     (implies (and (strncmp-statep s str1 n1 lst1 str2 n2 lst2 n)
                   (not (zerop n)))
              (strncmp-s0p (stepn s 8) 0 0 str1 n1 lst1 str2 n2 lst2 n n)))

(prove-lemma strncmp-s-s0-else (rewrite)
     (implies (and (strncmp-statep s str1 n1 lst1 str2 n2 lst2 n)
                   (not (zerop n)))
              (and (equal (linked-rts-addr (stepn s 8)) (rts-addr s))
                   (equal (linked-a6 (stepn s 8)) (read-an 32 6 s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 8)))
                          (sub 32 4 (read-sp s)))
                   (equal (movem-saved (stepn s 8) 4 8 2)
                          (readm-rn 32 '(2 3) (mc-rfile s))))))

(prove-lemma strncmp-s-s0-rfile (rewrite)
     (implies (and (strncmp-statep s str1 n1 lst1 str2 n2 lst2 n)
                   (not (zerop n))
                   (d4-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 8)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strncmp-s-s0-mem (rewrite)
     (implies (and (strncmp-statep s str1 n1 lst1 str2 n2 lst2 n)
                   (not (zerop n))
                   (disjoint x k (sub 32 12 (read-sp s)) 28))
              (equal (read-mem x (mc-mem (stepn s 8)) k)
                     (read-mem x (mc-mem s) k))))

; from s0 to exit: s0 --> sn.
; base case 1:  s0 --> sn, when lst1[i] =- lst2[i].
(prove-lemma strncmp-s0-sn-base1 (rewrite)
     (implies (and (strncmp-s0p s i* i str1 n1 lst1 str2 n2 lst2 n n_)
                   (not (equal (get-nth i lst1) (get-nth i lst2))))
              (and (equal (mc-status (stepn s 11)) 'running)
                   (equal (mc-pc (stepn s 11)) (linked-rts-addr s))
                   (equal (iread-dn 32 0 (stepn s 11))
                          (idifference (get-nth i lst1) (get-nth i lst2)))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 11)))
                          (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 11)))
                          (add 32 (read-an 32 6 s) 8))
                   (equal (read-mem x (mc-mem (stepn s 11)) k)
                          (read-mem x (mc-mem s) k)))))
```

```
(prove-lemma strncmp-s0-sn-rfile-base1 (rewrite)
     (implies (and (strncmp-s0p s i* i str1 n1 lst1 str2 n2 lst2 n n_)
                   (not (equal (get-nth i lst1) (get-nth i lst2)))
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 11)))
                     (if (d4-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (get-vlst oplen 0 rn '(2 3) (movem-saved s 4 8 2))))))

; base case 2: s0 --> sn, when lst[i] = lst2[i] and lst[i] = 0.
(prove-lemma strncmp-s0-sn-base2 (rewrite)
     (implies (and (strncmp-s0p s i* i str1 n1 lst1 str2 n2 lst2 n n_)
                   (equal (get-nth i lst1) (get-nth i lst2))
                   (equal (get-nth i lst1) 0))
              (and (equal (mc-status (stepn s 9)) 'running)
                   (equal (mc-pc (stepn s 9)) (linked-rts-addr s))
                   (equal (iread-dn 32 0 (stepn s 9)) 0)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 9)))
                          (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 9)))
                          (add 32 (read-an 32 6 s) 8))
                   (equal (read-mem x (mc-mem (stepn s 9)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma strncmp-s0-sn-rfile-base2 (rewrite)
     (implies (and (strncmp-s0p s i* i str1 n1 lst1 str2 n2 lst2 n n_)
                   (equal (get-nth i lst1) (get-nth i lst2))
                   (equal (get-nth i lst1) 0)
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 9)))
                     (if (d4-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (get-vlst oplen 0 rn '(2 3) (movem-saved s 4 8 2))))))

; base case 3: s0 --> sn, when lst[i] = lst2[i], lst[i] =~ 0, and n-1 = 0.
(prove-lemma strncmp-s0-sn-base3 (rewrite)
     (implies (and (strncmp-s0p s i* i str1 n1 lst1 str2 n2 lst2 n n_)
                   (equal (get-nth i lst1) (get-nth i lst2))
                   (not (equal (get-nth i lst1) 0))
                   (equal (sub1 n) 0))
              (and (equal (mc-status (stepn s 11)) 'running)
                   (equal (mc-pc (stepn s 11)) (linked-rts-addr s))
                   (equal (iread-dn 32 0 (stepn s 11)) 0)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 11)))
                          (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 11)))
                          (add 32 (read-an 32 6 s) 8))
                   (equal (read-mem x (mc-mem (stepn s 11)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma strncmp-s0-sn-rfile-base3 (rewrite)
     (implies (and (strncmp-s0p s i* i str1 n1 lst1 str2 n2 lst2 n n_)
```

```
                (equal (get-nth i lst1) (get-nth i lst2))
                (not (equal (get-nth i lst1) 0))
                (equal (sub1 n) 0)
                (leq oplen 32)
                (d2-7a2-5p rn))
           (equal (read-rn oplen rn (mc-rfile (stepn s 11)))
                  (if (d4-7a2-5p rn)
                      (read-rn oplen rn (mc-rfile s))
                    (get-vlst oplen 0 rn '(2 3) (movem-saved s 4 8 2))))))

; induction case: s0 --> s0, lst[i] = lst2[i], lst[i] =- 0 and n-1 =- 0.
(prove-lemma strncmp-s0-s0 (rewrite)
     (implies (and (strncmp-s0p s i* i str1 n1 lst1 str2 n2 lst2 n n_)
                (equal (get-nth i lst1) (get-nth i lst2))
                (not (equal (get-nth i lst1) 0))
                (not (equal (sub1 n) 0)))
           (and (strncmp-s0p (stepn s 7) (add 32 i* 1) (add1 i)
                             str1 n1 lst1 str2 n2 lst2 (sub1 n) n_)
                (equal (read-rn 32 14 (mc-rfile (stepn s 7)))
                       (read-rn 32 14 (mc-rfile s)))
                (equal (linked-a6 (stepn s 7)) (linked-a6 s))
                (equal (linked-rts-addr (stepn s 7)) (linked-rts-addr s))
                (equal (movem-saved (stepn s 7) 4 8 2)
                       (movem-saved s 4 8 2))
                (equal (read-mem x (mc-mem (stepn s 7)) k)
                       (read-mem x (mc-mem s) k)))))

(prove-lemma strncmp-s0-s0-rfile (rewrite)
     (implies (and (strncmp-s0p s i* i str1 n1 lst1 str2 n2 lst2 n n_)
                (equal (get-nth i lst1) (get-nth i lst2))
                (not (equal (get-nth i lst1) 0))
                (not (equal (sub1 n) 0))
                (d4-7a2-5p rn))
           (equal (read-rn oplen rn (mc-rfile (stepn s 7)))
                  (read-rn oplen rn (mc-rfile s)))))

; put together. s0 --> exit.
(prove-lemma strncmp-s0-sn (rewrite)
     (let ((sn (stepn s (strncmp1-t i n lst1 lst2))))
       (implies (strncmp-s0p s i* i str1 n1 lst1 str2 n2 lst2 n n_)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (iread-dn 32 0 sn) (strncmp1 i n lst1 lst2))
                     (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))
                     (equal (read-mem x (mc-mem sn) k)
                            (read-mem x (mc-mem s) k))))))
     ((induct (strncmp-induct s i* i n lst1 lst2))
      (disable strncmp-s0p iread-dn)))

(prove-lemma strncmp-s0-sn-rfile (rewrite)
     (let ((sn (stepn s (strncmp1-t i n lst1 lst2))))
       (implies (and (strncmp-s0p s i* i str1 n1 lst1 str2 n2 lst2 n n_)
```

```
                    (d2-7a2-5p rn)
                    (leq oplen 32))
               (equal (read-rn oplen rn (mc-rfile sn))
                      (if (d4-7a2-5p rn)
                          (read-rn oplen rn (mc-rfile s))
                        (get-vlst oplen 0 rn '(2 3) (movem-saved s 4 8 2))))))
     ((induct (strncmp-induct s i* i n lst1 lst2))
      (disable strncmp-s0p)))

; the correctness of strncmp.
(prove-lemma strncmp-correctness (rewrite)
     (let ((sn (stepn s (strncmp-t n lst1 lst2))))
       (implies (strncmp-statep s str1 n1 lst1 str2 n2 lst2 n)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 7 s) 4))
                     (implies (and (d2-7a2-5p rn)
                                   (leq oplen 32))
                              (equal (read-rn oplen rn (mc-rfile sn))
                                     (read-rn oplen rn (mc-rfile s))))
                     (implies (disjoint x k (sub 32 12 (read-sp s)) 28)
                              (equal (read-mem x (mc-mem sn) k)
                                     (read-mem x (mc-mem s) k)))
                     (equal (iread-dn 32 0 sn) (strncmp n lst1 lst2))))))
     ((use (strncmp-s-s0))
      (disable strncmp-statep strncmp-s0p linked-rts-addr linked-a6 iread-dn)))

(disable strncmp-t)

; some properties of strncmp.
; see file cstring.events.
```

C.24 The strncpy Function

```
;          Proof of the Correctness of the STRNCPY Function
#|
This is part of our effort to verify the Berkeley string library.  The
Berkeley string library is widely used as part of the Berkeley Unix OS.

This is the source code of strncpy function in the Berkeley string library.

/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 */
char *
strncpy(dst, src, n)
        char *dst;
        const char *src;
```

```
        register size_t n;
{
        if (n != 0) {
                register char *d = dst;
                register const char *s = src;

                do {
                        if ((*d++ = *s++) == 0) {
                                /* NUL pad the remaining n-1 bytes */
                                while (--n != 0)
                                        *d++ = 0;
                                break;
                        }
                } while (--n != 0);
        }
        return (dst);
}
```

The MC68020 assembly code of the C function strncpy on SUN-3 is given as follows. This binary is generated by "gcc -O".

```
0x2650 <strncpy>:       linkw fp,#0
0x2654 <strncpy+4>:     movel d2,sp@-
0x2656 <strncpy+6>:     movel fp@(8),d2
0x265a <strncpy+10>:    movel fp@(16),d1
0x265e <strncpy+14>:    beq 0x267a <strncpy+42>
0x2660 <strncpy+16>:    moveal d2,a0
0x2662 <strncpy+18>:    moveal fp@(12),a1
0x2666 <strncpy+22>:    moveb a1@+,d0
0x2668 <strncpy+24>:    moveb d0,a0@+
0x266a <strncpy+26>:    bne 0x2676 <strncpy+38>
0x266c <strncpy+28>:    bra 0x2670 <strncpy+32>
0x266e <strncpy+30>:    clrb a0@+
0x2670 <strncpy+32>:    subl #1,d1
0x2672 <strncpy+34>:    bne 0x266e <strncpy+30>
0x2674 <strncpy+36>:    bra 0x267a <strncpy+42>
0x2676 <strncpy+38>:    subl #1,d1
0x2678 <strncpy+40>:    bne 0x2666 <strncpy+22>
0x267a <strncpy+42>:    movel d2,d0
0x267c <strncpy+44>:    movel fp@(-4),d2
0x2680 <strncpy+48>:    unlk fp
0x2682 <strncpy+50>:    rts
```

The machine code of the above program is:

```
<strncpy>:     0x4e56  0x0000  0x2f02  0x242e  0x0008  0x222e  0x0010  0x671a
<strncpy+16>:  0x2042  0x226e  0x000c  0x1019  0x10c0  0x660a  0x6002  0x4218
<strncpy+32>:  0x5381  0x66fa  0x6004  0x5381  0x66ec  0x2002  0x242e  0xfffc
<strncpy+48>:  0x4e5e  0x4e75
```

```
'(78      86      0       0       47      2       36      46
  0       8       34      46      0       16      103     26
  32      66      34      110     0       12      16      25
  16      192     102     10      96      2       66      24
```

```
  83      129     102     250     96      4       83      129
  102     236     32      2       36      46      255     252
  78      94      78      117)
|#

; in the logic, the above program is defined by (strncpy-code).
(defn strncpy-code ()
  '(78      86      0       0       47      2       36      46
    0       8       34      46      0       16      103     26
    32      66      34      110     0       12      16      25
    16      192     102     10      96      2       66      24
    83      129     102     250     96      4       83      129
    102     236     32      2       36      46      255     252
    78      94      78      117))

; the computation time of the program.
(defn strncpy-t0 (i n)
  (if (equal (sub1 n) 0)
      7
    (splus 3 (strncpy-t0 (add1 i) (sub1 n)))))

(defn strncpy-t1 (i n)
  (splus 4 (strncpy-t0 (add1 i) n)))

(defn strncpy-t2 (i n lst2)
  (if (equal (get-nth i lst2) 0)
      (strncpy-t1 i n)
    (if (equal (sub1 n) 0)
        9
      (splus 5 (strncpy-t2 (add1 i) (sub1 n) lst2)))))

(defn strncpy-t (n lst2)
  (if (zerop n)
      9
    (splus 7 (strncpy-t2 0 n lst2))))

; two induction hints.
(defn strncpy-induct1 (s i* i n lst1)
  (if (equal (sub1 n) 0)
      t
    (strncpy-induct1 (stepn s 3) (add 32 i* 1) (add1 i) (sub1 n)
                     (put-nth 0 i lst1))))

(defn strncpy-induct2 (s i* i n lst1 lst2)
  (if (equal (get-nth i lst2) 0)
      t
    (if (equal (sub1 n) 0)
        t
      (strncpy-induct2 (stepn s 5) (add 32 i* 1) (add1 i) (sub1 n)
                       (put-nth (get-nth i lst2) i lst1) lst2))))

; the preconditions of the initial state.
(defn strncpy-statep (s str1 n lst1 str2 n2 lst2)
  (and (equal (mc-status s) 'running)
```

```
       (evenp (mc-pc s))
       (rom-addrp (mc-pc s) (mc-mem s) 52)
       (mcode-addrp (mc-pc s) (mc-mem s) (strncpy-code))
       (ram-addrp (sub 32 8 (read-sp s)) (mc-mem s) 24)
       (ram-addrp str1 (mc-mem s) n)
       (mem-lst 1 str1 (mc-mem s) n lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (sub 32 8 (read-sp s)) 24 str1 n)
       (disjoint (sub 32 8 (read-sp s)) 24 str2 n2)
       (disjoint str1 n str2 n2)
       (lessp (slen 0 n lst2) n2)
       (equal str1 (read-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal str2 (read-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
       (equal n (uread-mem (add 32 (read-sp s) 12) (mc-mem s) 4))
       (numberp n2)))

; an intermediate state s0.
(defn strncpy-s0p (s i* i str1 n_ lst1 str2 n2 lst2 n)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 22 (mc-pc s)) (mc-mem s) 52)
       (mcode-addrp (sub 32 22 (mc-pc s)) (mc-mem s) (strncpy-code))
       (ram-addrp (sub 32 4 (read-an 32 6 s)) (mc-mem s) 24)
       (ram-addrp str1 (mc-mem s) n_)
       (mem-lst 1 str1 (mc-mem s) n_ lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (sub 32 4 (read-an 32 6 s)) 24 str1 n_)
       (disjoint (sub 32 4 (read-an 32 6 s)) 24 str2 n2)
       (disjoint str1 n_ str2 n2)
       (lessp (slen i n_ lst2) n2)
       (equal* (read-an 32 0 s) (add 32 str1 i*))
       (equal* (read-an 32 1 s) (add 32 str2 i*))
       (equal str1 (read-dn 32 2 s))
       (equal n (nat-to-uint (read-dn 32 1 s)))
       (leq (plus i n) n_)
       (not (equal n 0))
       (numberp i*)
       (nat-rangep i* 32)
       (equal i (nat-to-uint i*))
       (numberp n2)
       (numberp n_)
       (uint-rangep n_ 32)))

; an intermediate state s1.
(defn strncpy-s1p (s i* i str1 n_ lst1 str2 n2 lst2 n)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 32 (mc-pc s)) (mc-mem s) 52)
       (mcode-addrp (sub 32 32 (mc-pc s)) (mc-mem s) (strncpy-code))
       (ram-addrp (sub 32 4 (read-an 32 6 s)) (mc-mem s) 24)
       (ram-addrp str1 (mc-mem s) n_)
       (mem-lst 1 str1 (mc-mem s) n_ lst1)
```

```
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (sub 32 4 (read-an 32 6 s)) 24 str1 n_)
       (disjoint (sub 32 4 (read-an 32 6 s)) 24 str2 n2)
       (disjoint str1 n_ str2 n2)
       (equal* (read-an 32 0 s) (add 32 str1 i*))
       (equal str1 (read-dn 32 2 s))
       (equal n (nat-to-uint (read-dn 32 1 s)))
       (equal i (nat-to-uint i*))
       (leq (plus i (sub1 n)) n_)
       (not (equal n 0))
       (numberp i*)
       (nat-rangep i* 32)
       (numberp n2)
       (numberp n_)
       (uint-rangep n_ 32)))

; from the initial state s to exit: s --> sn, when n = 0.
(prove-lemma strncpy-s-sn (rewrite)
     (implies (and (strncpy-statep s str1 n lst1 str2 n2 lst2)
                   (zerop n))
              (and (equal (mc-status (stepn s 9)) 'running)
                   (equal (mc-pc (stepn s 9)) (rts-addr s))
                   (equal (read-dn 32 0 (stepn s 9)) str1)
                   (mem-lst 1 str1 (mc-mem (stepn s 9)) n lst1)
                   (equal (read-rn 32 15 (mc-rfile (stepn s 9)))
                          (add 32 (read-an 32 7 s) 4))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 9)))
                          (read-an 32 6 s)))))

(prove-lemma strncpy-s-sn-rfile (rewrite)
     (implies (and (strncpy-statep s str1 n lst1 str2 n2 lst2)
                   (zerop n)
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 9)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strncpy-s-sn-mem (rewrite)
     (implies (and (strncpy-statep s str1 n lst1 str2 n2 lst2)
                   (zerop n)
                   (disjoint x k (sub 32 8 (read-sp s)) 24))
              (equal (read-mem x (mc-mem (stepn s 9)) k)
                     (read-mem x (mc-mem s) k))))

; from the initial state s to s0: s --> s0.
(prove-lemma strncpy-s-s0 ()
     (implies (and (strncpy-statep s str1 n lst1 str2 n2 lst2)
                   (not (zerop n)))
              (strncpy-s0p (stepn s 7) 0 0 str1 n lst1 str2 n2 lst2 n)))

(prove-lemma strncpy-s-s0-else (rewrite)
     (implies (and (strncpy-statep s str1 n lst1 str2 n2 lst2)
                   (not (zerop n)))
```

```
              (and (equal (linked-rts-addr (stepn s 7)) (rts-addr s))
                   (equal (linked-a6 (stepn s 7)) (read-an 32 6 s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 7)))
                          (sub 32 4 (read-sp s)))
                   (equal (rn-saved (stepn s 7))
                          (read-rn 32 2 (mc-rfile s))))))

(prove-lemma strncpy-s-s0-rfile (rewrite)
     (implies (and (strncpy-statep s str1 n lst1 str2 n2 lst2)
                   (not (zerop n))
                   (d3-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 7)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strncpy-s-s0-mem (rewrite)
     (implies (and (strncpy-statep s str1 n lst1 str2 n2 lst2)
                   (not (zerop n))
                   (disjoint x k (sub 32 8 (read-sp s)) 24))
              (equal (read-mem x (mc-mem (stepn s 7)) k)
                     (read-mem x (mc-mem s) k))))

; from s0 to exit: s0 --> sn.  By induction.
; base case 1:  s0 --> sn, when lst1[i] = 0.
; s0 --> s1.
(prove-lemma strncpy-s0-s1 (rewrite)
     (implies (and (strncpy-s0p s i* i str1 n_ lst1 str2 n2 lst2 n)
                   (equal (get-nth i lst2) 0))
              (and (strncpy-s1p (stepn s 4) (add 32 i* 1) (add1 i) str1 n_
                                (put-nth (get-nth i lst2) i lst1) str2 n2
                                lst2 n)
                   (equal (linked-rts-addr (stepn s 4))
                          (linked-rts-addr s))
                   (equal (linked-a6 (stepn s 4)) (linked-a6 s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 4)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (rn-saved (stepn s 4)) (rn-saved s)))))

(prove-lemma strncpy-s0-s1-rfile (rewrite)
     (implies (and (strncpy-s0p s i* i str1 n_ lst1 str2 n2 lst2 n)
                   (equal (get-nth i lst2) 0)
                   (d3-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 4)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strncpy-s0-s1-mem (rewrite)
     (implies (and (strncpy-s0p s i* i str1 n_ lst1 str2 n2 lst2 n)
                   (equal (get-nth i lst2) 0)
                   (disjoint x k str1 n_))
              (equal (read-mem x (mc-mem (stepn s 4)) k)
                     (read-mem x (mc-mem s) k))))

; s1 --> sn.  By induction.
; base case: s1 --> exit, when n-1 = 0.
(prove-lemma strncpy-s1-sn-base (rewrite)
```

```
     (implies (and (strncpy-s1p s i* i str1 n_ lst1 str2 n2 lst2 n)
                   (equal (sub1 n) 0))
              (and (equal (mc-status (stepn s 7)) 'running)
                   (equal (mc-pc (stepn s 7)) (linked-rts-addr s))
                   (equal (read-dn 32 0 (stepn s 7)) str1)
                   (mem-lst 1 str1 (mc-mem (stepn s 7)) n_ lst1)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 7)))
                          (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 7)))
                          (add 32 (read-an 32 6 s) 8))
                   (equal (read-mem x (mc-mem (stepn s 7)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma strncpy-s1-sn-rfile-base (rewrite)
     (implies (and (strncpy-s1p s i* i str1 n_ lst1 str2 n2 lst2 n)
                   (equal (sub1 n) 0)
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 7)))
                     (if (d3-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (head (rn-saved s) oplen)))))

; induction case: s1 --> s1, when n-1 =- 0.
(prove-lemma strncpy-s1-s1 (rewrite)
     (implies (and (strncpy-s1p s i* i str1 n_ lst1 str2 n2 lst2 n)
                   (not (equal (sub1 n) 0)))
              (and (strncpy-s1p (stepn s 3) (add 32 i* 1) (add1 i) str1
                                n_ (put-nth 0 i lst1) str2 n2 lst2 (sub1 n))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 3)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (linked-a6 (stepn s 3)) (linked-a6 s))
                   (equal (linked-rts-addr (stepn s 3)) (linked-rts-addr s))
                   (equal (rn-saved (stepn s 3)) (rn-saved s)))))

(prove-lemma strncpy-s1-s1-rfile (rewrite)
     (implies (and (strncpy-s1p s i* i str1 n_ lst1 str2 n2 lst2 n)
                   (not (equal (sub1 n) 0))
                   (d3-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 3)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strncpy-s1-s1-mem (rewrite)
     (implies (and (strncpy-s1p s i* i str1 n_ lst1 str2 n2 lst2 n)
                   (disjoint x k str1 n_)
                   (not (equal (sub1 n) 0)))
              (equal (read-mem x (mc-mem (stepn s 3)) k)
                     (read-mem x (mc-mem s) k))))

; put together. s1 --> exit.
(prove-lemma strncpy-s1-sn (rewrite)
     (let ((sn (stepn s (strncpy-t0 i n))))
       (implies (strncpy-s1p s i* i str1 n_ lst1 str2 n2 lst2 n)
                (and (equal (mc-status sn) 'running)
```

```
                    (equal (mc-pc sn) (linked-rts-addr s))
                    (equal (read-dn 32 0 sn) str1)
                    (mem-lst 1 str1 (mc-mem sn) n_ (zero-list1 i n lst1))
                    (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
                    (equal (read-rn 32 15 (mc-rfile sn))
                           (add 32 (read-an 32 6 s) 8)))))
     ((induct (strncpy-induct1 s i* i n lst1))
      (disable strncpy-s1p read-dn)))

(prove-lemma strncpy-s1-sn-rfile (rewrite)
     (implies (and (strncpy-s1p s i* i str1 n_ lst1 str2 n2 lst2 n)
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s (strncpy-t0 i n))))
                     (if (d3-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (head (rn-saved s) oplen))))
     ((induct (strncpy-induct1 s i* i n lst1))
      (disable strncpy-s1p)))

(prove-lemma strncpy-s1-sn-mem (rewrite)
     (implies (and (strncpy-s1p s i* i str1 n_ lst1 str2 n2 lst2 n)
                   (disjoint x k str1 n_))
              (equal (read-mem x (mc-mem (stepn s (strncpy-t0 i n))) k)
                     (read-mem x (mc-mem s) k)))
     ((induct (strncpy-induct1 s i* i n lst1))
      (disable strncpy-s1p)))

; put together (base case 1). s0 --> exit.
(prove-lemma strncpy-s0-sn-base1 (rewrite)
     (let ((sn (stepn s (strncpy-t1 i n))))
       (implies (and (strncpy-s0p s i* i str1 n_ lst1 str2 n2 lst2 n)
                     (equal (get-nth i lst2) 0))
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) str1)
                     (mem-lst 1 str1 (mc-mem sn) n_ (zero-list i n lst1))
                     (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8)))))
     ((use (strncpy-s0-s1))
      (disable strncpy-s0p strncpy-s1p read-dn)))

(prove-lemma strncpy-s0-sn-rfile-base1 (rewrite)
     (implies (and (strncpy-s0p s i* i str1 n_ lst1 str2 n2 lst2 n)
                   (equal (get-nth i lst2) 0)
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s (strncpy-t1 i n))))
                     (if (d3-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (head (rn-saved s) oplen))))
     ((use (strncpy-s0-s1))
      (disable strncpy-s0p strncpy-s1p)))
```

```
(prove-lemma strncpy-s0-sn-mem-base1 (rewrite)
     (implies (and (strncpy-s0p s i* i str1 n_ lst1 str2 n2 lst2 n)
                   (equal (get-nth i lst2) 0)
                   (disjoint x k str1 n_))
              (equal (read-mem x (mc-mem (stepn s (strncpy-t1 i n))) k)
                     (read-mem x (mc-mem s) k)))
     ((use (strncpy-s0-s1))
      (disable strncpy-s0p strncpy-s1p)))

; base case 2: s0 --> sn, when lst2[i] =- 0, n-1 = 0.
(prove-lemma strncpy-s0-sn-base2 (rewrite)
     (implies (and (strncpy-s0p s i* i str1 n_ lst1 str2 n2 lst2 n)
                   (not (equal (get-nth i lst2) 0))
                   (equal (sub1 n) 0))
              (and (equal (mc-status (stepn s 9)) 'running)
                   (equal (mc-pc (stepn s 9)) (linked-rts-addr s))
                   (equal (read-dn 32 0 (stepn s 9)) str1)
                   (mem-lst 1 str1 (mc-mem (stepn s 9)) n_
                            (put-nth (get-nth i lst2) i lst1))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 9)))
                          (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 9)))
                          (add 32 (read-an 32 6 s) 8)))))

(prove-lemma strncpy-s0-sn-rfile-base2 (rewrite)
     (implies (and (strncpy-s0p s i* i str1 n_ lst1 str2 n2 lst2 n)
                   (not (equal (get-nth i lst2) 0))
                   (equal (sub1 n) 0)
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 9)))
                     (if (d3-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (head (rn-saved s) oplen)))))

(prove-lemma strncpy-s0-sn-mem-base2 (rewrite)
     (implies (and (strncpy-s0p s i* i str1 n_ lst1 str2 n2 lst2 n)
                   (disjoint x k str1 n_)
                   (not (equal (get-nth i lst2) 0))
                   (equal (sub1 n) 0))
              (equal (read-mem x (mc-mem (stepn s 9)) k)
                     (read-mem x (mc-mem s) k))))

; induction case:  s0 --> s0, when lst[i] =- 0, n-1 =- 0.
(prove-lemma strncpy-s0-s0 (rewrite)
     (implies (and (strncpy-s0p s i* i str1 n_ lst1 str2 n2 lst2 n)
                   (not (equal (get-nth i lst2) 0))
                   (not (equal (sub1 n) 0)))
              (and (strncpy-s0p (stepn s 5) (add 32 i* 1) (add1 i) str1
                                n_ (put-nth (get-nth i lst2) i lst1)
                                str2 n2 lst2 (sub1 n))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 5)))
                          (read-rn 32 14 (mc-rfile s)))
```

```
                (equal (linked-a6 (stepn s 5)) (linked-a6 s))
                (equal (linked-rts-addr (stepn s 5)) (linked-rts-addr s))
                (equal (rn-saved (stepn s 5)) (rn-saved s)))))

(prove-lemma strncpy-s0-s0-rfile (rewrite)
     (implies (and (strncpy-s0p s i* i str1 n_ lst1 str2 n2 lst2 n)
                   (not (equal (get-nth i lst2) 0))
                   (not (equal (sub1 n) 0))
                   (d3-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 5)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strncpy-s0-s0-mem (rewrite)
     (implies (and (strncpy-s0p s i* i str1 n_ lst1 str2 n2 lst2 n)
                   (disjoint x k str1 n_)
                   (not (equal (get-nth i lst2) 0))
                   (not (equal (sub1 n) 0)))
              (equal (read-mem x (mc-mem (stepn s 5)) k)
                     (read-mem x (mc-mem s) k))))

; put together. s0 --> exit.
(prove-lemma strncpy-s0-sn (rewrite)
     (let ((sn (stepn s (strncpy-t2 i n lst2))))
       (implies (strncpy-s0p s i* i str1 n_ lst1 str2 n2 lst2 n)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) str1)
                     (mem-lst 1 str1 (mc-mem sn) n_ (strncpy1 i n lst1 lst2))
                     (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))))))
     ((induct (strncpy-induct2 s i* i n lst1 lst2))
      (disable strncpy-s0p strncpy-t1 read-dn zero-list)))

(prove-lemma strncpy-s0-sn-rfile (rewrite)
     (implies
      (and (strncpy-s0p s i* i str1 n_ lst1 str2 n2 lst2 n)
           (d2-7a2-5p rn)
           (leq oplen 32))
      (equal (read-rn oplen rn (mc-rfile (stepn s (strncpy-t2 i n lst2))))
             (if (d3-7a2-5p rn)
                 (read-rn oplen rn (mc-rfile s))
               (head (rn-saved s) oplen))))
     ((induct (strncpy-induct2 s i* i n lst1 lst2))
      (disable strncpy-s0p strncpy-t1)))

(prove-lemma strncpy-s0-sn-mem (rewrite)
     (implies
      (and (strncpy-s0p s i* i str1 n_ lst1 str2 n2 lst2 n)
           (disjoint x k str1 n_))
      (equal (read-mem x (mc-mem (stepn s (strncpy-t2 i n lst2))) k)
             (read-mem x (mc-mem s) k)))
     ((induct (strncpy-induct2 s i* i n lst1 lst2))
      (disable strncpy-s0p strncpy-t1)))
```

```
; the correctness of strncpy.
(prove-lemma strncpy-correctness (rewrite)
     (let ((sn (stepn s (strncpy-t n lst2))))
       (implies (strncpy-statep s str1 n lst1 str2 n2 lst2)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 7 s) 4))
                     (implies (and (d2-7a2-5p rn)
                                   (leq oplen 32))
                              (equal (read-rn oplen rn (mc-rfile sn))
                                     (read-rn oplen rn (mc-rfile s))))
                     (implies (and (disjoint x k str1 n)
                                   (disjoint x k (sub 32 8 (read-sp s)) 24))
                              (equal (read-mem x (mc-mem sn) k)
                                     (read-mem x (mc-mem s) k)))
                     (equal (read-dn 32 0 sn) str1)
                     (mem-lst 1 str1 (mc-mem sn) n (strncpy n lst1 lst2)))))
     ((use (strncpy-s-s0))
      (disable strncpy-statep strncpy-s0p read-dn linked-rts-addr linked-a6)))

(disable strncpy-t)

; some properties of strncpy.
; see file cstring.events.
```

C.25 The strpbrk Function

```
;          Proof of the Correctness of the STRPBRK Function
#|
This is part of our effort to verify the Berkeley string library.  The
Berkeley string library is widely used as part of the Berkeley Unix OS.

This is the source code of strpbrk function in the Berkeley string library.

/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 */
char *
strpbrk(s1, s2)
        register const char *s1, *s2;
{
        register const char *scanp;
        register int c, sc;

        while ((c = *s1++) != 0) {
                for (scanp = s2; (sc = *scanp++) != 0;)
                        if (sc == c)
```

```
                                return ((char *)(s1 - 1));
        }
        return (NULL);
}
```

Remark. Should the local variables c and sc have type register char?

The MC68020 assembly code of the C function strpbrk on SUN-3 is given as follows. This binary is generated by "gcc -O".

```
0x2688 <strpbrk>:       linkw fp,#0
0x268c <strpbrk+4>:     moveml d2-d3,sp@-
0x2690 <strpbrk+8>:     moveal fp@(8),a1
0x2694 <strpbrk+12>:    movel fp@(12),d3
0x2698 <strpbrk+16>:    bra 0x26b0 <strpbrk+40>
0x269a <strpbrk+18>:    moveal d3,a0
0x269c <strpbrk+20>:    movel a1,d1
0x269e <strpbrk+22>:    subl #1,d1
0x26a0 <strpbrk+24>:    bra 0x26aa <strpbrk+34>
0x26a2 <strpbrk+26>:    cmpl d0,d2
0x26a4 <strpbrk+28>:    bne 0x26aa <strpbrk+34>
0x26a6 <strpbrk+30>:    movel d1,d0
0x26a8 <strpbrk+32>:    bra 0x26b8 <strpbrk+48>
0x26aa <strpbrk+34>:    moveb a0@+,d0
0x26ac <strpbrk+36>:    extbl d0
0x26ae <strpbrk+38>:    bne 0x26a2 <strpbrk+26>
0x26b0 <strpbrk+40>:    moveb a1@+,d2
0x26b2 <strpbrk+42>:    extbl d2
0x26b4 <strpbrk+44>:    bne 0x269a <strpbrk+18>
0x26b6 <strpbrk+46>:    clrl d0
0x26b8 <strpbrk+48>:    moveml fp@(-8),d2-d3
0x26be <strpbrk+54>:    unlk fp
0x26c0 <strpbrk+56>:    rts
```

The machine code of the above program is:

```
<strpbrk>:     0x4e56  0x0000  0x48e7  0x3000  0x226e  0x0008  0x262e  0x000c
<strpbrk+16>:  0x6016  0x2043  0x2209  0x5381  0x6008  0xb480  0x6604  0x2001
<strpbrk+32>:  0x600e  0x1018  0x49c0  0x66f2  0x1419  0x49c2  0x66e4  0x4280
<strpbrk+48>:  0x4cee  0x000c  0xfff8  0x4e5e  0x4e75
```

```
'(78      86      0       0       72      231     48      0
  34      110     0       8       38      46      0       12
  96      22      32      67      34      9       83      129
  96      8       180     128     102     4       32      1
  96      14      16      24      73      192     102     242
  20      25      73      194     102     228     66      128
  76      238     0       12      255     248     78      94
  78      117)
|#

; in the logic, the above program is defined by (strpbrk-code).
(defn strpbrk-code ()
  '(78      86      0       0       72      231     48      0
```

```
    34      110     0       8       38      46      0       12
    96      22      32      67      34      9       83      129
    96      8       180     128     102     4       32      1
    96      14      16      24      73      192     102     242
    20      25      73      194     102     228     66      128
    76      238     0       12      255     248     78      94
    78      117))

; the computation time of the program.
(defn strpbrk-t0 (i2 n2 lst2 ch)
  (if (lessp i2 n2)
      (if (equal (get-nth i2 lst2) 0)
          3
        (if (equal (get-nth i2 lst2) ch)
            10
          (splus 5 (strpbrk-t0 (add1 i2) n2 lst2 ch))))
    0)
  ((lessp (difference n2 i2))))

(defn strpbrk-t1 (n2 lst2 ch)
  (splus 7 (strpbrk-t0 0 n2 lst2 ch)))

(defn strpbrk-t2 (i1 n1 lst1 n2 lst2)
  (if (lessp i1 n1)
      (if (equal (get-nth i1 lst1) 0)
          7
        (if (strchr1 0 n2 lst2 (get-nth i1 lst1))
            (strpbrk-t1 n2 lst2 (get-nth i1 lst1))
          (splus (strpbrk-t1 n2 lst2 (get-nth i1 lst1))
                 (strpbrk-t2 (add1 i1) n1 lst1 n2 lst2))))
    0)
  ((lessp (difference n1 i1))))

(defn strpbrk-t (n1 lst1 n2 lst2)
  (splus 5 (strpbrk-t2 0 n1 lst1 n2 lst2)))

; two induction hints.
(defn strpbrk-induct0 (s i2* i2 n2 lst2 ch)
  (if (lessp i2 n2)
      (if (equal (get-nth i2 lst2) 0)
          t
        (if (equal (get-nth i2 lst2) ch)
            t
          (strpbrk-induct0 (stepn s 5) (add 32 i2* 1) (add1 i2) n2 lst2 ch)))
    t)
  ((lessp (difference n2 i2))))

(defn strpbrk-induct1 (s i1* i1 n1 lst1 n2 lst2)
  (if (lessp i1 n1)
      (if (or (equal (get-nth i1 lst1) 0)
              (strchr1 0 n2 lst2 (get-nth i1 lst1)))
          t
        (strpbrk-induct1 (stepn s (strpbrk-t1 n2 lst2 (get-nth i1 lst1)))
                         (add 32 i1* 1) (add1 i1) n1 lst1 n2 lst2))
```

```
    t)
  ((lessp (difference n1 i1))))

; the preconditions of the initial state.
(defn strpbrk-statep (s str1 n1 lst1 str2 n2 lst2)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (mc-pc s) (mc-mem s) 58)
       (mcode-addrp (mc-pc s) (mc-mem s) (strpbrk-code))
       (ram-addrp (sub 32 12 (read-sp s)) (mc-mem s) 24)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (sub 32 12 (read-sp s)) 24 str1 n1)
       (disjoint (sub 32 12 (read-sp s)) 24 str2 n2)
       (equal str1 (read-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal str2 (read-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
       (lessp (slen 0 n1 lst1) n1)
       (lessp (slen 0 n2 lst2) n2)
       (numberp n1)
       (numberp n2)
       (uint-rangep n1 32)
       (uint-rangep n2 32)
       (not (equal (nat-to-uint str1) 0))
       (uint-rangep (plus (nat-to-uint str1) n1) 32)))

(defn strpbrk-s0p (s i1* i1 str1 n1 lst1 str2 n2 lst2)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 40 (mc-pc s)) (mc-mem s) 58)
       (mcode-addrp (sub 32 40 (mc-pc s)) (mc-mem s) (strpbrk-code))
       (ram-addrp (sub 32 8 (read-an 32 6 s)) (mc-mem s) 24)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (sub 32 8 (read-an 32 6 s)) 24 str1 n1)
       (disjoint (sub 32 8 (read-an 32 6 s)) 24 str2 n2)
       (equal* (read-an 32 1 s) (add 32 str1 i1*))
       (equal str2 (read-dn 32 3 s))
       (numberp i1*)
       (nat-rangep i1* 32)
       (equal i1 (nat-to-uint i1*))
       (lessp (slen i1 n1 lst1) n1)
       (lessp (slen 0 n2 lst2) n2)
       (numberp n1)
       (numberp n2)
       (uint-rangep n1 32)
       (uint-rangep n2 32)))

(defn strpbrk-s1p (s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
```

```
       (rom-addrp (sub 32 34 (mc-pc s)) (mc-mem s) 58)
       (mcode-addrp (sub 32 34 (mc-pc s)) (mc-mem s) (strpbrk-code))
       (ram-addrp (sub 32 8 (read-an 32 6 s)) (mc-mem s) 24)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (sub 32 8 (read-an 32 6 s)) 24 str1 n1)
       (disjoint (sub 32 8 (read-an 32 6 s)) 24 str2 n2)
       (equal* (read-an 32 1 s) (add 32 str1 i1*))
       (equal* (read-an 32 0 s) (add 32 str2 i2*))
       (equal* (read-dn 32 1 s) (sub 32 1 (read-an 32 1 s)))
       (equal* (read-dn 32 2 s) (ext 8 (read-dn 8 2 s) 32))
       (equal str2 (read-dn 32 3 s))
       (equal ch (uread-dn 8 2 s))
       (lessp (slen i1 n1 lst1) n1)
       (lessp (slen 0 n2 lst2) n2)
       (lessp (slen i2 n2 lst2) n2)
       (numberp i1*)
       (nat-rangep i1* 32)
       (equal i1 (nat-to-uint i1*))
       (numberp i2*)
       (nat-rangep i2* 32)
       (equal i2 (nat-to-uint i2*))
       (numberp n1)
       (uint-rangep n1 32)
       (numberp n2)
       (uint-rangep n2 32)))

; from the initial state s to s0: s --> s0.
(prove-lemma strpbrk-s-s0 ()
     (implies (strpbrk-statep s str1 n1 lst1 str2 n2 lst2)
              (strpbrk-s0p (stepn s 5) 0 0 str1 n1 lst1 str2 n2 lst2)))

(prove-lemma strpbrk-s-s0-else (rewrite)
     (implies (strpbrk-statep s str1 n1 lst1 str2 n2 lst2)
              (and (equal (linked-rts-addr (stepn s 5)) (rts-addr s))
                   (equal (linked-a6 (stepn s 5)) (read-an 32 6 s))
                   (equal (read-an 32 6 (stepn s 5)) (sub 32 4 (read-sp s)))
                   (equal (movem-saved (stepn s 5) 4 8 2)
                          (readm-rn 32 '(2 3) (mc-rfile s))))))

(prove-lemma strpbrk-s-s0-rfile (rewrite)
     (implies (and (strpbrk-statep s str1 n1 lst1 str2 n2 lst2)
                   (d4-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 5)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strpbrk-s-s0-mem (rewrite)
     (implies (and (strpbrk-statep s str1 n1 lst1 str2 n2 lst2)
                   (disjoint x k (sub 32 12 (read-sp s)) 24))
              (equal (read-mem x (mc-mem (stepn s 5)) k)
                     (read-mem x (mc-mem s) k))))
```

```
; from s0 to exit.
; base case.  s0 --> sn, when lst1[i1] = 0.
(prove-lemma strpbrk-s0-sn-base1 (rewrite)
     (implies (and (strpbrk-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                   (equal (get-nth i1 lst1) 0))
              (and (equal (mc-status (stepn s 7)) 'running)
                   (equal (mc-pc (stepn s 7)) (linked-rts-addr s))
                   (equal (read-dn 32 0 (stepn s 7)) 0)
                   (equal (read-an 32 6 (stepn s 7)) (linked-a6 s))
                   (equal (read-an 32 7 (stepn s 7))
                          (add 32 (read-an 32 6 s) 8))
                   (equal (read-mem x (mc-mem (stepn s 7)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma strpbrk-s0-sn-rfile-base1 (rewrite)
     (implies (and (strpbrk-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                   (equal (get-nth i1 lst1) 0)
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 7)))
                     (if (d4-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (get-vlst oplen 0 rn '(2 3) (movem-saved s 4 8 2))))))

; induction case. s0 --> s0, when lst1[i1] =- 0.
; from s0 to s1: s0 --> s1, when lst1[i1] =- 0.
(prove-lemma strpbrk-s0-s1 ()
     (implies (and (strpbrk-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                   (not (equal (get-nth i1 lst1) 0)))
              (strpbrk-s1p (stepn s 7) (add 32 i1* 1) (add1 i1) str1 n1
                           lst1 0 0 str2 n2 lst2 (get-nth i1 lst1))))

(prove-lemma strpbrk-s0-s1-else (rewrite)
     (implies (and (strpbrk-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                   (not (equal (get-nth i1 lst1) 0)))
              (and (equal (read-an 32 6 (stepn s 7)) (read-an 32 6 s))
                   (equal (linked-a6 (stepn s 7)) (linked-a6 s))
                   (equal (linked-rts-addr (stepn s 7)) (linked-rts-addr s))
                   (equal (movem-saved (stepn s 7) 4 8 2)
                          (movem-saved s 4 8 2))
                   (equal (read-mem x (mc-mem (stepn s 7)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma strpbrk-s0-s1-rfile (rewrite)
     (implies (and (strpbrk-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                   (not (equal (get-nth i1 lst1) 0))
                   (d4-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 7)))
                     (read-rn oplen rn (mc-rfile s)))))

; from s1 to s0:
; base case. s1 --> s0, when lst2[i2] = 0.
(prove-lemma strpbrk-s1-s0-base (rewrite)
     (implies (and (strpbrk-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
```

```
                (equal (get-nth i2 lst2) 0))
           (and (strpbrk-s0p (stepn s 3) i1* i1 str1 n1 lst1 str2 n2 lst2)
                (equal (read-rn 32 14 (mc-rfile (stepn s 3)))
                       (read-rn 32 14 (mc-rfile s)))
                (equal (linked-a6 (stepn s 3)) (linked-a6 s))
                (equal (linked-rts-addr (stepn s 3))
                       (linked-rts-addr s))
                (equal (movem-saved (stepn s 3) 4 8 2)
                       (movem-saved s 4 8 2))
                (equal (read-mem x (mc-mem (stepn s 3)) k)
                       (read-mem x (mc-mem s) k)))))

(prove-lemma strpbrk-s1-s0-rfile-base (rewrite)
     (implies (and (strpbrk-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                (equal (get-nth i2 lst2) 0)
                (d4-7a2-5p rn))
           (equal (read-rn oplen rn (mc-rfile (stepn s 3)))
                  (read-rn oplen rn (mc-rfile s)))))

; induction case. s1 --> s1, when lst2[i2] =- 0 and lst2[i2] =- ch.
(prove-lemma strpbrk-s1-s1 (rewrite)
     (implies (and (strpbrk-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                (not (equal (get-nth i2 lst2) 0))
                (not (equal (get-nth i2 lst2) ch)))
           (and (strpbrk-s1p (stepn s 5) i1* i1 str1 n1 lst1 (add 32 i2* 1)
                             (add1 i2) str2 n2 lst2 ch)
                (equal (read-rn 32 14 (mc-rfile (stepn s 5)))
                       (read-rn 32 14 (mc-rfile s)))
                (equal (linked-a6 (stepn s 5)) (linked-a6 s))
                (equal (linked-rts-addr (stepn s 5))
                       (linked-rts-addr s))
                (equal (movem-saved (stepn s 5) 4 8 2)
                       (movem-saved s 4 8 2))
                (equal (read-mem x (mc-mem (stepn s 5)) k)
                       (read-mem x (mc-mem s) k)))))

(prove-lemma strpbrk-s1-s1-rfile (rewrite)
     (implies (and (strpbrk-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                (not (equal (get-nth i2 lst2) 0))
                (not (equal (get-nth i2 lst2) ch))
                (d4-7a2-5p rn))
           (equal (read-rn oplen rn (mc-rfile (stepn s 5)))
                  (read-rn oplen rn (mc-rfile s)))))

; put together.  s1 --> s0.
(prove-lemma strpbrk-s1p-info (rewrite)
     (implies (strpbrk-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
              (equal (lessp i2 n2) t)))

(prove-lemma strpbrk-s1-s0 (rewrite)
     (let ((s0 (stepn s (strpbrk-t0 i2 n2 lst2 ch))))
       (implies (and (strpbrk-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                     (not (strchr1 i2 n2 lst2 ch)))
                (and (strpbrk-s0p s0 i1* i1 str1 n1 lst1 str2 n2 lst2)
```

```
                    (equal (read-an 32 6 s0) (read-an 32 6 s))
                    (equal (linked-a6 s0) (linked-a6 s))
                    (equal (linked-rts-addr s0) (linked-rts-addr s))
                    (equal (movem-saved s0 4 8 2) (movem-saved s 4 8 2))
                    (equal (read-mem x (mc-mem s0) k)
                           (read-mem x (mc-mem s) k)))))
     ((induct (strpbrk-induct0 s i2* i2 n2 lst2 ch))
      (disable strpbrk-s1p strpbrk-s0p)))

(prove-lemma strpbrk-s1-s0-rfile (rewrite)
     (implies
      (and (strpbrk-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
           (not (strchr1 i2 n2 lst2 ch))
           (d4-7a2-5p rn))
      (equal (read-rn oplen rn (mc-rfile (stepn s (strpbrk-t0 i2 n2 lst2 ch))))
             (read-rn oplen rn (mc-rfile s))))
     ((induct (strpbrk-induct0 s i2* i2 n2 lst2 ch))
      (disable strpbrk-s1p)))

; from s1 to exit:
; base case. s1 --> sn, when lst2[i2] =~ 0 and lst2[i2] = ch.
(prove-lemma strpbrk-s1-sn-base (rewrite)
     (implies (and (strpbrk-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                   (not (equal (get-nth i2 lst2) 0))
                   (equal (get-nth i2 lst2) ch))
              (and (equal (mc-status (stepn s 10)) 'running)
                   (equal (mc-pc (stepn s 10)) (linked-rts-addr s))
                   (equal (read-dn 32 0 (stepn s 10))
                          (add 32 str1 (sub 32 1 i1*)))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 10)))
                          (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 10)))
                          (add 32 (read-an 32 6 s) 8))
                   (equal (read-mem x (mc-mem (stepn s 10)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma strpbrk-s1-sn-rfile-base (rewrite)
     (implies (and (strpbrk-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                   (not (equal (get-nth i2 lst2) 0))
                   (equal (get-nth i2 lst2) ch)
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 10)))
                     (if (d4-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (get-vlst oplen 0 rn '(2 3) (movem-saved s 4 8 2))))))

; put together.  s1 --> sn.
(prove-lemma strpbrk-s1-sn (rewrite)
     (let ((sn (stepn s (strpbrk-t0 i2 n2 lst2 ch))))
       (implies (and (strpbrk-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                     (strchr1 i2 n2 lst2 ch))
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
```

```
                    (equal (read-dn 32 0 sn) (add 32 str1 (sub 32 1 i1*)))
                    (equal (read-an 32 6 sn) (linked-a6 s))
                    (equal (read-an 32 7 sn) (add 32 (read-an 32 6 s) 8))
                    (equal (read-mem x (mc-mem sn) k)
                           (read-mem x (mc-mem s) k)))))
     ((induct (strpbrk-induct0 s i2* i2 n2 lst2 ch))
      (disable strpbrk-s1p read-dn)))

(prove-lemma strpbrk-s1-sn-rfile (rewrite)
     (implies
      (and (strpbrk-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
           (strchr1 i2 n2 lst2 ch)
           (d2-7a2-5p rn)
           (leq oplen 32))
      (equal (read-rn oplen rn (mc-rfile (stepn s (strpbrk-t0 i2 n2 lst2 ch))))
             (if (d4-7a2-5p rn)
                 (read-rn oplen rn (mc-rfile s))
               (get-vlst oplen 0 rn '(2 3) (movem-saved s 4 8 2)))))
     ((induct (strpbrk-induct0 s i2* i2 n2 lst2 ch))
      (disable strpbrk-s1p)))

; base case 2. from s0 --> sn.
(prove-lemma strpbrk-s0-sn-base2 (rewrite)
     (let ((ch (get-nth i1 lst1))
           (sn (stepn s (strpbrk-t1 n2 lst2 (get-nth i1 lst1)))))
       (implies (and (strpbrk-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                     (not (equal (get-nth i1 lst1) 0))
                     (strchr1 0 n2 lst2 ch))
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) (add 32 str1 i1*))
                     (equal (read-an 32 6 sn) (linked-a6 s))
                     (equal (read-an 32 7 sn) (add 32 (read-an 32 6 s) 8))
                     (equal (read-mem x (mc-mem sn) k)
                            (read-mem x (mc-mem s) k)))))
     ((use (strpbrk-s0-s1))
      (disable strpbrk-s0p strpbrk-s1p strchr1 strpbrk-t0 read-an read-dn)))

(prove-lemma strpbrk-s0-sn-rfile-base2 (rewrite)
     (let ((ch (get-nth i1 lst1)))
       (implies
        (and (strpbrk-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
             (not (equal (get-nth i1 lst1) 0))
             (strchr1 0 n2 lst2 ch)
             (leq oplen 32)
             (d2-7a2-5p rn))
        (equal (read-rn oplen rn (mc-rfile (stepn s (strpbrk-t1 n2 lst2 ch))))
               (if (d4-7a2-5p rn)
                   (read-rn oplen rn (mc-rfile s))
                 (get-vlst oplen 0 rn '(2 3) (movem-saved s 4 8 2)))))))
     ((use (strpbrk-s0-s1))
      (disable strpbrk-s0p strpbrk-s1p strchr1 strpbrk-t0)))

; induction case.  from s0 --> s0.
```

```
(prove-lemma strpbrk-s0-s0 (rewrite)
     (let ((ch (get-nth i1 lst1))
           (s0 (stepn s (strpbrk-t1 n2 lst2 (get-nth i1 lst1)))))
       (implies (and (strpbrk-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                     (not (equal (get-nth i1 lst1) 0))
                     (not (strchr1 0 n2 lst2 ch)))
                (and (strpbrk-s0p s0 (add 32 i1* 1) (add1 i1) str1 n1 lst1
                                  str2 n2 lst2)
                     (equal (read-an 32 6 s0) (read-an 32 6 s))
                     (equal (linked-a6 s0) (linked-a6 s))
                     (equal (linked-rts-addr s0) (linked-rts-addr s))
                     (equal (movem-saved s0 4 8 2) (movem-saved s 4 8 2))
                     (equal (read-mem x (mc-mem s0) k)
                            (read-mem x (mc-mem s) k)))))
       ((use (strpbrk-s0-s1))
        (disable strpbrk-s0p strpbrk-s1p strpbrk-t0 strchr1 read-an)))

(prove-lemma strpbrk-s0-s0-rfile (rewrite)
     (let ((ch (get-nth i1 lst1)))
       (implies
        (and (strpbrk-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
             (not (equal (get-nth i1 lst1) 0))
             (not (strchr1 0 n2 lst2 ch))
             (d4-7a2-5p rn))
        (equal (read-rn oplen rn (mc-rfile (stepn s (strpbrk-t1 n2 lst2 ch))))
               (read-rn oplen rn (mc-rfile s)))))
     ((use (strpbrk-s0-s1))
      (disable strpbrk-s0p strpbrk-s1p strpbrk-t0 strchr1)))

; put together.  s0 --> sn.
(prove-lemma strpbrk-s0p-info (rewrite)
     (implies (strpbrk-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
              (equal (lessp i1 n1) t)))

(prove-lemma strpbrk-s0-sn (rewrite)
     (let ((sn (stepn s (strpbrk-t2 i1 n1 lst1 n2 lst2))))
       (implies (strpbrk-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn)
                            (if (strpbrk i1 n1 lst1 n2 lst2)
                                (add 32 str1 (strpbrk* i1* i1 n1 lst1 n2 lst2))
                              0))
                     (equal (read-an 32 6 sn) (linked-a6 s))
                     (equal (read-an 32 7 sn) (add 32 (read-an 32 6 s) 8))
                     (equal (read-mem x (mc-mem sn) k)
                            (read-mem x (mc-mem s) k)))))
     ((induct (strpbrk-induct1 s i1* i1 n1 lst1 n2 lst2))
      (disable strpbrk-s0p strchr1 strpbrk-t1 read-an read-dn)))

(prove-lemma strpbrk-s0-sn-rfile (rewrite)
     (let ((sn (stepn s (strpbrk-t2 i1 n1 lst1 n2 lst2))))
       (implies (and (strpbrk-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                     (d2-7a2-5p rn)
```

```
                    (leq oplen 32))
               (equal (read-rn oplen rn (mc-rfile sn))
                      (if (d4-7a2-5p rn)
                          (read-rn oplen rn (mc-rfile s))
                        (get-vlst oplen 0 rn '(2 3) (movem-saved s 4 8 2))))))
     ((induct (strpbrk-induct1 s i1* i1 n1 lst1 n2 lst2))
      (disable strpbrk-s0p strchr1 strpbrk-t1)))

; the correctness of strpbrk.
(prove-lemma strpbrk-correctness (rewrite)
     (let ((sn (stepn s (strpbrk-t n1 lst1 n2 lst2))))
       (implies (strpbrk-statep s str1 n1 lst1 str2 n2 lst2)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-an 32 6 sn) (read-an 32 6 s))
                     (equal (read-an 32 7 sn)
                            (add 32 (read-an 32 7 s) 4))
                     (implies (and (d2-7a2-5p rn)
                                   (leq oplen 32))
                              (equal (read-rn oplen rn (mc-rfile sn))
                                     (read-rn oplen rn (mc-rfile s))))
                     (implies (disjoint x k (sub 32 12 (read-sp s)) 24)
                              (equal (read-mem x (mc-mem sn) k)
                                     (read-mem x (mc-mem s) k)))
                     (equal (read-dn 32 0 sn)
                            (if (strpbrk 0 n1 lst1 n2 lst2)
                                (add 32 str1 (strpbrk* 0 0 n1 lst1 n2 lst2))
                              0)))))
     ((use (strpbrk-s-s0))
      (disable strpbrk-statep strpbrk-s0p linked-rts-addr linked-a6
               strpbrk-t2 read-an read-dn)))

(disable strpbrk-t)

; strpbrk* --> strpbrk.
(prove-lemma strpbrk*-strpbrk (rewrite)
     (implies (and (strpbrk i1 n1 lst1 n2 lst2)
                   (equal i1 (nat-to-uint i1*))
                   (nat-rangep i1* 32)
                   (uint-rangep n1 32))
              (equal (nat-to-uint (strpbrk* i1* i1 n1 lst1 n2 lst2))
                     (strpbrk i1 n1 lst1 n2 lst2)))
     ((induct (strpbrk* i1* i1 n1 lst1 n2 lst2))))

(prove-lemma strpbrk-non-zerop-la ()
     (let ((sn (stepn s (strpbrk-t n1 lst1 n2 lst2))))
       (implies (and (strpbrk-statep s str1 n1 lst1 str2 n2 lst2)
                     (nat-rangep str1 32)
                     (not (equal (nat-to-uint str1) 0))
                     (uint-rangep (plus (nat-to-uint str1) n1) 32)
                     (strpbrk 0 n1 lst1 n2 lst2))
                (not (equal (nat-to-uint (read-dn 32 0 sn)) 0)))))
     ((enable nat-rangep-la)
      (disable strpbrk-statep read-dn)))
```

```
(prove-lemma strpbrk-non-zerop (rewrite)
     (let ((sn (stepn s (strpbrk-t n1 lst1 n2 lst2))))
       (implies (and (strpbrk-statep s str1 n1 lst1 str2 n2 lst2)
                     (strpbrk 0 n1 lst1 n2 lst2))
              (not (equal (nat-to-uint (read-dn 32 0 sn)) 0))))
     ((use (strpbrk-non-zerop-1a))))

(disable strpbrk*)

; some properties of strrchr.
; see file cstring.events.
```

C.26 The strrchr Function

```
;          Proof of the Correctness of the STRRCHR Function
#|
This is part of our effort to verify the Berkeley string library.  The
Berkeley string library is widely used as part of the Berkeley Unix OS.

This is the source code of strrchr function in the Berkeley string library.

/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 */
char *
strrchr(p, ch)
        register const char *p;
        register char ch;
{
        register const char *save;

        for (save = NULL;; ++p) {
                if (*p == ch)
                        save = p;
                if (!*p)
                        return((char *)save);
        }
        /* NOTREACHED */
}

The MC68020 assembly code of the C function strrchr on SUN-3 is given as
follows.  This binary is generated by "gcc -O".

0x26c8 <strrchr>:       linkw fp,#0
0x26cc <strrchr+4>:     moveal fp@(8),a0
0x26d0 <strrchr+8>:     moveb fp@(15),d1
0x26d4 <strrchr+12>:    clrl d0
0x26d6 <strrchr+14>:    cmpb a0@,d1
0x26d8 <strrchr+16>:    bne 0x26dc <strrchr+20>
0x26da <strrchr+18>:    movel a0,d0
```

```
0x26dc <strrchr+20>:    tstb a0@
0x26de <strrchr+22>:    beq 0x26e4 <strrchr+28>
0x26e0 <strrchr+24>:    addqw #1,a0
0x26e2 <strrchr+26>:    bra 0x26d6 <strrchr+14>
0x26e4 <strrchr+28>:    unlk fp
0x26e6 <strrchr+30>:    rts

The machine code of the above program is:

<strrchr>:     0x4e56  0x0000  0x206e  0x0008  0x122e  0x000f  0x4280  0xb210
<strrchr+16>:  0x6602  0x2008  0x4a10  0x6704  0x5248  0x60f2  0x4e5e  0x4e75

'(78      86      0       0       32      110     0       8
  18      46      0       15      66      128     178     16
  102     2       32      8       74      16      103     4
  82      72      96      242     78      94      78      117)
|#

; in the logic, the above program is defined by (strrchr-code).
(defn strrchr-code ()
  '(78      86      0       0       32      110     0       8
    18      46      0       15      66      128     178     16
    102     2       32      8       74      16      103     4
    82      72      96      242     78      94      78      117))

; the computation time of the program.
(defn strrchr-t1 (i n lst ch)
  (if (lessp i n)
      (if (equal (get-nth i lst) ch)
          (if (equal (get-nth i lst) 0)
              7
            (splus 7 (strrchr-t1 (add1 i) n lst ch)))
        (if (equal (get-nth i lst) 0)
            6
          (splus 6 (strrchr-t1 (add1 i) n lst ch))))
    0)
  ((lessp (difference n i))))

(defn strrchr-t (n lst ch)
  (splus 4 (strrchr-t1 0 n lst ch)))

; an induction hint.
(defn strrchr-induct (s i* i n lst ch j* j)
  (if (lessp i n)
      (if (equal (get-nth i lst) ch)
          (if (equal (get-nth i lst) 0)
              t
            (strrchr-induct (stepn s 7) (add 32 i* 1) (add1 i) n lst ch
                            i* i))
        (if (equal (get-nth i lst) 0)
            t
          (strrchr-induct (stepn s 6) (add 32 i* 1) (add1 i) n lst ch j* j)))
    t)
  ((lessp (difference n i))))
```

```
; the preconditions of the initial state.
(defn strrchr-statep (s str n lst ch)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (mc-pc s) (mc-mem s) 32)
       (mcode-addrp (mc-pc s) (mc-mem s) (strrchr-code))
       (ram-addrp (sub 32 4 (read-sp s)) (mc-mem s) 16)
       (ram-addrp str (mc-mem s) n)
       (mem-lst 1 str (mc-mem s) n lst)
       (disjoint (sub 32 4 (read-sp s)) 16 str n)
       (equal str (read-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal ch (uread-mem (add 32 (read-sp s) 11) (mc-mem s) 1))
       (stringp 0 n lst)
       (numberp n)
       (uint-rangep n 32)
       (not (equal (nat-to-uint str) 0))
       (uint-rangep (plus (nat-to-uint str) n) 32)))

; an intermediate state.
(defn index-j (str j* j)
  (if j (add 32 str j*) 0))

(defn strrchr-s0p (s i* i str n lst ch j* j)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 14 (mc-pc s)) (mc-mem s) 32)
       (mcode-addrp (sub 32 14 (mc-pc s)) (mc-mem s) (strrchr-code))
       (ram-addrp (read-an 32 6 s) (mc-mem s) 16)
       (ram-addrp str (mc-mem s) n)
       (mem-lst 1 str (mc-mem s) n lst)
       (disjoint (read-an 32 6 s) 16 str n)
       (equal* (read-an 32 0 s) (add 32 str i*))
       (equal ch (nat-to-uint (read-dn 8 1 s)))
       (equal* (read-dn 32 0 s) (index-j str j* j))
       (stringp i n lst)
       (lessp i n)
       (numberp i*)
       (nat-rangep i* 32)
       (equal i (nat-to-uint i*))
       (numberp n)
       (uint-rangep n 32)))

; from the initial state s to s0: s --> s0;
(prove-lemma strrchr-s-s0 ()
     (implies (strrchr-statep s str n lst ch)
              (strrchr-s0p (stepn s 4) 0 0 str n lst ch f f)))

(prove-lemma strrchr-s-s0-else (rewrite)
     (implies (strrchr-statep s str n lst ch)
              (and (equal (linked-rts-addr (stepn s 4)) (rts-addr s))
                   (equal (linked-a6 (stepn s 4)) (read-an 32 6 s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 4)))
                          (sub 32 4 (read-sp s))))))
```

```
(prove-lemma strrchr-s-s0-rfile (rewrite)
     (implies (and (strrchr-statep s str n lst ch)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 4)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strrchr-s-s0-mem (rewrite)
     (implies (and (strrchr-statep s str n lst ch)
                   (disjoint x k (sub 32 4 (read-sp s)) 16))
              (equal (read-mem x (mc-mem (stepn s 4)) k)
                     (read-mem x (mc-mem s) k))))

; from s0 to exit: s0 --> sn.
; base case 1. s0 --> sn, when lst[i] = ch and lst[i] = 0.
(prove-lemma strrchr-s0-sn-base1 (rewrite)
     (implies (and (strrchr-s0p s i* i str n lst ch j* j)
                   (not (equal (get-nth i lst) ch))
                   (equal (get-nth i lst) 0))
              (and (equal (mc-status (stepn s 6)) 'running)
                   (equal (mc-pc (stepn s 6)) (linked-rts-addr s))
                   (equal (read-dn 32 0 (stepn s 6))
                          (index-j str j* j))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 6)))
                          (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 6)))
                          (add 32 (read-an 32 6 s) 8))
                   (equal (read-mem x (mc-mem (stepn s 6)) k)
                          (read-mem x (mc-mem s) k))))
     ((disable index-j)))

(prove-lemma strrchr-s0-sn-rfile-base1 (rewrite)
     (implies (and (strrchr-s0p s i* i str n lst ch j* j)
                   (not (equal (get-nth i lst) ch))
                   (equal (get-nth i lst) 0)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 6)))
                     (read-rn oplen rn (mc-rfile s))))
     ((disable index-j)))

; base case 2: s0 --> sn, when lst[i] =~ ch and lst[i] = 0.
(prove-lemma strrchr-s0-sn-base2 (rewrite)
     (implies (and (strrchr-s0p s i* i str n lst ch j* j)
                   (equal (get-nth i lst) ch)
                   (equal (get-nth i lst) 0))
              (and (equal (mc-status (stepn s 7)) 'running)
                   (equal (mc-pc (stepn s 7)) (linked-rts-addr s))
                   (equal (read-dn 32 0 (stepn s 7)) (add 32 str i*))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 7)))
                          (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 7)))
                          (add 32 (read-an 32 6 s) 8))
                   (equal (read-mem x (mc-mem (stepn s 7)) k)
                          (read-mem x (mc-mem s) k)))))
```

```
     ((disable index-j)))

(prove-lemma strrchr-s0-sn-rfile-base2 (rewrite)
     (implies (and (strrchr-s0p s i* i str n lst ch j* j)
                   (equal (get-nth i lst) ch)
                   (equal (get-nth i lst) 0)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 7)))
                     (read-rn oplen rn (mc-rfile s))))
     ((disable index-j)))

; induction case 1: s0 --> s0, when lst[i] = ch and lst[i] =- 0.
(prove-lemma index-j-la (rewrite)
     (implies j
              (equal (index-j str j* j) (add 32 str j*))))

(prove-lemma strrchr-s0-s0-1 (rewrite)
     (implies (and (strrchr-s0p s i* i str n lst ch j* j)
                   (equal (get-nth i lst) ch)
                   (not (equal (get-nth i lst) 0)))
              (and (strrchr-s0p (stepn s 7) (add 32 i* 1) (add1 i)
                                str n lst ch i* i)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 7)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (linked-a6 (stepn s 7)) (linked-a6 s))
                   (equal (linked-rts-addr (stepn s 7)) (linked-rts-addr s))
                   (equal (read-mem x (mc-mem (stepn s 7)) k)
                          (read-mem x (mc-mem s) k))))
     ((disable index-j)))

(prove-lemma strrchr-s0-s0-rfile-1 (rewrite)
     (implies (and (strrchr-s0p s i* i str n lst ch j* j)
                   (equal (get-nth i lst) ch)
                   (not (equal (get-nth i lst) 0))
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 7)))
                     (read-rn oplen rn (mc-rfile s))))
     ((disable index-j)))

; induction case 2: s0 --> s0, when lst[i] =- ch and lst[i] =- 0.
(prove-lemma strrchr-s0-s0-2 (rewrite)
     (implies (and (strrchr-s0p s i* i str n lst ch j* j)
                   (not (equal (get-nth i lst) ch))
                   (not (equal (get-nth i lst) 0)))
              (and (strrchr-s0p (stepn s 6) (add 32 i* 1) (add1 i)
                                str n lst ch j* j)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 6)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (linked-a6 (stepn s 6)) (linked-a6 s))
                   (equal (linked-rts-addr (stepn s 6)) (linked-rts-addr s))
                   (equal (read-mem x (mc-mem (stepn s 6)) k)
                          (read-mem x (mc-mem s) k))))
     ((disable index-j)))
```

```
(prove-lemma strrchr-s0-s0-rfile-2 (rewrite)
     (implies (and (strrchr-s0p s i* i str n lst ch j* j)
                   (not (equal (get-nth i lst) ch))
                   (not (equal (get-nth i lst) 0))
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 6)))
                     (read-rn oplen rn (mc-rfile s))))
     ((disable index-j)))

; put together. s0 --> exit.
(prove-lemma strrchr-s0p-info (rewrite)
     (implies (strrchr-s0p s i* i str n lst ch j* j)
              (and (equal (lessp i n) t)
                   (numberp i))))

(prove-lemma strrchr-s0p-la (rewrite)
     (not (strrchr-s0p s i* f str n lst ch j* j)))

(prove-lemma strrchr-s0-sn (rewrite)
     (let ((sn (stepn s (strrchr-t1 i n lst ch))))
       (implies (strrchr-s0p s i* i str n lst ch j* j)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn)
                            (if (strrchr i n lst ch j)
                                (add 32 str (strrchr* i* i n lst ch j*))
                              0))
                     (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))
                     (equal (read-mem x (mc-mem sn) k)
                            (read-mem x (mc-mem s) k))))))
     ((induct (strrchr-induct s i* i n lst ch j* j))
      (disable strrchr-s0p read-dn)))

(disable strrchr-s0p-info)

(prove-lemma strrchr-s0-sn-rfile (rewrite)
     (implies
      (and (strrchr-s0p s i* i str n lst ch j* j)
           (d2-7a2-5p rn))
      (equal (read-rn oplen rn (mc-rfile (stepn s (strrchr-t1 i n lst ch))))
             (read-rn oplen rn (mc-rfile s))))
     ((induct (strrchr-induct s i* i n lst ch j* j))
      (disable strrchr-s0p)))

; the correctness of strrchr.
(prove-lemma strrchr-correctness (rewrite)
     (let ((sn (stepn s (strrchr-t n lst ch))))
       (implies (strrchr-statep s str n lst ch)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (read-rn 32 14 (mc-rfile s)))
```

```
                    (equal (read-rn 32 15 (mc-rfile sn))
                           (add 32 (read-sp s) 4))
                    (implies (d2-7a2-5p rn)
                             (equal (read-rn oplen rn (mc-rfile sn))
                                    (read-rn oplen rn (mc-rfile s))))
                    (implies (disjoint x k (sub 32 4 (read-sp s)) 16)
                             (equal (read-mem x (mc-mem sn) k)
                                    (read-mem x (mc-mem s) k)))
                    (equal (read-dn 32 0 sn)
                           (if (strrchr 0 n lst ch f)
                               (add 32 str (strrchr* 0 0 n lst ch f))
                             0)))))
     ((use (strrchr-s-s0))
      (disable strrchr-statep strrchr-s0p read-dn linked-rts-addr linked-a6)))

(disable strrchr-t)

; strrchr* --> strrchr.
(prove-lemma strrchr*-strrchr (rewrite)
     (implies (and (strrchr i n lst ch j)
                   (equal i (nat-to-uint i*))
                   (nat-rangep i* 32)
                   (uint-rangep n 32))
              (equal (nat-to-uint (strrchr* i* i n lst ch j*))
                     (if (equal j f)
                         (strrchr i n lst ch f)
                       (strrchr i n lst ch (nat-to-uint j*))))))
     ((induct (strrchr-induct s i* i n lst ch j* j))))

(prove-lemma strrchr-non-zerop-la ()
     (let ((sn (stepn s (strrchr-t n lst ch))))
       (implies (and (strrchr-statep s str n lst ch)
                     (nat-rangep str 32)
                     (not (equal (nat-to-uint str) 0))
                     (uint-rangep (plus (nat-to-uint str) n) 32)
                     (strrchr 0 n lst ch f))
              (not (equal (nat-to-uint (read-dn 32 0 sn)) 0))))
     ((enable nat-rangep-la)
      (disable strrchr-statep read-dn)))

(prove-lemma strrchr-non-zerop (rewrite)
     (let ((sn (stepn s (strrchr-t n lst ch))))
       (implies (and (strrchr-statep s str n lst ch)
                     (strrchr 0 n lst ch f))
              (not (equal (nat-to-uint (read-dn 32 0 sn)) 0))))
     ((use (strrchr-non-zerop-la))))

(disable strrchr*)

; some properties of the function strrchr.
; see file cstring.events.
```

C.27 The strspn Function

```
;          Proof of the Correctness of the STRSPN Function
#|
This is part of our effort to verify the Berkeley C string library.  The
Berkeley string library is widely used as part of the Berkeley Unix OS.

This is the source code of strspn function in the Berkeley string library.

/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 */
size_t
strspn(s1, s2)
        const char *s1;
        register const char *s2;
{
        register const char *p = s1, *spanp;
        register char c, sc;

        /*
         * Skip any characters in s2, excluding the terminating \0.
         */
cont:
        c = *p++;
        for (spanp = s2; (sc = *spanp++) != 0;)
                if (sc == c)
                        goto cont;
        return (p - 1 - s1);
}

The MC68020 assembly code of the C function strspn on SUN-3 is given as
follows.  This binary is generated by "gcc -O".

0x26e8 <strspn>:        linkw fp,#0
0x26ec <strspn+4>:      movel d2,sp@-
0x26ee <strspn+6>:      movel fp@(8),d2
0x26f2 <strspn+10>:     moveal d2,a1
0x26f4 <strspn+12>:     moveb a1@+,d1
0x26f6 <strspn+14>:     moveal fp@(12),a0
0x26fa <strspn+18>:     bra 0x2700 <strspn+24>
0x26fc <strspn+20>:     cmpb d0,d1
0x26fe <strspn+22>:     beq 0x26f4 <strspn+12>
0x2700 <strspn+24>:     moveb a0@+,d0
0x2702 <strspn+26>:     bne 0x26fc <strspn+20>
0x2704 <strspn+28>:     movel d2,d0
0x2706 <strspn+30>:     addql #1,d0
0x2708 <strspn+32>:     subl a1,d0
0x270a <strspn+34>:     negl d0
0x270c <strspn+36>:     movel fp@(-4),d2
0x2710 <strspn+40>:     unlk fp
0x2712 <strspn+42>:     rts
```

```
The machine code of the above program is:

<strspn>:      0x4e56  0x0000  0x2f02  0x242e  0x0008  0x2242  0x1219  0x206e
<strspn+16>:   0x000c  0x6004  0xb200  0x67f4  0x1018  0x66f8  0x2002  0x5280
<strspn+32>:   0x9089  0x4480  0x242e  0xfffc  0x4e5e  0x4e75

'(78      86      0       0       47      2       36      46
  0       8       34      66      18      25      32      110
  0       12      96      4       178     0       103     244
  16      24      102     248     32      2       82      128
  144     137     68      128     36      46      255     252
  78      94      78      117)
|#

; in the logic, the above program is defined by (strspn-code).
(defn strspn-code ()
  '(78      86      0       0       47      2       36      46
    0       8       34      66      18      25      32      110
    0       12      96      4       178     0       103     244
    16      24      102     248     32      2       82      128
    144     137     68      128     36      46      255     252
    78      94      78      117))

; the computation time of the program.
(defn strspn-t0 (i2 n2 lst2 ch)
  (if (lessp i2 n2)
      (if (equal (get-nth i2 lst2) 0)
          9
        (if (equal (get-nth i2 lst2) ch)
            4
          (splus 4 (strspn-t0 (add1 i2) n2 lst2 ch))))
    0)
  ((lessp (difference n2 i2))))

(defn strspn-t1 (n2 lst2 ch)
  (splus 3 (strspn-t0 0 n2 lst2 ch)))

(defn strspn-t2 (i1 n1 lst1 n2 lst2)
  (if (lessp i1 n1)
      (if (strchr1 0 n2 lst2 (get-nth i1 lst1))
          (splus (strspn-t1 n2 lst2 (get-nth i1 lst1))
                 (strspn-t2 (add1 i1) n1 lst1 n2 lst2))
        (strspn-t1 n2 lst2 (get-nth i1 lst1)))
    0)
  ((lessp (difference n1 i1))))

(defn strspn-t (n1 lst1 n2 lst2)
  (splus 4 (strspn-t2 0 n1 lst1 n2 lst2)))

; two induction hints.
(defn strspn-induct0 (s i2* i2 n2 lst2 ch)
  (if (lessp i2 n2)
      (if (equal (get-nth i2 lst2) 0)
          t
```

```
       (if (equal (get-nth i2 lst2) ch)
           t
         (strspn-induct0 (stepn s 4) (add 32 i2* 1) (add1 i2) n2 lst2 ch)))
    t)
  ((lessp (difference n2 i2))))

(defn strspn-induct1 (s i1* i1 n1 lst1 n2 lst2)
  (if (lessp i1 n1)
      (if (strchr1 0 n2 lst2 (get-nth i1 lst1))
          (strspn-induct1 (stepn s (strspn-t1 n2 lst2 (get-nth i1 lst1)))
                          (add 32 i1* 1) (add1 i1) n1 lst1 n2 lst2)
        t)
    t)
  ((lessp (difference n1 i1))))

; the preconditions of the initial state.
(defn strspn-statep (s str1 n1 lst1 str2 n2 lst2)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (mc-pc s) (mc-mem s) 44)
       (mcode-addrp (mc-pc s) (mc-mem s) (strspn-code))
       (ram-addrp (sub 32 8 (read-sp s)) (mc-mem s) 20)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (sub 32 8 (read-sp s)) 20 str1 n1)
       (disjoint (sub 32 8 (read-sp s)) 20 str2 n2)
       (equal str1 (read-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal str2 (read-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
       (lessp (slen 0 n1 lst1) n1)
       (lessp (slen 0 n2 lst2) n2)
       (numberp n1)
       (numberp n2)
       (uint-rangep n1 32)
       (uint-rangep n2 32)))

(defn strspn-s0p (s i1* i1 str1 n1 lst1 str2 n2 lst2)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 12 (mc-pc s)) (mc-mem s) 44)
       (mcode-addrp (sub 32 12 (mc-pc s)) (mc-mem s) (strspn-code))
       (ram-addrp (sub 32 4 (read-an 32 6 s)) (mc-mem s) 20)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (sub 32 4 (read-an 32 6 s)) 20 str1 n1)
       (disjoint (sub 32 4 (read-an 32 6 s)) 20 str2 n2)
       (equal* (read-an 32 1 s) (add 32 str1 i1*))
       (equal str2 (read-mem (add 32 (read-an 32 6 s) 12) (mc-mem s) 4))
       (equal str1 (read-dn 32 2 s))
       (numberp i1*)
       (nat-rangep i1* 32)
```

```
       (equal i1 (nat-to-uint i1*))
       (lessp (slen i1 n1 lst1) n1)
       (lessp (slen 0 n2 lst2) n2)
       (numberp n1)
       (numberp n2)
       (uint-rangep n1 32)
       (uint-rangep n2 32)))

(defn strspn-s1-1p (s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 24 (mc-pc s)) (mc-mem s) 44)
       (mcode-addrp (sub 32 24 (mc-pc s)) (mc-mem s) (strspn-code))
       (ram-addrp (sub 32 4 (read-an 32 6 s)) (mc-mem s) 20)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (sub 32 4 (read-an 32 6 s)) 20 str2 n2)
       (equal str1 (read-dn 32 2 s))
       (equal str2 (read-mem (add 32 (read-an 32 6 s) 12) (mc-mem s) 4))
       (equal ch (uread-dn 8 1 s))
       (equal* (read-an 32 1 s) (add 32 str1 i1*))
       (equal* (read-an 32 0 s) (add 32 str2 i2*))
       (numberp i1*)
       (nat-rangep i1* 32)
       (equal i1 (nat-to-uint i1*))
       (numberp i2*)
       (nat-rangep i2* 32)
       (equal i2 (nat-to-uint i2*))
       (lessp (slen i2 n2 lst2) n2)
       (numberp n2)
       (uint-rangep n2 32)))

(defn strspn-s1p (s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
  (and (strspn-s1-1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (disjoint (sub 32 4 (read-an 32 6 s)) 20 str1 n1)
       (lessp (slen i1 n1 lst1) n1)
       (lessp (slen 0 n2 lst2) n2)
       (numberp n1)
       (uint-rangep n1 32)))

; from the initial state s to s0: s --> s0.
(prove-lemma strspn-s-s0 ()
     (implies (strspn-statep s str1 n1 lst1 str2 n2 lst2)
              (strspn-s0p (stepn s 4) 0 0 str1 n1 lst1 str2 n2 lst2)))

(prove-lemma strspn-s-s0-else (rewrite)
     (implies (strspn-statep s str1 n1 lst1 str2 n2 lst2)
              (and (equal (linked-rts-addr (stepn s 4)) (rts-addr s))
                   (equal (linked-a6 (stepn s 4)) (read-an 32 6 s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 4)))
                          (sub 32 4 (read-sp s)))
                   (equal (rn-saved (stepn s 4))
```

```
                        (read-rn 32 2 (mc-rfile s))))))

(prove-lemma strspn-s-s0-rfile (rewrite)
     (implies (and (strspn-statep s str1 n1 lst1 str2 n2 lst2)
                   (d3-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 4)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strspn-s-s0-mem (rewrite)
     (implies (and (strspn-statep s str1 n1 lst1 str2 n2 lst2)
                   (disjoint x k (sub 32 8 (read-sp s)) 20))
              (equal (read-mem x (mc-mem (stepn s 4)) k)
                     (read-mem x (mc-mem s) k))))

; loop 0.
; from s0 to s1: s0 --> s1.
(prove-lemma strspn-s0-s1-1 ()
     (implies (strspn-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
              (strspn-s1-1p (stepn s 3) (add 32 i1* 1) (add1 i1) str1 n1
                            lst1 0 0 str2 n2 lst2 (get-nth i1 lst1))))

(prove-lemma strchr1-la ()
     (implies (strchr1 i2 n2 lst2 ch)
              (not (equal ch 0))))

(prove-lemma strspn-s0-s1 ()
     (implies (and (strspn-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                   (strchr1 0 n2 lst2 (get-nth i1 lst1)))
              (strspn-s1p (stepn s 3) (add 32 i1* 1) (add1 i1) str1 n1
                          lst1 0 0 str2 n2 lst2 (get-nth i1 lst1)))
     ((use (strchr1-la (i2 0) (ch (get-nth i1 lst1))))))

(prove-lemma strspn-s0-s1-else (rewrite)
     (let ((s1 (stepn s 3)))
       (implies (strspn-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                (and (equal (read-rn 32 14 (mc-rfile s1))
                            (read-rn 32 14 (mc-rfile s)))
                   (equal (linked-a6 s1) (linked-a6 s))
                   (equal (linked-rts-addr s1) (linked-rts-addr s))
                   (equal (rn-saved s1) (rn-saved s))
                   (equal (read-mem x (mc-mem s1) k)
                          (read-mem x (mc-mem s) k))))))

(prove-lemma strspn-s0-s1-rfile (rewrite)
     (implies (and (strspn-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                   (d3-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 3)))
                     (read-rn oplen rn (mc-rfile s)))))

; loop 1.
; base case 1. from s1 to exit: s1 --> sn, when lst1[i] == 0.
(prove-lemma strspn-s1-sn-base (rewrite)
     (implies (and (strspn-s1-1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                   (equal (get-nth i2 lst2) 0))
```

```
               (and (equal (mc-status (stepn s 9)) 'running)
                    (equal (mc-pc (stepn s 9)) (linked-rts-addr s))
                    (equal (read-dn 32 0 (stepn s 9)) (sub 32 1 i1*))
                    (equal (read-rn 32 14 (mc-rfile (stepn s 9)))
                           (linked-a6 s))
                    (equal (read-rn 32 15 (mc-rfile (stepn s 9)))
                           (add 32 (read-an 32 6 s) 8))
                    (equal (read-mem x (mc-mem (stepn s 9)) k)
                           (read-mem x (mc-mem s) k)))))

(prove-lemma strspn-s1-sn-rfile-base (rewrite)
     (implies (and (strspn-s1-1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                    (equal (get-nth i2 lst2) 0)
                    (leq oplen 32)
                    (d2-7a2-5p rn))
               (equal (read-rn oplen rn (mc-rfile (stepn s 9)))
                      (if (d3-7a2-5p rn)
                          (read-rn oplen rn (mc-rfile s))
                        (head (rn-saved s) oplen)))))

; base case 2.  s1 --> s0, when lst1[i] =- 0  and lst2[j] = ch.
(prove-lemma strspn-s1-s0-base (rewrite)
     (implies (and (strspn-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                    (not (equal (get-nth i2 lst2) 0))
                    (equal (get-nth i2 lst2) ch))
               (and (strspn-s0p (stepn s 4) i1* i1 str1 n1 lst1 str2 n2 lst2)
                    (equal (read-rn 32 14 (mc-rfile (stepn s 4)))
                           (read-rn 32 14 (mc-rfile s)))
                    (equal (linked-a6 (stepn s 4)) (linked-a6 s))
                    (equal (linked-rts-addr (stepn s 4))
                           (linked-rts-addr s))
                    (equal (read-mem x (mc-mem (stepn s 4)) k)
                           (read-mem x (mc-mem s) k)))))

(prove-lemma strspn-s1-s0-rfile-base (rewrite)
     (implies (and (strspn-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                    (not (equal (get-nth i2 lst2) 0))
                    (equal (get-nth i2 lst2) ch)
                    (d3-7a2-5p rn))
               (equal (read-rn oplen rn (mc-rfile (stepn s 4)))
                      (read-rn oplen rn (mc-rfile s)))))

; induction case. s1 --> s1, when lst1[i] =- 0 and lst2[j] =- ch.
(prove-lemma strspn-s1-s1-1 (rewrite)
     (implies (and (strspn-s1-1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                    (not (equal (get-nth i2 lst2) 0))
                    (not (equal (get-nth i2 lst2) ch)))
               (and (strspn-s1-1p (stepn s 4) i1* i1 str1 n1 lst1 (add 32 i2* 1)
                                  (add1 i2) str2 n2 lst2 ch)
                    (equal (read-rn 32 14 (mc-rfile (stepn s 4)))
                           (read-rn 32 14 (mc-rfile s)))
                    (equal (linked-a6 (stepn s 4)) (linked-a6 s))
                    (equal (linked-rts-addr (stepn s 4))
                           (linked-rts-addr s))
```

```
                (equal (rn-saved (stepn s 4)) (rn-saved s))
                (equal (read-mem x (mc-mem (stepn s 4)) k)
                       (read-mem x (mc-mem s) k)))))

(prove-lemma strspn-s1-s1 (rewrite)
     (implies (and (strspn-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                   (not (equal (get-nth i2 lst2) 0))
                   (not (equal (get-nth i2 lst2) ch)))
              (strspn-s1p (stepn s 4) i1* i1 str1 n1 lst1 (add 32 i2* 1)
                          (add1 i2) str2 n2 lst2 ch)))

(prove-lemma strspn-s1-s1-rfile (rewrite)
     (implies (and (strspn-s1-1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                   (not (equal (get-nth i2 lst2) 0))
                   (not (equal (get-nth i2 lst2) ch))
                   (d3-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 4)))
                     (read-rn oplen rn (mc-rfile s)))))

; put together.
; case 1. s1 --> exit, when (not (strchr1 i2 n2 lst2 ch)).
(prove-lemma strspn-s1-1p-info (rewrite)
     (implies (strspn-s1-1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
              (equal (lessp i2 n2) t)))

(prove-lemma strspn-s1-sn (rewrite)
     (let ((sn (stepn s (strspn-t0 i2 n2 lst2 ch))))
       (implies
        (and (strspn-s1-1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
             (not (strchr1 i2 n2 lst2 ch)))
        (and (equal (mc-status sn) 'running)
             (equal (mc-pc sn) (linked-rts-addr s))
             (equal (read-dn 32 0 sn) (sub 32 1 i1*))
             (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
             (equal (read-rn 32 15 (mc-rfile sn)) (add 32 (read-an 32 6 s) 8))
             (equal (read-mem x (mc-mem sn) k)
                    (read-mem x (mc-mem s) k))))))
     ((induct (strspn-induct0 s i2* i2 n2 lst2 ch))
      (disable strspn-s1-1p read-dn)))

(prove-lemma strspn-s1-sn-rfile (rewrite)
     (implies
      (and (strspn-s1-1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
           (not (strchr1 i2 n2 lst2 ch))
           (d2-7a2-5p rn)
           (leq oplen 32))
      (equal (read-rn oplen rn (mc-rfile (stepn s (strspn-t0 i2 n2 lst2 ch))))
             (if (d3-7a2-5p rn)
                 (read-rn oplen rn (mc-rfile s))
               (head (rn-saved s) oplen))))
     ((induct (strspn-induct0 s i2* i2 n2 lst2 ch))
      (disable strspn-s1-1p)))

(disable strspn-s1-1p-info)
```

```
; case 2. s1 --> s0, when (strchr1 i2 n2 lst2 ch).
(prove-lemma strspn-s1p-s1-1p ()
     (implies (strspn-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
              (strspn-s1-1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)))

(prove-lemma strspn-s1-s0 (rewrite)
     (let ((s0 (stepn s (strspn-t0 i2 n2 lst2 ch))))
       (implies (and (strspn-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                     (strchr1 i2 n2 lst2 ch))
                (and (strspn-s0p s0 i1* i1 str1 n1 lst1 str2 n2 lst2)
                     (equal (read-rn 32 14 (mc-rfile s0))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (linked-a6 s0) (linked-a6 s))
                     (equal (linked-rts-addr s0) (linked-rts-addr s))
                     (equal (rn-saved s0) (rn-saved s))
                     (equal (read-mem x (mc-mem s0) k)
                            (read-mem x (mc-mem s) k)))))
     ((induct (strspn-induct0 s i2* i2 n2 lst2 ch))
      (use (strspn-s1p-s1-1p))
      (disable strspn-s0p strspn-s1p strspn-s1-1p)))

(prove-lemma strspn-s1-s0-rfile (rewrite)
     (implies
      (and (strspn-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
           (strchr1 i2 n2 lst2 ch)
           (d3-7a2-5p rn))
      (equal (read-rn oplen rn (mc-rfile (stepn s (strspn-t0 i2 n2 lst2 ch))))
             (read-rn oplen rn (mc-rfile s))))
     ((induct (strspn-induct0 s i2* i2 n2 lst2 ch))
      (use (strspn-s1p-s1-1p))
      (disable strspn-s1p strspn-s1-1p)))

; from s0 --> exit. s0 --> sn.
; base case:  s0 --> sn,  when (not (strchr1 i2 n2 lst2 ch)).
(prove-lemma strspn-s0-sn-base (rewrite)
     (let ((sn (stepn s (strspn-t1 n2 lst2 (get-nth i1 lst1)))))
       (implies (and (strspn-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                     (not (strchr1 0 n2 lst2 (get-nth i1 lst1))))
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) (head i1* 32))
                     (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))
                     (equal (read-mem x (mc-mem sn) k)
                            (read-mem x (mc-mem s) k))))))
     ((use (strspn-s0-s1-1))
      (disable strspn-s0p strspn-s1-1p strchr1 strspn-t0 read-dn)))

(prove-lemma strspn-s0-sn-rfile-base (rewrite)
     (let ((ch (get-nth i1 lst1)))
       (implies
        (and (strspn-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
```

```
            (not (strchr1 0 n2 lst2 ch))
            (leq oplen 32)
            (d2-7a2-5p rn))
       (equal (read-rn oplen rn (mc-rfile (stepn s (strspn-t1 n2 lst2 ch))))
              (if (d3-7a2-5p rn)
                  (read-rn oplen rn (mc-rfile s))
                (head (rn-saved s) oplen)))))
    ((use (strspn-s0-s1-1))
     (disable strspn-s0p strspn-s1-1p strchr1 strspn-t0)))

; induction case: s0 --> s0,  when (strchr1 i2 n2 lst2 ch).
(prove-lemma strspn-s0-s0 (rewrite)
    (let ((s0 (stepn s (strspn-t1 n2 lst2 (get-nth i1 lst1)))))
      (implies (and (strspn-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                    (strchr1 0 n2 lst2 (get-nth i1 lst1)))
               (and (strspn-s0p s0 (add 32 i1* 1) (add1 i1) str1 n1 lst1
                                str2 n2 lst2)
                    (equal (read-rn 32 14 (mc-rfile s0))
                           (read-rn 32 14 (mc-rfile s)))
                    (equal (linked-a6 s0) (linked-a6 s))
                    (equal (linked-rts-addr s0) (linked-rts-addr s))
                    (equal (rn-saved s0) (rn-saved s))
                    (equal (read-mem x (mc-mem s0) k)
                           (read-mem x (mc-mem s) k)))))
    ((use (strspn-s0-s1))
     (disable strspn-s0p strspn-s1p strspn-t0 strchr1)))

(prove-lemma strspn-s0-s0-rfile (rewrite)
    (let ((ch (get-nth i1 lst1)))
      (implies
       (and (strspn-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
            (strchr1 0 n2 lst2 ch)
            (d3-7a2-5p rn))
       (equal (read-rn oplen rn (mc-rfile (stepn s (strspn-t1 n2 lst2 ch))))
              (read-rn oplen rn (mc-rfile s))))))
    ((use (strspn-s0-s1))
     (disable strspn-s0p strspn-s1p strspn-t0 strchr1)))

; put together: s0 --> sn.
(prove-lemma strspn-s0p-info (rewrite)
    (implies (strspn-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
             (and (equal (lessp i1 n1) t)
                  (equal (nat-to-uint (head i1* 32)) (fix i1))))))

(prove-lemma strspn-s0-sn (rewrite)
    (let ((sn (stepn s (strspn-t2 i1 n1 lst1 n2 lst2))))
      (implies (strspn-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
               (and (equal (mc-status sn) 'running)
                    (equal (mc-pc sn) (linked-rts-addr s))
                    (equal (uread-dn 32 0 sn) (strspn i1 n1 lst1 n2 lst2))
                    (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
                    (equal (read-rn 32 15 (mc-rfile sn))
                           (add 32 (read-an 32 6 s) 8))
                    (equal (read-mem x (mc-mem sn) k)
```

```
                          (read-mem x (mc-mem s) k)))))
     ((induct (strspn-induct1 s i1* i1 n1 lst1 n2 lst2))
      (disable strspn-s0p strchr1 strspn-t1 read-dn)))

(prove-lemma strspn-s0-sn-rfile (rewrite)
     (implies
      (and (strspn-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
           (d2-7a2-5p rn)
           (leq oplen 32))
      (equal (read-rn oplen rn
                      (mc-rfile (stepn s (strspn-t2 i1 n1 lst1 n2 lst2))))
             (if (d3-7a2-5p rn)
                 (read-rn oplen rn (mc-rfile s))
               (head (rn-saved s) oplen))))
     ((induct (strspn-induct1 s i1* i1 n1 lst1 n2 lst2))
      (disable strspn-s0p strchr1 strspn-t1)))

; the correctness of strspn.
(prove-lemma strspn-correctness (rewrite)
     (let ((sn (stepn s (strspn-t n1 lst1 n2 lst2))))
       (implies (strspn-statep s str1 n1 lst1 str2 n2 lst2)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile sn)) (read-an 32 6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 7 s) 4))
                     (implies (and (d2-7a2-5p rn)
                                   (leq oplen 32))
                              (equal (read-rn oplen rn (mc-rfile sn))
                                     (read-rn oplen rn (mc-rfile s))))
                     (implies (disjoint x k (sub 32 8 (read-sp s)) 20)
                              (equal (read-mem x (mc-mem sn) k)
                                     (read-mem x (mc-mem s) k)))
                     (equal (uread-dn 32 0 sn) (strspn 0 n1 lst1 n2 lst2))))))
     ((use (strspn-s-s0))
      (disable strspn-statep strspn-s0p linked-rts-addr linked-a6 uread-dn)))

(disable strspn-s0p-info)
(disable strspn-t)

; some properties of strspn.
; see file cstring.events
```

C.28 The strstr Function

```
;          Proof of the Correctness of the STRSTR Function
#|
This is part of our effort to verify the Berkeley string library.  The
Berkeley string library is widely used as part of the Berkeley Unix OS.

This is the source code of strstr function in the Berkeley string library.
```

```
//*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 */
* find pointer to first occurrence of find[] in s[] */
char *
strstr(s, find)
        register const char *s, *find;
{
        register char c, sc;
        register size_t len;

        if ((c = *find++) != 0) {
                len = strlen(find);
                do {
                        do {
                                if ((sc = *s++) == 0)
                                        return (NULL);
                        } while (sc != c);
                } while (strncmp(s, find, len) != 0);
                s--;
        }
        return ((char *)s);
}
```

The MC68020 assembly code of the C function strstr on SUN-3 is given as follows. This binary is generated by "gcc -O".

```
0x2718 <strstr>:        linkw fp,#0
0x271c <strstr+4>:      moveml d2-d3/a2-a3,sp@-
0x2720 <strstr+8>:      moveal fp@(8),a2
0x2724 <strstr+12>:     moveal fp@(12),a3
0x2728 <strstr+16>:     moveb a3@+,d2
0x272a <strstr+18>:     beq 0x275a <strstr+66>
0x272c <strstr+20>:     movel a3,sp@-
0x272e <strstr+22>:     jsr @#0x25b0 <strlen>
0x2734 <strstr+28>:     movel d0,d3
0x2736 <strstr+30>:     addqw #4,sp
0x2738 <strstr+32>:     moveb a2@+,d0
0x273a <strstr+34>:     bne 0x2740 <strstr+40>
0x273c <strstr+36>:     clrl d0
0x273e <strstr+38>:     bra 0x275c <strstr+68>
0x2740 <strstr+40>:     cmpb d0,d2
0x2742 <strstr+42>:     bne 0x2738 <strstr+32>
0x2744 <strstr+44>:     movel d3,sp@-
0x2746 <strstr+46>:     movel a3,sp@-
0x2748 <strstr+48>:     movel a2,sp@-
0x274a <strstr+50>:     jsr @#0x2608 <strncmp>
0x2750 <strstr+56>:     addaw #12,sp
0x2754 <strstr+60>:     tstl d0
0x2756 <strstr+62>:     bne 0x2738 <strstr+32>
0x2758 <strstr+64>:     subqw #1,a2
0x275a <strstr+66>:     movel a2,d0
0x275c <strstr+68>:     moveml fp@(-16),d2-d3/a2-a3
```

```
0x2762 <strstr+74>:     unlk fp
0x2764 <strstr+76>:     rts

The machine code of the above program is:

<strstr>:      0x4e56  0x0000  0x48e7  0x3030  0x246e  0x0008  0x266e  0x000c
<strstr+16>:   0x141b  0x672e  0x2f0b  0x4eb9  0x0000  0x25b0  0x2600  0x584f
<strstr+32>:   0x101a  0x6604  0x4280  0x601c  0xb400  0x66f4  0x2f03  0x2f0b
<strstr+48>:   0x2f0a  0x4eb9  0x0000  0x2608  0xdefc  0x000c  0x4a80  0x66e0
<strstr+64>:   0x534a  0x200a  0x4cee  0x0c0c  0xfff0  0x4e5e  0x4e75

'(78      86      0       0       72      231     48      48
  36      110     0       8       38      110     0       12
  20      27      103     46      47      11      78      185
  0       0       37      176     38      0       88      79
  16      26      102     4       66      128     96      28
  180     0       102     244     47      3       47      11
  47      10      78      185     0       0       38      8
  222     252     0       12      74      128     102     224
  83      74      32      10      76      238     12      12
  255     240     78      94      78      117)
|#

; in the logic, the above program is defined by (strstr-code).
(defn strstr-code ()
  '(78      86      0       0       72      231     48      48
    36      110     0       8       38      110     0       12
    20      27      103     46      47      11      78      185
    -1      -1      -1      -1      38      0       88      79
    16      26      102     4       66      128     96      28
    180     0       102     244     47      3       47      11
    47      10      78      185     -1      -1      -1      -1
    222     252     0       12      74      128     102     224
    83      74      32      10      76      238     12      12
    255     240     78      94      78      117))

(constrain strstr-load (rewrite)
   (equal (strstr-loadp s)
          (and (evenp (strstr-addr))
               (numberp (strstr-addr))
               (nat-rangep (strstr-addr) 32)
               (rom-addrp (strstr-addr) (mc-mem s) 78)
               (mcode-addrp (strstr-addr) (mc-mem s) (strstr-code))
               (strlen-loadp s)
               (strncmp-loadp s)
               (equal (pc-read-mem (add 32 (strstr-addr) 24) (mc-mem s) 4)
                      (strlen-addr))
               (equal (pc-read-mem (add 32 (strstr-addr) 52) (mc-mem s) 4)
                      (strncmp-addr))))
   ((strstr-loadp (lambda (s) f))
    (strstr-addr (lambda () 1))))

(prove-lemma stepn-strstr-loadp (rewrite)
     (equal (strstr-loadp (stepn s n))
```

```
           (strstr-loadp s)))

; the computation time of the program.
(defn strstr-t0 (n2 lst2)
  (splus 8 (splus (strlen-t (sub1 n2) (cdr lst2)) 2)))

(defn strstr-t1 (i n1 lst1 lst2)
  (if (lessp i n1)
      (if (equal (get-nth i lst1) 0)
          7
        (if (equal (get-nth i lst1) (get-nth 0 lst2))
            8
          (splus 4 (strstr-t1 (add1 i) n1 lst1 lst2))))
    0)
  ((lessp (difference n1 i))))

(defn strstr-t2 (i n1 lst1 lst2 len)
  (let ((j (strchr1 i n1 lst1 (get-nth 0 lst2))))
    (if (numberp j)
        (if (equal (strncmp len (mcdr (add1 j) lst1) (mcdr 1 lst2)) 0)
            (splus (strstr-t1 i n1 lst1 lst2)
                   (splus (strncmp-t len (mcdr (add1 j) lst1) (mcdr 1 lst2))
                          8))
          (splus (strstr-t1 i n1 lst1 lst2)
                 (splus (strncmp-t len (mcdr (add1 j) lst1) (mcdr 1 lst2))
                        3)))
      (strstr-t1 i n1 lst1 lst2))))

(defn strstr-t3 (i n1 lst1 lst2 len)
  (if (lessp i n1)
      (let ((j (strchr1 i n1 lst1 (get-nth 0 lst2))))
        (if (numberp j)
            (if (equal (strncmp len (mcdr (add1 j) lst1) (mcdr 1 lst2)) 0)
                (strstr-t2 i n1 lst1 lst2 len)
              (splus (strstr-t2 i n1 lst1 lst2 len)
                     (strstr-t3 (add1 j) n1 lst1 lst2 len)))
          (strstr-t1 i n1 lst1 lst2)))
    0)
  ((lessp (difference n1 i))))

(defn strstr-t (n1 lst1 n2 lst2)
  (if (equal (get-nth 0 lst2) 0)
      10
    (splus (strstr-t0 n2 lst2)
           (strstr-t3 0 n1 lst1 lst2 (strlen 0 (sub1 n2) (mcdr 1 lst2))))))

; two induction hints.
(defn strstr-induct1 (s i* i n1 lst1 lst2)
  (if (lessp i n1)
      (if (equal (get-nth i lst1) 0)
          t
        (if (equal (get-nth i lst1) (get-nth 0 lst2))
            t
          (strstr-induct1 (stepn s 4) (add 32 i* 1) (add1 i) n1 lst1 lst2)))
```

```
    t)
  ((lessp (difference n1 i))))

(defn strstr-induct2 (s i* i n1 lst1 lst2 len)
  (if (lessp i n1)
      (let ((j* (strchr1* i* i n1 lst1 (get-nth 0 lst2)))
            (j (strchr1 i n1 lst1 (get-nth 0 lst2))))
        (if (numberp j)
            (if (equal (strncmp len (mcdr (add1 j) lst1) (mcdr 1 lst2)) 0)
                t
              (strstr-induct2 (stepn s (strstr-t2 i n1 lst1 lst2 len))
                              (add 32 j* 1) (add1 j) n1 lst1 lst2 len))
          t))
    t)
  ((lessp (difference n1 i))))

; the preconditions of the initial state.
(defn strstr-statep (s str1 n1 lst1 str2 n2 lst2)
  (and (equal (mc-status s) 'running)
       (strstr-loadp s)
       (equal (mc-pc s) (strstr-addr))
       (ram-addrp (sub 32 48 (read-sp s)) (mc-mem s) 60)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint str1 n1 (sub 32 48 (read-sp s)) 60)
       (disjoint str2 n2 (sub 32 48 (read-sp s)) 60)
       (equal str1 (read-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal str2 (read-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
       (lessp (slen 0 n1 lst1) n1)
       (lessp (slen 0 n2 lst2) n2)
       (not (equal (nat-to-uint str1) 0))
       (not (zerop n1))
       (uint-rangep (plus (nat-to-uint str1) n1) 32)
       (not (zerop n2))
       (uint-rangep n2 32)))

; the intermediate state right before the call to strlen.
(defn strstr-s0p (s str1 n1 lst1 str2 n2 lst2)
  (and (equal (mc-status s) 'running)
       (strstr-loadp s)
       (equal (mc-pc s) (strlen-addr))
       (equal (rts-addr s) (add 32 (strstr-addr) 28))
       (ram-addrp (sub 32 44 (read-an 32 6 s)) (mc-mem s) 60)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint str1 n1 (sub 32 44 (read-an 32 6 s)) 60)
       (disjoint str2 n2 (sub 32 44 (read-an 32 6 s)) 60)
       (equal str1 (read-an 32 2 s))
       (equal (nat-to-uint (read-dn 8 2 s)) (get-nth 0 lst2))
       (equal* (read-an 32 3 s) (add 32 str2 1))
```

```
      (equal* (read-sp s) (sub 32 24 (read-an 32 6 s)))
      (equal (read-an 32 3 s) (read-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
      (lessp (slen 0 n1 lst1) n1)
      (lessp (slen 1 n2 lst2) n2)
      (not (zerop n1))
      (uint-rangep n1 32)
      (not (zerop n2))
      (uint-rangep n2 32)))

; the intermediate state returned from the call to strlen.
(defn strstr-s1p (s str1 n1 lst1 str2 n2 lst2 len)
  (and (equal (mc-status s) 'running)
       (strstr-loadp s)
       (equal (mc-pc s) (add 32 (strstr-addr) 28))
       (ram-addrp (sub 32 44 (read-an 32 6 s)) (mc-mem s) 60)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint str1 n1 (sub 32 44 (read-an 32 6 s)) 60)
       (disjoint str2 n2 (sub 32 44 (read-an 32 6 s)) 60)
       (equal str1 (read-an 32 2 s))
       (equal (nat-to-uint (read-dn 8 2 s)) (get-nth 0 lst2))
       (equal* (read-an 32 3 s) (add 32 str2 1))
       (equal* (read-sp s) (sub 32 20 (read-an 32 6 s)))
       (equal len (uread-dn 32 0 s))
       (lessp (slen 0 n1 lst1) n1)
       (lessp (slen 1 n2 lst2) n2)
       (numberp n1)
       (uint-rangep n1 32)
       (numberp n2)
       (uint-rangep n2 32)))

; the intermediate state right before the outer loop.
(defn strstr-s2p (s i* i str1 n1 lst1 str2 n2 lst2 len)
  (and (equal (mc-status s) 'running)
       (strstr-loadp s)
       (equal (mc-pc s) (add 32 (strstr-addr) 32))
       (ram-addrp (sub 32 44 (read-an 32 6 s)) (mc-mem s) 60)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint str1 n1 (sub 32 44 (read-an 32 6 s)) 60)
       (disjoint str2 n2 (sub 32 44 (read-an 32 6 s)) 60)
       (equal (uread-dn 8 2 s) (get-nth 0 lst2))
       (equal* (read-an 32 2 s) (add 32 str1 i*))
       (equal* (read-an 32 3 s) (add 32 str2 1))
       (equal* (read-sp s) (sub 32 16 (read-an 32 6 s)))
       (equal len (uread-dn 32 3 s))
       (lessp (slen i n1 lst1) n1)
       (lessp (slen 1 n2 lst2) n2)
       (numberp i*)
       (nat-rangep i* 32)
```

```
      (equal i (nat-to-uint i*))
      (not (zerop n1))
      (uint-rangep n1 32)
      (not (zerop n2))
      (uint-rangep n2 32)))

; the intermediate state right before the call to strncmp.
(defn strstr-s3p (s i* i str1 n1 lst1 str2 n2 lst2 len)
  (and (equal (mc-status s) 'running)
      (strstr-loadp s)
      (equal (mc-pc s) (strncmp-addr))
      (equal (rts-addr s) (add 32 (strstr-addr) 56))
      (ram-addrp (sub 32 44 (read-an 32 6 s)) (mc-mem s) 60)
      (ram-addrp str1 (mc-mem s) n1)
      (mem-lst 1 str1 (mc-mem s) n1 lst1)
      (ram-addrp str2 (mc-mem s) n2)
      (mem-lst 1 str2 (mc-mem s) n2 lst2)
      (disjoint (sub 32 44 (read-an 32 6 s)) 60 str1 n1)
      (disjoint (sub 32 44 (read-an 32 6 s)) 60 str2 n2)
      (equal (uread-dn 8 2 s) (get-nth 0 lst2))
      (equal* (read-an 32 2 s) (add 32 str1 i*))
      (equal* (read-an 32 3 s) (add 32 str2 1))
      (equal* (read-sp s) (sub 32 32 (read-an 32 6 s)))
      (equal len (uread-dn 32 3 s))
      (equal (read-an 32 2 s) (read-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
      (equal (read-an 32 3 s) (read-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
      (equal (read-dn 32 3 s) (read-mem (add 32 (read-sp s) 12) (mc-mem s) 4))
      (lessp (slen i n1 lst1) n1)
      (lessp (slen 1 n2 lst2) n2)
      (numberp i*)
      (nat-rangep i* 32)
      (equal i (nat-to-uint i*))
      (not (zerop n1))
      (uint-rangep n1 32)
      (not (zerop n2))
      (uint-rangep n2 32)))

; the intermediate state right after the call to strncmp.
(defn strstr-s4p (s i* i str1 n1 lst1 str2 n2 lst2 len)
  (and (equal (mc-status s) 'running)
      (strstr-loadp s)
      (equal (mc-pc s) (add 32 (strstr-addr) 56))
      (ram-addrp (sub 32 44 (read-an 32 6 s)) (mc-mem s) 60)
      (ram-addrp str1 (mc-mem s) n1)
      (mem-lst 1 str1 (mc-mem s) n1 lst1)
      (ram-addrp str2 (mc-mem s) n2)
      (mem-lst 1 str2 (mc-mem s) n2 lst2)
      (disjoint str1 n1 (sub 32 44 (read-an 32 6 s)) 60)
      (disjoint str2 n2 (sub 32 44 (read-an 32 6 s)) 60)
      (equal (iread-dn 32 0 s) (strncmp len (mcdr i lst1) (mcdr 1 lst2)))
      (equal (uread-dn 8 2 s) (get-nth 0 lst2))
      (equal* (read-an 32 2 s) (add 32 str1 i*))
      (equal* (read-an 32 3 s) (add 32 str2 1))
      (equal* (read-sp s) (sub 32 28 (read-an 32 6 s)))
```

```
       (equal len (uread-dn 32 3 s))
       (lessp (slen i n1 lst1) n1)
       (lessp (slen 1 n2 lst2) n2)
       (numberp i*)
       (nat-rangep i* 32)
       (equal i (nat-to-uint i*))
       (not (zerop n1))
       (uint-rangep n1 32)
       (not (zerop n2))
       (uint-rangep n2 32)))

; from the initial state to exit:  s --> sn, when lst2[0] == 0.
(prove-lemma strstr-s-sn (rewrite)
     (implies (and (strstr-statep s str1 n1 lst1 str2 n2 lst2)
                   (equal (get-nth 0 lst2) 0))
              (and (equal (mc-status (stepn s 10)) 'running)
                   (equal (mc-pc (stepn s 10)) (rts-addr s))
                   (equal (read-dn 32 0 (stepn s 10)) str1)
                   (equal (read-rn 32 15 (mc-rfile (stepn s 10)))
                          (add 32 (read-an 32 7 s) 4))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 10)))
                          (read-an 32 6 s)))))

(prove-lemma strstr-s-sn-rfile (rewrite)
     (implies (and (strstr-statep s str1 n1 lst1 str2 n2 lst2)
                   (equal (get-nth 0 lst2) 0)
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 10)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strstr-s-sn-mem (rewrite)
     (implies (and (strstr-statep s str1 n1 lst1 str2 n2 lst2)
                   (equal (get-nth 0 lst2) 0)
                   (disjoint x k (sub 32 48 (read-sp s)) 60))
              (equal (read-mem x (mc-mem (stepn s 10)) k)
                     (read-mem x (mc-mem s) k))))

; from the initial state to s0.  s --> s0.
(prove-lemma strstr-s-s0 ()
     (implies (and (strstr-statep s str1 n1 lst1 str2 n2 lst2)
                   (not (equal (get-nth 0 lst2) 0)))
              (strstr-s0p (stepn s 8) str1 n1 lst1 str2 n2 lst2)))

(prove-lemma strstr-s-s0-else (rewrite)
     (implies (and (strstr-statep s str1 n1 lst1 str2 n2 lst2)
                   (not (equal (get-nth 0 lst2) 0)))
              (and (equal (linked-rts-addr (stepn s 8)) (rts-addr s))
                   (equal (linked-a6 (stepn s 8)) (read-an 32 6 s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 8)))
                          (sub 32 4 (read-sp s)))
                   (equal (movem-saved (stepn s 8) 4 16 4)
                          (readm-rn 32 '(2 3 10 11) (mc-rfile s))))))
```

```
(prove-lemma strstr-s-s0-rfile (rewrite)
     (implies (and (strstr-statep s str1 n1 lst1 str2 n2 lst2)
                   (not (equal (get-nth 0 lst2) 0))
                   (d4-7a4-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 8)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strstr-s-s0-mem (rewrite)
     (implies (and (strstr-statep s str1 n1 lst1 str2 n2 lst2)
                   (not (equal (get-nth 0 lst2) 0))
                   (disjoint x k (sub 32 48 (read-sp s)) 60))
              (equal (read-mem x (mc-mem (stepn s 8)) k)
                     (read-mem x (mc-mem s) k))))

; from s0 to s1:  s0 --> s1.  by strlen.
(prove-lemma strstr-s0p-strlen-statep ()
     (implies (strstr-s0p s str1 n1 lst1 str2 n2 lst2)
              (strlen-statep s (add 32 str2 1) (sub1 n2) (mcdr 1 lst2))))

(prove-lemma strstr-s0-s1 ()
     (let ((s1 (stepn s (strlen-t (sub1 n2) (mcdr 1 lst2)))))
       (implies (strstr-s0p s str1 n1 lst1 str2 n2 lst2)
                (strstr-s1p s1 str1 n1 lst1 str2 n2 lst2
                            (strlen 0 (sub1 n2) (mcdr 1 lst2))))))
     ((use (strstr-s0p-strlen-statep))
      (disable strlen-statep strlen-load strncmp-load uread-dn)))

(prove-lemma strstr-s0-s1-else (rewrite)
     (let ((s1 (stepn s (strlen-t (sub1 n2) (cdr lst2)))))
       (implies (strstr-s0p s str1 n1 lst1 str2 n2 lst2)
                (and (equal (read-rn 32 14 (mc-rfile s1))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (linked-rts-addr s1) (linked-rts-addr s))
                     (equal (linked-a6 s1) (linked-a6 s))
                     (equal (movem-saved s1 4 16 4)
                            (movem-saved s 4 16 4))))))
     ((use (strstr-s0p-strlen-statep))
      (disable strlen-statep strstr-load)))

(prove-lemma strstr-s0-s1-rfile (rewrite)
     (let ((s1 (stepn s (strlen-t (sub1 n2) (cdr lst2)))))
       (implies (and (strstr-s0p s str1 n1 lst1 str2 n2 lst2)
                     (d2-7a2-5p rn))
                (equal (read-rn oplen rn (mc-rfile s1))
                       (read-rn oplen rn (mc-rfile s))))))
     ((use (strstr-s0p-strlen-statep))
      (disable strstr-s0p strlen-statep)))

(prove-lemma strstr-s0-s1-mem (rewrite)
     (let ((s1 (stepn s (strlen-t (sub1 n2) (cdr lst2)))))
       (implies (and (strstr-s0p s str1 n1 lst1 str2 n2 lst2)
                     (disjoint x k (sub 32 44 (read-an 32 6 s)) 60))
                (equal (read-mem x (mc-mem s1) k)
                       (read-mem x (mc-mem s) k)))))
```

```
     ((use (strstr-s0p-strlen-statep))
      (disable strlen-statep strstr-load)))

; from s1 to s2:  s1 --> s2.
(prove-lemma strstr-s1-s2 (rewrite)
     (implies (strstr-s1p s str1 n1 lst1 str2 n2 lst2 len)
              (strstr-s2p (stepn s 2) 0 0 str1 n1 lst1 str2 n2 lst2 len)))

(prove-lemma strstr-s1-s2-else (rewrite)
     (implies (strstr-s1p s str1 n1 lst1 str2 n2 lst2 len)
              (and (equal (linked-rts-addr (stepn s 2)) (linked-rts-addr s))
                   (equal (linked-a6 (stepn s 2)) (linked-a6 s))
                   (equal (read-rn oplen 14 (mc-rfile (stepn s 2)))
                          (read-rn oplen 14 (mc-rfile s)))
                   (equal (movem-saved (stepn s 2) 4 16 4)
                          (movem-saved s 4 16 4))
                   (equal (read-mem x (mc-mem (stepn s 2)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma strstr-s1-s2-rfile (rewrite)
     (implies (and (strstr-s1p s str1 n1 lst1 str2 n2 lst2 len)
                   (d4-7a4-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 2)))
                     (read-rn oplen rn (mc-rfile s)))))

; from s2 to exit:  s2 --> sn, when lst1[i] == 0.
(prove-lemma strstr-s2-sn-base (rewrite)
     (implies (and (strstr-s2p s i* i str1 n1 lst1 str2 n2 lst2 len)
                   (equal (get-nth i lst1) 0))
              (and (equal (mc-status (stepn s 7)) 'running)
                   (equal (mc-pc (stepn s 7))
                          (linked-rts-addr s))
                   (equal (read-dn 32 0 (stepn s 7)) 0)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 7)))
                          (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 7)))
                          (add 32 (read-an 32 6 s) 8))
                   (equal (read-mem x (mc-mem (stepn s 7)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma strstr-s2-sn-base-rfile (rewrite)
     (implies (and (strstr-s2p s i* i str1 n1 lst1 str2 n2 lst2 len)
                   (equal (get-nth i lst1) 0)
                   (d2-7a2-5p rn)
                   (leq oplen 32))
              (equal (read-rn oplen rn (mc-rfile (stepn s 7)))
                     (if (d4-7a4-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (get-vlst oplen 0 rn '(2 3 10 11)
                                 (movem-saved s 4 16 4))))))

; from s2 to s3:  s2 --> s3, when lst1[i] =~ 0 and lst1[i] == lst2[0].
(prove-lemma strstr-s2-s3-base (rewrite)
     (implies (and (strstr-s2p s i* i str1 n1 lst1 str2 n2 lst2 len)
```

```
                (not (equal (get-nth i lst1) 0))
                (equal (get-nth i lst1) (get-nth 0 lst2)))
           (and (strstr-s3p (stepn s 8) (add 32 i* 1) (add1 i) str1 n1 lst1
                            str2 n2 lst2 len)
                (equal (read-rn 32 14 (mc-rfile (stepn s 8)))
                       (read-rn 32 14 (mc-rfile s)))
                (equal (linked-a6 (stepn s 8)) (linked-a6 s))
                (equal (linked-rts-addr (stepn s 8)) (linked-rts-addr s))
                (equal (movem-saved (stepn s 8) 4 16 4)
                       (movem-saved s 4 16 4)))))

(prove-lemma strstr-s2-s3-base-rfile (rewrite)
     (implies (and (strstr-s2p s i* i str1 n1 lst1 str2 n2 lst2 len)
                (not (equal (get-nth i lst1) 0))
                (equal (get-nth i lst1) (get-nth 0 lst2))
                (d4-7a4-5p rn))
           (equal (read-rn oplen rn (mc-rfile (stepn s 8)))
                  (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strstr-s2-s3-base-mem (rewrite)
     (implies (and (strstr-s2p s i* i str1 n1 lst1 str2 n2 lst2 len)
                (not (equal (get-nth i lst1) 0))
                (equal (get-nth i lst1) (get-nth 0 lst2))
                (disjoint x k (sub 32 44 (read-an 32 6 s)) 60))
           (equal (read-mem x (mc-mem (stepn s 8)) k)
                  (read-mem x (mc-mem s) k))))

; from s2 to s2:  s2 --> s2.
(prove-lemma strstr-s2-s2 (rewrite)
     (implies (and (strstr-s2p s i* i str1 n1 lst1 str2 n2 lst2 len)
                (not (equal (get-nth i lst1) 0))
                (not (equal (get-nth i lst1) (get-nth 0 lst2))))
           (and (strstr-s2p (stepn s 4) (add 32 i* 1) (add1 i) str1 n1
                            lst1 str2 n2 lst2 len)
                (equal (read-rn 32 14 (mc-rfile (stepn s 4)))
                       (read-rn 32 14 (mc-rfile s)))
                (equal (linked-rts-addr (stepn s 4)) (linked-rts-addr s))
                (equal (rts-addr (stepn s 4)) (rts-addr s))
                (equal (movem-saved (stepn s 4) 4 16 4)
                       (movem-saved s 4 16 4))
                (equal (read-mem x (mc-mem (stepn s 4)) k)
                       (read-mem x (mc-mem s) k)))))

(prove-lemma strstr-s2-s2-rfile (rewrite)
     (implies (and (strstr-s2p s i* i str1 n1 lst1 str2 n2 lst2 len)
                (not (equal (get-nth i lst1) 0))
                (not (equal (get-nth i lst1) (get-nth 0 lst2)))
                (d4-7a4-5p rn))
           (equal (read-rn oplen rn (mc-rfile (stepn s 4)))
                  (read-rn oplen rn (mc-rfile s)))))

; from s3 to s4:  s3 --> s4. The call to strncmp.
(prove-lemma strstr-s3p-strncmp-statep ()
     (implies (strstr-s3p s i* i str1 n1 lst1 str2 n2 lst2 len)
```

```
             (strncmp-statep s (add 32 str1 i*) (difference n1 i)
                             (mcdr i lst1) (add 32 str2 1) (sub1 n2)
                             (mcdr 1 lst2) len)))

(prove-lemma strstr-s3-s4 ()
     (let ((s4 (stepn s (strncmp-t len (mcdr i lst1) (mcdr 1 lst2)))))
       (implies (strstr-s3p s i* i str1 n1 lst1 str2 n2 lst2 len)
                (strstr-s4p s4 i* i str1 n1 lst1 str2 n2 lst2 len)))
     ((use (strstr-s3p-strncmp-statep))
      (disable strncmp-statep strlen-load strncmp-load iread-dn)))

(prove-lemma strstr-s3-s4-else (rewrite)
     (let ((s4 (stepn s (strncmp-t len (mcdr i lst1) (mcdr 1 lst2)))))
       (implies (strstr-s3p s i* i str1 n1 lst1 str2 n2 lst2 len)
                (and (equal (read-rn 32 14 (mc-rfile s4))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (linked-rts-addr s4) (linked-rts-addr s))
                     (equal (linked-a6 s4) (linked-a6 s))
                     (equal (movem-saved s4 4 16 4)
                            (movem-saved s 4 16 4)))))
     ((use (strstr-s3p-strncmp-statep))
      (disable strncmp-statep strstr-load)))

(prove-lemma strstr-s3-s4-rfile (rewrite)
     (let ((s4 (stepn s (strncmp-t len (mcdr i lst1) (mcdr 1 lst2)))))
       (implies (and (strstr-s3p s i* i str1 n1 lst1 str2 n2 lst2 len)
                     (d2-7a2-5p rn)
                     (leq oplen 32))
                (equal (read-rn oplen rn (mc-rfile s4))
                       (read-rn oplen rn (mc-rfile s))))))
     ((use (strstr-s3p-strncmp-statep))
      (disable strncmp-statep strstr-s3p)))

(prove-lemma strstr-s3-s4-mem (rewrite)
     (let ((s4 (stepn s (strncmp-t len (mcdr i lst1) (mcdr 1 lst2)))))
       (implies (and (strstr-s3p s i* i str1 n1 lst1 str2 n2 lst2 len)
                     (disjoint x k (sub 32 44 (read-an 32 6 s)) 60))
                (equal (read-mem x (mc-mem s4) k)
                       (read-mem x (mc-mem s) k)))))
     ((use (strstr-s3p-strncmp-statep))
      (disable strncmp-statep strlen-load strncmp-load)))

; from s4 to exit:  s4 --> sn, when strncmp == 0.
(prove-lemma strstr-s4-sn (rewrite)
     (implies (and (strstr-s4p s i* i str1 n1 lst1 str2 n2 lst2 len)
                   (equal (strncmp len (mcdr i lst1) (mcdr 1 lst2)) 0))
              (and (equal (mc-status (stepn s 8)) 'running)
                   (equal (mc-pc (stepn s 8)) (linked-rts-addr s))
                   (equal (read-dn 32 0 (stepn s 8))
                          (add 32 str1 (sub 32 1 i*)))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 8)))
                          (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 8)))
                          (add 32 (read-an 32 6 s) 8))
```

```
                (equal (read-mem x (mc-mem (stepn s 8)) k)
                       (read-mem x (mc-mem s) k))))
     ((disable strncmp)))

(prove-lemma strstr-s4-sn-rfile (rewrite)
     (implies (and (strstr-s4p s i* i str1 n1 lst1 str2 n2 lst2 len)
                   (equal (strncmp len (mcdr i lst1) (mcdr 1 lst2)) 0)
                   (d2-7a2-5p rn)
                   (leq oplen 32))
              (equal (read-rn oplen rn (mc-rfile (stepn s 8)))
                     (if (d4-7a4-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (get-vlst oplen 0 rn '(2 3 10 11)
                                 (movem-saved s 4 16 4)))))
     ((disable strncmp)))

; from s4 to s2: s4 --> s2, when strncmp =~ 0.
(prove-lemma strstr-s4-s2 (rewrite)
     (implies (and (strstr-s4p s i* i str1 n1 lst1 str2 n2 lst2 len)
                   (not (equal (strncmp len (mcdr i lst1) (mcdr 1 lst2)) 0)))
              (and (strstr-s2p (stepn s 3) i* i str1 n1 lst1 str2 n2 lst2 len)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 3)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (linked-rts-addr (stepn s 3)) (linked-rts-addr s))
                   (equal (linked-a6 (stepn s 3)) (linked-a6 s))
                   (equal (movem-saved (stepn s 3) 4 16 4)
                          (movem-saved s 4 16 4))
                   (equal (read-mem x (mc-mem (stepn s 3)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma strstr-s4-s2-rfile (rewrite)
     (implies (and (strstr-s4p s i* i str1 n1 lst1 str2 n2 lst2 len)
                   (not (equal (strncmp len (mcdr i lst1) (mcdr 1 lst2)) 0))
                   (d4-7a4-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 3)))
                     (read-rn oplen rn (mc-rfile s)))))

; put together: s2 --> s3.
(prove-lemma strstr-s2p-info (rewrite)
     (implies (strstr-s2p s i* i str1 n1 lst1 str2 n2 lst2 len)
              (and (numberp i)
                   (equal (lessp i n1) t))))

(prove-lemma strstr-s2-s3 ()
     (implies (and (strstr-s2p s i* i str1 n1 lst1 str2 n2 lst2 len)
                   (numberp (strchr1 i n1 lst1 (get-nth 0 lst2))))
              (strstr-s3p (stepn s (strstr-t1 i n1 lst1 lst2))
                          (add 32 (strchr1* i* i n1 lst1 (get-nth 0 lst2)) 1)
                          (add1 (strchr1 i n1 lst1 (get-nth 0 lst2)))
                          str1 n1 lst1 str2 n2 lst2 len))
     ((induct (strstr-induct1 s i* i n1 lst1 lst2))
      (disable strstr-s2p strstr-s3p)))

(prove-lemma strstr-s2-s3-else (rewrite)
```

```
     (let ((s3 (stepn s (strstr-t1 i n1 lst1 lst2))))
       (implies (and (strstr-s2p s i* i str1 n1 lst1 str2 n2 lst2 len)
                     (numberp (strchr1 i n1 lst1 (get-nth 0 lst2))))
                (and (equal (read-rn 32 14 (mc-rfile s3))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (linked-rts-addr s3) (linked-rts-addr s))
                     (equal (linked-a6 s3) (linked-a6 s))
                     (equal (movem-saved s3 4 16 4) (movem-saved s 4 16 4)))))
     ((induct (strstr-induct1 s i* i n1 lst1 lst2))
      (disable strstr-s2p)))

(prove-lemma strstr-s2-s3-rfile (rewrite)
     (let ((s3 (stepn s (strstr-t1 i n1 lst1 lst2))))
       (implies (and (strstr-s2p s i* i str1 n1 lst1 str2 n2 lst2 len)
                     (numberp (strchr1 i n1 lst1 (get-nth 0 lst2)))
                     (d4-7a4-5p rn))
                (equal (read-rn oplen rn (mc-rfile s3))
                       (read-rn oplen rn (mc-rfile s)))))
     ((induct (strstr-induct1 s i* i n1 lst1 lst2))
      (disable strstr-s2p)))

(prove-lemma strstr-s2-s3-mem (rewrite)
     (let ((s3 (stepn s (strstr-t1 i n1 lst1 lst2))))
       (implies (and (strstr-s2p s i* i str1 n1 lst1 str2 n2 lst2 len)
                     (numberp (strchr1 i n1 lst1 (get-nth 0 lst2)))
                     (disjoint x k (sub 32 44 (read-an 32 6 s)) 60))
                (equal (read-mem x (mc-mem s3) k)
                       (read-mem x (mc-mem s) k))))
     ((induct (strstr-induct1 s i* i n1 lst1 lst2))
      (disable strstr-s2p)))

; put together: s2 --> exit.
(prove-lemma strstr-s2-sn (rewrite)
     (let ((sn (stepn s (strstr-t1 i n1 lst1 lst2))))
       (implies (and (strstr-s2p s i* i str1 n1 lst1 str2 n2 lst2 len)
                     (not (numberp (strchr1 i n1 lst1 (get-nth 0 lst2)))))
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) 0)
                     (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))
                     (equal (read-mem x (mc-mem sn) k)
                            (read-mem x (mc-mem s) k)))))
     ((induct (strstr-induct1 s i* i n1 lst1 lst2))
      (disable strstr-s2p read-dn)))

(prove-lemma strstr-s2-sn-rfile (rewrite)
     (let ((sn (stepn s (strstr-t1 i n1 lst1 lst2))))
       (implies (and (strstr-s2p s i* i str1 n1 lst1 str2 n2 lst2 len)
                     (not (numberp (strchr1 i n1 lst1 (get-nth 0 lst2))))
                     (d2-7a2-5p rn)
                     (leq oplen 32))
                (equal (read-rn oplen rn (mc-rfile sn))
```

```
                    (if (d4-7a4-5p rn)
                        (read-rn oplen rn (mc-rfile s))
                      (get-vlst oplen 0 rn '(2 3 10 11)
                                (movem-saved s 4 16 4))))))
     ((induct (strstr-induct1 s i* i n1 lst1 lst2))
      (disable strstr-s2p)))

; put together: s --> s1.
(prove-lemma strstr-s-s2 ()
     (let ((s2 (stepn s (strstr-t0 n2 lst2))))
       (implies (and (strstr-statep s str1 n1 lst1 str2 n2 lst2)
                     (not (equal (get-nth 0 lst2) 0)))
                (strstr-s2p s2 0 0 str1 n1 lst1 str2 n2 lst2
                            (strlen 0 (sub1 n2) (mcdr 1 lst2))))))
     ((use (strstr-s-s0) (strstr-s0-s1 (s (stepn s 8))))
      (disable strstr-statep strstr-s0p strstr-s1p strstr-s2p)))

(prove-lemma strstr-s-s2-else (rewrite)
     (let ((s2 (stepn s (strstr-t0 n2 lst2))))
       (implies (and (strstr-statep s str1 n1 lst1 str2 n2 lst2)
                     (not (equal (get-nth 0 lst2) 0)))
                (and (equal (linked-rts-addr s2) (rts-addr s))
                     (equal (linked-a6 s2) (read-an 32 6 s))
                     (equal (read-rn 32 14 (mc-rfile s2))
                            (sub 32 4 (read-sp s)))
                     (equal (movem-saved s2 4 16 4)
                            (readm-rn 32 '(2 3 10 11) (mc-rfile s)))))))
     ((use (strstr-s-s0) (strstr-s0-s1 (s (stepn s 8))))
      (disable strstr-statep strstr-s0p strstr-s1p strstr-s2p linked-rts-addr
               rts-addr linked-a6 movem-saved strlen)))

(prove-lemma strstr-s-s2-rfile (rewrite)
     (let ((s2 (stepn s (strstr-t0 n2 lst2))))
       (implies (and (strstr-statep s str1 n1 lst1 str2 n2 lst2)
                     (not (equal (get-nth 0 lst2) 0))
                     (d4-7a4-5p rn))
                (equal (read-rn oplen rn (mc-rfile s2))
                       (read-rn oplen rn (mc-rfile s))))))
     ((use (strstr-s-s0) (strstr-s0-s1 (s (stepn s 8))))
      (disable strstr-statep strstr-s0p strstr-s1p strstr-s2p strlen)))

(prove-lemma strstr-s-s2-mem (rewrite)
     (let ((s2 (stepn s (strstr-t0 n2 lst2))))
       (implies (and (strstr-statep s str1 n1 lst1 str2 n2 lst2)
                     (not (equal (get-nth 0 lst2) 0))
                     (disjoint x k (sub 32 48 (read-sp s)) 60))
                (equal (read-mem x (mc-mem s2) k)
                       (read-mem x (mc-mem s) k)))))
     ((use (strstr-s-s0) (strstr-s0-s1 (s (stepn s 8))))
      (disable strstr-statep strstr-s0p strstr-s1p strstr-s2p strlen)))

; put together: s2 --> s2.
(prove-lemma strstr-s2-s2-1 (rewrite)
     (let ((s2 (stepn s (strstr-t2 i n1 lst1 lst2 len)))
```

```
          (j* (strchr1* i* i n1 lst1 (get-nth 0 lst2)))
          (j (strchr1 i n1 lst1 (get-nth 0 lst2))))
       (implies
        (and (strstr-s2p s i* i str1 n1 lst1 str2 n2 lst2 len)
             (numberp j)
             (not (equal (strncmp len (mcdr (add1 j) lst1) (mcdr 1 lst2)) 0)))
        (and (strstr-s2p s2 (add 32 j* 1) (add1 j) str1 n1 lst1
                         str2 n2 lst2 len)
             (equal (read-rn 32 14 (mc-rfile s2)) (read-rn 32 14 (mc-rfile s)))
             (equal (linked-rts-addr s2) (linked-rts-addr s))
             (equal (linked-a6 s2) (linked-a6 s))
             (equal (movem-saved s2 4 16 4) (movem-saved s 4 16 4)))))
     ((use (strstr-s2-s3)
           (strstr-s3-s4
            (s (stepn s (strstr-t1 i n1 lst1 lst2)))
            (i* (add 32 (strchr1* i* i n1 lst1 (get-nth 0 lst2)) 1))
            (i (add1 (strchr1 i n1 lst1 (get-nth 0 lst2))))))
      (disable strstr-s2p strstr-s3p strstr-s4p linked-rts-addr
               rts-addr linked-a6 movem-saved mcdr)))

(prove-lemma strstr-s2-s2-rfile-1 (rewrite)
     (let ((s2 (stepn s (strstr-t2 i n1 lst1 lst2 len)))
           (j (strchr1 i n1 lst1 (get-nth 0 lst2))))
       (implies
        (and (strstr-s2p s i* i str1 n1 lst1 str2 n2 lst2 len)
             (numberp j)
             (not (equal (strncmp len (mcdr (add1 j) lst1) (mcdr 1 lst2)) 0))
             (leq oplen 32)
             (d4-7a4-5p rn))
        (equal (read-rn oplen rn (mc-rfile s2))
               (read-rn oplen rn (mc-rfile s)))))
     ((use (strstr-s2-s3)
           (strstr-s3-s4
            (s (stepn s (strstr-t1 i n1 lst1 lst2)))
            (i* (add 32 (strchr1* i* i n1 lst1 (get-nth 0 lst2)) 1))
            (i (add1 (strchr1 i n1 lst1 (get-nth 0 lst2))))))
      (disable strstr-s2p strstr-s3p strstr-s4p mcdr)))

(prove-lemma strstr-s2-s2-mem-1 (rewrite)
     (let ((s2 (stepn s (strstr-t2 i n1 lst1 lst2 len)))
           (j (strchr1 i n1 lst1 (get-nth 0 lst2))))
       (implies
        (and (strstr-s2p s i* i str1 n1 lst1 str2 n2 lst2 len)
             (numberp j)
             (not (equal (strncmp len (mcdr (add1 j) lst1) (mcdr 1 lst2)) 0))
             (disjoint x k (sub 32 44 (read-an 32 6 s)) 60))
        (equal (read-mem x (mc-mem s2) k) (read-mem x (mc-mem s) k))))
     ((use (strstr-s2-s3)
           (strstr-s3-s4
            (s (stepn s (strstr-t1 i n1 lst1 lst2)))
            (i* (add 32 (strchr1* i* i n1 lst1 (get-nth 0 lst2)) 1))
            (i (add1 (strchr1 i n1 lst1 (get-nth 0 lst2))))))
      (disable strstr-s2p strstr-s3p strstr-s4p mcdr)))
```

```
; put together: s2 --> sn.
(prove-lemma strstr-s2-sn-base-1 (rewrite)
     (let ((sn (stepn s (strstr-t2 i n1 lst1 lst2 len)))
           (j* (strchr1* i* i n1 lst1 (get-nth 0 lst2)))
           (j (strchr1 i n1 lst1 (get-nth 0 lst2))))
       (implies
        (and (strstr-s2p s i* i str1 n1 lst1 str2 n2 lst2 len)
             (numberp j)
             (equal (strncmp len (mcdr (add1 j) lst1) (mcdr 1 lst2)) 0))
        (and (equal (mc-status sn) 'running)
             (equal (mc-pc sn) (linked-rts-addr s))
             (equal (read-dn 32 0 sn) (add 32 str1 j*))
             (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
             (equal (read-rn 32 15 (mc-rfile sn))
                    (add 32 (read-an 32 6 s) 8))))))
     ((use (strstr-s2-s3)
           (strstr-s3-s4
            (s (stepn s (strstr-t1 i n1 lst1 lst2)))
            (i* (add 32 (strchr1* i* i n1 lst1 (get-nth 0 lst2)) 1))
            (i (add1 (strchr1 i n1 lst1 (get-nth 0 lst2))))))
      (disable strstr-s2p strstr-s3p strstr-s4p linked-rts-addr
               rts-addr linked-a6 movem-saved mcdr read-dn)))

(prove-lemma strstr-s2-sn-base-rfile-1 (rewrite)
     (let ((sn (stepn s (strstr-t2 i n1 lst1 lst2 len)))
           (j (strchr1 i n1 lst1 (get-nth 0 lst2))))
       (implies
        (and (strstr-s2p s i* i str1 n1 lst1 str2 n2 lst2 len)
             (numberp j)
             (equal (strncmp len (mcdr (add1 j) lst1) (mcdr 1 lst2)) 0)
             (leq oplen 32)
             (d2-7a2-5p rn))
        (equal (read-rn oplen rn (mc-rfile sn))
               (if (d4-7a4-5p rn)
                   (read-rn oplen rn (mc-rfile s))
                 (get-vlst oplen 0 rn '(2 3 10 11) (movem-saved s 4 16 4)))))))
     ((use (strstr-s2-s3)
           (strstr-s3-s4
            (s (stepn s (strstr-t1 i n1 lst1 lst2)))
            (i* (add 32 (strchr1* i* i n1 lst1 (get-nth 0 lst2)) 1))
            (i (add1 (strchr1 i n1 lst1 (get-nth 0 lst2))))))
      (disable strstr-s2p strstr-s3p strstr-s4p mcdr)))

(prove-lemma strstr-s2-sn-base-mem-1 (rewrite)
     (let ((sn (stepn s (strstr-t2 i n1 lst1 lst2 len)))
           (j (strchr1 i n1 lst1 (get-nth 0 lst2))))
       (implies
        (and (strstr-s2p s i* i str1 n1 lst1 str2 n2 lst2 len)
             (numberp j)
             (equal (strncmp len (mcdr (add1 j) lst1) (mcdr 1 lst2)) 0)
             (disjoint x k (sub 32 44 (read-an 32 6 s)) 60))
        (equal (read-mem x (mc-mem sn) k) (read-mem x (mc-mem s) k))))
     ((use (strstr-s2-s3)
           (strstr-s3-s4
```

```
           (s (stepn s (strstr-t1 i n1 lst1 lst2)))
           (i* (add 32 (strchr1* i* i n1 lst1 (get-nth 0 lst2)) 1))
           (i (add1 (strchr1 i n1 lst1 (get-nth 0 lst2))))))
      (disable strstr-s2p strstr-s3p strstr-s4p mcdr)))

; put together: s2 --> sn.
(prove-lemma strstr-s2-sn-2 (rewrite)
     (let ((sn (stepn s (strstr-t3 i n1 lst1 lst2 len))))
       (implies
        (strstr-s2p s i* i str1 n1 lst1 str2 n2 lst2 len)
        (and (equal (mc-status sn) 'running)
             (equal (mc-pc sn) (linked-rts-addr s))
             (equal (read-dn 32 0 sn)
                    (if (strstr1 i n1 lst1 n2 lst2 len)
                        (add 32 str1 (strstr1* i* i n1 lst1 n2 lst2 len))
                      0))
             (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
             (equal (read-rn 32 15 (mc-rfile sn))
                    (add 32 (read-an 32 6 s) 8)))))
     ((induct (strstr-induct2 s i* i n1 lst1 lst2 len))
      (disable strstr-s2p strchr1 strstr-t2 read-dn)))

(prove-lemma strstr-s2-sn-rfile-2 (rewrite)
     (let ((sn (stepn s (strstr-t3 i n1 lst1 lst2 len))))
       (implies (and (strstr-s2p s i* i str1 n1 lst1 str2 n2 lst2 len)
                     (d2-7a2-5p rn)
                     (leq oplen 32))
                (equal (read-rn oplen rn (mc-rfile sn))
                     (if (d4-7a4-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (get-vlst oplen 0 rn '(2 3 10 11)
                                 (movem-saved s 4 16 4)))))))
     ((induct (strstr-induct2 s i* i n1 lst1 lst2 len))
      (disable strstr-s2p strchr1 strstr-t2)))

(prove-lemma strstr-s2-sn-mem-2 (rewrite)
     (let ((sn (stepn s (strstr-t3 i n1 lst1 lst2 len))))
       (implies (and (strstr-s2p s i* i str1 n1 lst1 str2 n2 lst2 len)
                     (disjoint x k (sub 32 44 (read-an 32 6 s)) 60))
                (equal (read-mem x (mc-mem sn) k)
                       (read-mem x (mc-mem s) k))))
     ((induct (strstr-induct2 s i* i n1 lst1 lst2 len))
      (disable strstr-s2p strchr1 strstr-t2)))

(disable strstr-s2p-info)

; the correctness of strstr.
(prove-lemma strstr-statep-info (rewrite)
     (implies (strstr-statep s str1 n1 lst1 str2 n2 lst2)
              (and (numberp str1)
                   (nat-rangep str1 32))))

(prove-lemma strstr-correctness (rewrite)
     (let ((sn (stepn s (strstr-t n1 lst1 n2 lst2))))
```

```
       (implies (strstr-statep s str1 n1 lst1 str2 n2 lst2)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-sp s) 4))
                     (implies (and (d2-7a2-5p rn)
                                   (leq oplen 32))
                              (equal (read-rn oplen rn (mc-rfile sn))
                                     (read-rn oplen rn (mc-rfile s))))
                     (implies (disjoint x k (sub 32 48 (read-sp s)) 60)
                              (equal (read-mem x (mc-mem sn) k)
                                     (read-mem x (mc-mem s) k)))
                     (equal (read-dn 32 0 sn)
                            (if (strstr n1 lst1 n2 lst2)
                                (add 32 str1 (strstr* n1 lst1 n2 lst2))
                              0)))))
     ((use (strstr-s-s2))
      (disable strstr-statep strstr-s2p strstr-t0 strlen strstr1 strstr1*
               linked-rts-addr linked-a6 read-dn)))

(disable strstr-statep-info)
(disable strstr-t)

; strstr* --> strstr.
(prove-lemma strchr1*-strchr1 (rewrite)
     (implies (and (strchr1 i n lst ch)
                   (equal i (nat-to-uint i*))
                   (nat-rangep i* 32)
                   (uint-rangep n 32))
              (equal (nat-to-uint (strchr1* i* i n lst ch))
                     (strchr1 i n lst ch)))
     ((induct (strchr1* i* i n lst ch))))

(prove-lemma strstr*-strstr (rewrite)
     (implies (and (strstr1 i n1 lst1 n2 lst2 len)
                   (equal i (nat-to-uint i*))
                   (nat-rangep i* 32)
                   (uint-rangep n1 32))
              (equal (nat-to-uint (strstr1* i* i n1 lst1 n2 lst2 len))
                     (strstr1 i n1 lst1 n2 lst2 len)))
     ((induct (strstr1* i* i n1 lst1 n2 lst2 len))
      (enable nat-rangep-la)))

(prove-lemma strstr-non-zerop-la ()
     (let ((sn (stepn s (strstr-t n1 lst1 n2 lst2))))
       (implies (and (strstr-statep s str1 n1 lst1 str2 n2 lst2)
                     (nat-rangep str1 32)
                     (not (equal (nat-to-uint str1) 0))
                     (uint-rangep (plus (nat-to-uint str1) n1) 32)
                     (numberp str1)
                     (strstr n1 lst1 n2 lst2))
              (not (equal (nat-to-uint (read-dn 32 0 sn)) 0)))))
```

```
     ((enable nat-rangep-la)
      (disable strstr-statep read-dn)))

(prove-lemma strstr-non-zerop (rewrite)
     (let ((sn (stepn s (strstr-t n1 lst1 n2 lst2))))
       (implies (and (strstr-statep s str1 n1 lst1 str2 n2 lst2)
                     (strstr n1 lst1 n2 lst2))
              (not (equal (nat-to-uint (read-dn 32 0 sn)) 0))))
     ((use (strstr-non-zerop-la))))

(disable strstr*)

; some properties of strstr.
; see the file cstring.events.
```

C.29 The strtok Function

```
;          Proof of the Correctness of the STRTOK Function
#|
This is part of our effort to verify the Berkeley string library.  The
Berkeley string library is widely used as part of the Berkeley Unix OS.

This is the source code of strtok function in the Berkeley string library.

/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 */
char *
strtok(s, delim)
        register char *s;
        register const char *delim;
{
        register const char *spanp;
        register int c, sc;
        char *tok;
        static char *last;


        if (s == NULL && (s = last) == NULL)
                return (NULL);

        /*
         * Skip (span) leading delimiters (s += strspn(s, delim), sort of).
         */
cont:
        c = *s++;
        for (spanp = delim; (sc = *spanp++) != 0;) {
                if (c == sc)
                        goto cont;
        }
```

```
        if (c == 0) {           /* no non-delimiter characters */
                last = NULL;
                return (NULL);
        }
        tok = s - 1;

        /*
         * Scan token (scan for delimiters: s += strcspn(s, delim), sort of).
         * Note that delim must have one NUL; we stop if we see that, too.
         */
        for (;;) {
                c = *s++;
                spanp = delim;
                do {
                        if ((sc = *spanp++) == c) {
                                if (c == 0)
                                        s = NULL;
                                else
                                        s[-1] = 0;
                                last = s;
                                return (tok);
                        }
                } while (sc != 0);
        }
        /* NOTREACHED */
}
```

The MC68020 assembly code of the C function strtok on SUN-3 is given as follows. This binary is generated by "gcc -O".

```
0x2768 <strtok>:        linkw fp,#0
0x276c <strtok+4>:      moveml d2-d3,sp@-
0x2770 <strtok+8>:      moveal fp@(8),a1
0x2774 <strtok+12>:     movel fp@(12),d3
0x2778 <strtok+16>:     tstl a1
0x277a <strtok+18>:     bne 0x278a <strtok+34>
0x277c <strtok+20>:     moveal @#0x20098 <edata>,a1
0x2782 <strtok+26>:     tstl a1
0x2784 <strtok+28>:     bne 0x278a <strtok+34>
0x2786 <strtok+30>:     clrl d0
0x2788 <strtok+32>:     bra 0x27d8 <strtok+112>
0x278a <strtok+34>:     moveb a1@+,d1
0x278c <strtok+36>:     extbl d1
0x278e <strtok+38>:     moveal d3,a0
0x2790 <strtok+40>:     bra 0x2796 <strtok+46>
0x2792 <strtok+42>:     cmpl d1,d0
0x2794 <strtok+44>:     beq 0x278a <strtok+34>
0x2796 <strtok+46>:     moveb a0@+,d0
0x2798 <strtok+48>:     extbl d0
0x279a <strtok+50>:     bne 0x2792 <strtok+42>
0x279c <strtok+52>:     tstl d1
0x279e <strtok+54>:     bne 0x27aa <strtok+66>
0x27a0 <strtok+56>:     clrl @#0x20098 <edata>
0x27a6 <strtok+62>:     clrl d0
```

```
0x27a8 <strtok+64>:     bra 0x27d8 <strtok+112>
0x27aa <strtok+66>:     movel a1,d2
0x27ac <strtok+68>:     subl #1,d2
0x27ae <strtok+70>:     moveb a1@+,d1
0x27b0 <strtok+72>:     extbl d1
0x27b2 <strtok+74>:     moveal d3,a0
0x27b4 <strtok+76>:     moveb a0@+,d0
0x27b6 <strtok+78>:     extbl d0
0x27b8 <strtok+80>:     cmpl d0,d1
0x27ba <strtok+82>:     bne 0x27d2 <strtok+106>
0x27bc <strtok+84>:     tstl d1
0x27be <strtok+86>:     bne 0x27c4 <strtok+92>
0x27c0 <strtok+88>:     subal a1,a1
0x27c2 <strtok+90>:     bra 0x27c8 <strtok+96>
0x27c4 <strtok+92>:     clrb a1@(-1)
0x27c8 <strtok+96>:     movel a1,@#0x20098 <edata>
0x27ce <strtok+102>:    movel d2,d0
0x27d0 <strtok+104>:    bra 0x27d8 <strtok+112>
0x27d2 <strtok+106>:    tstl d0
0x27d4 <strtok+108>:    bne 0x27b4 <strtok+76>
0x27d6 <strtok+110>:    bra 0x27ae <strtok+70>
0x27d8 <strtok+112>:    moveml fp@(-8),d2-d3
0x27de <strtok+118>:    unlk fp
0x27e0 <strtok+120>:    rts

The machine code of the above program is:

<strtok>:      0x4e56  0x0000  0x48e7  0x3000  0x226e  0x0008  0x262e  0x000c
<strtok+16>:   0x4a89  0x660e  0x2279  0x0002  0x0098  0x4a89  0x6604  0x4280
<strtok+32>:   0x604e  0x1219  0x49c1  0x2043  0x6004  0xb081  0x67f4  0x1018
<strtok+48>:   0x49c0  0x66f6  0x4a81  0x660a  0x42b9  0x0002  0x0098  0x4280
<strtok+64>:   0x602e  0x2409  0x5382  0x1219  0x49c1  0x2043  0x1018  0x49c0
<strtok+80>:   0xb280  0x6616  0x4a81  0x6604  0x93c9  0x6004  0x4229  0xffff
<strtok+96>:   0x23c9  0x0002  0x0098  0x2002  0x6006  0x4a80  0x66de  0x60d6
<strtok+112>:  0x4cee  0x000c  0xfff8  0x4e5e  0x4e75

'(78      86      0       0       72      231     48      0
  34      110     0       8       38      46      0       12
  74      137     102     14      34      121     0       2
  0       152     74      137     102     4       66      128
  96      78      18      25      73      193     32      67
  96      4       176     129     103     244     16      24
  73      192     102     246     74      129     102     10
  66      185     0       2       0       152     66      128
  96      46      36      9       83      130     18      25
  73      193     32      67      16      24      73      192
  178     128     102     22      74      129     102     4
  147     201     96      4       66      41      255     255
  35      201     0       2       0       152     32      2
  96      6       74      128     102     222     96      214
  76      238     0       12      255     248     78      94
  78      117)
|#
```

```
;  in the logic, the above program is defined by (strtok-code).
(defn strtok-code ()
  '(78      86      0       0       72      231     48      0
    34      110     0       8       38      46      0       12
    74      137     102     14      34      121     -1      -1
    -1      -1      74      137     102     4       66      128
    96      78      18      25      73      193     32      67
    96      4       176     129     103     244     16      24
    73      192     102     246     74      129     102     10
    66      185     -1      -1      -1      -1      66      128
    96      46      36      9       83      130     18      25
    73      193     32      67      16      24      73      192
    178     128     102     22      74      129     102     4
    147     201     96      4       66      41      255     255
    35      201     -1      -1      -1      -1      32      2
    96      6       74      128     102     222     96      214
    76      238     0       12      255     248     78      94
    78      117))

(constrain strtok-load (rewrite)
   (equal (strtok-loadp s)
          (and (evenp (strtok-addr))
               (numberp (strtok-addr))
               (nat-rangep (strtok-addr) 32)
               (numberp (strtok-last-addr))
               (nat-rangep (strtok-last-addr) 32)
               (ram-addrp (strtok-last-addr) (mc-mem s) 4)
               (rom-addrp (strtok-addr) (mc-mem s) 122)
               (mcode-addrp (strtok-addr) (mc-mem s) (strtok-code))
               (equal (pc-read-mem (add 32 (strtok-addr) 22) (mc-mem s) 4)
                      (strtok-last-addr))
               (equal (pc-read-mem (add 32 (strtok-addr) 58) (mc-mem s) 4)
                      (strtok-last-addr))
               (equal (pc-read-mem (add 32 (strtok-addr) 98) (mc-mem s) 4)
                      (strtok-last-addr))))
   ((strtok-loadp (lambda (s) f))
    (strtok-addr (lambda () 1))
    (strtok-last-addr (lambda () 10))))

(prove-lemma stepn-strtok-loadp (rewrite)
     (equal (strtok-loadp (stepn s n))
            (strtok-loadp s)))

; the computation time of the program.
(defn strtok-t0 (i2 n2 lst2 ch)
  (if (lessp i2 n2)
      (if (equal (get-nth i2 lst2) 0)
          3
        (if (equal (get-nth i2 lst2) ch)
            5
          (splus 5 (strtok-t0 (add1 i2) n2 lst2 ch))))
    0)
  ((lessp (difference n2 i2))))
```

```
(defn strtok-t1 (n2 lst2 ch)
  (splus 4 (strtok-t0 0 n2 lst2 ch)))

(defn strtok-t2 (i1 n1 lst1 n2 lst2)
  (if (lessp i1 n1)
      (if (strchr1 0 n2 lst2 (get-nth i1 lst1))
          (splus (strtok-t1 n2 lst2 (get-nth i1 lst1))
                 (strtok-t2 (add1 i1) n1 lst1 n2 lst2))
        (strtok-t1 n2 lst2 (get-nth i1 lst1)))
    0)
  ((lessp (difference n1 i1))))

(defn strtok-t3 (i2 n2 lst2 ch)
  (if (lessp i2 n2)
      (if (equal ch (get-nth i2 lst2))
          (if (equal ch 0) 14 13)
        (if (equal (get-nth i2 lst2) 0)
            7
          (splus 6 (strtok-t3 (add1 i2) n2 lst2 ch))))
    0)
  ((lessp (difference n2 i2))))

(defn strtok-t4 (n2 lst2 ch)
  (splus 3 (strtok-t3 0 n2 lst2 ch)))

(defn strtok-t5 (i1 n1 lst1 n2 lst2)
  (if (lessp i1 n1)
      (if (strchr 0 n2 lst2 (get-nth i1 lst1))
          (strtok-t4 n2 lst2 (get-nth i1 lst1))
        (splus (strtok-t4 n2 lst2 (get-nth i1 lst1))
               (strtok-t5 (add1 i1) n1 lst1 n2 lst2)))
    0)
  ((lessp (difference n1 i1))))

(defn strtok-t6 (i1 n1 lst1 n2 lst2)
  (if (equal (get-nth i1 lst1) 0)
      8
    (splus 4 (strtok-t5 (add1 i1) n1 lst1 n2 lst2))))

(defn strtok-t (str1 last n1 lst1 n2 lst2)
  (if (equal (nat-to-uint str1) 0)
      (if (equal (nat-to-uint last) 0)
          14
        (splus 9
               (splus (strtok-t2 0 n1 lst1 n2 lst2)
                      (strtok-t6 (strspn 0 n1 lst1 n2 lst2) n1 lst1 n2 lst2))))
    (splus 6
           (splus (strtok-t2 0 n1 lst1 n2 lst2)
                  (strtok-t6 (strspn 0 n1 lst1 n2 lst2) n1 lst1 n2 lst2)))))

; two induction hints.
(defn strtok-induct0 (s i2* i2 n2 lst2 ch)
  (if (lessp i2 n2)
      (if (equal (get-nth i2 lst2) 0)
```

```
          t
        (if (equal (get-nth i2 lst2) ch)
            t
          (strtok-induct0 (stepn s 5) (add 32 i2* 1) (add1 i2) n2 lst2 ch)))
    t)
  ((lessp (difference n2 i2))))

(defn strtok-induct1 (s i1* i1 n1 lst1 n2 lst2)
  (if (lessp i1 n1)
      (if (strchr1 0 n2 lst2 (get-nth i1 lst1))
          (strtok-induct1 (stepn s (strtok-t1 n2 lst2 (get-nth i1 lst1)))
                          (add 32 i1* 1) (add1 i1) n1 lst1 n2 lst2)
        t)
    t)
  ((lessp (difference n1 i1))))

(defn strtok-induct2 (s i2* i2 n2 lst2 ch)
  (if (lessp i2 n2)
      (if (equal ch (get-nth i2 lst2))
          t
        (if (equal (get-nth i2 lst2) 0)
            t
          (strtok-induct2 (stepn s 6) (add 32 i2* 1) (add1 i2) n2 lst2 ch)))
    t)
  ((lessp (difference n2 i2))))

(defn strtok-induct3 (s i1* i1 n1 lst1 n2 lst2)
  (if (lessp i1 n1)
      (if (strchr 0 n2 lst2 (get-nth i1 lst1))
          t
        (strtok-induct3 (stepn s (strtok-t4 n2 lst2 (get-nth i1 lst1)))
                        (add 32 i1* 1) (add1 i1) n1 lst1 n2 lst2))
    t)
  ((lessp (difference n1 i1))))

; the preconditions of the initial state.
(defn strtok-statep (s str1 n1 lst1 str2 n2 lst2)
  (and (equal (mc-status s) 'running)
       (strtok-loadp s)
       (equal (mc-pc s) (strtok-addr))
       (ram-addrp (sub 32 12 (read-sp s)) (mc-mem s) 24)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint str2 n2 (sub 32 12 (read-sp s)) 24)
       (disjoint (strtok-last-addr) 4 (sub 32 12 (read-sp s)) 24)
       (equal str1 (read-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal str2 (read-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
       (if (equal (nat-to-uint str1) 0)
           (let ((last (read-mem (strtok-last-addr) (mc-mem s) 4)))
             (if (equal (nat-to-uint last) 0)
                 t
               (and (ram-addrp last (mc-mem s) n1)
                    (mem-lst 1 last (mc-mem s) n1 lst1)
                    (disjoint last n1 (sub 32 12 (read-sp s)) 24)
```

```
                    (disjoint last n1 (strtok-last-addr) 4))))
          (and (ram-addrp str1 (mc-mem s) n1)
               (mem-lst 1 str1 (mc-mem s) n1 lst1)
               (disjoint str1 n1 (sub 32 12 (read-sp s)) 24)
               (disjoint str1 n1 (strtok-last-addr) 4)))
        (lessp (slen 0 n1 lst1) n1)
        (lessp (slen 0 n2 lst2) n2)
        (numberp n1)
        (uint-rangep n1 32)
        (numberp n2)
        (uint-rangep n2 32)))

; intermediate states.
(defn strtok-s0p (s i1* i1 str1 n1 lst1 str2 n2 lst2)
  (and (equal (mc-status s) 'running)
       (strtok-loadp s)
       (equal (mc-pc s) (add 32 (strtok-addr) 34))
       (ram-addrp (sub 32 8 (read-an 32 6 s)) (mc-mem s) 24)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (strtok-last-addr) 4 (sub 32 8 (read-an 32 6 s)) 24)
       (disjoint str1 n1 (strtok-last-addr) 4)
       (disjoint str1 n1 (sub 32 8 (read-an 32 6 s)) 24)
       (disjoint str2 n2 (sub 32 8 (read-an 32 6 s)) 24)
       (equal* (read-an 32 1 s) (add 32 str1 i1*))
       (equal str2 (read-dn 32 3 s))
       (numberp i1*)
       (nat-rangep i1* 32)
       (equal i1 (nat-to-uint i1*))
       (lessp (slen i1 n1 lst1) n1)
       (lessp (slen 0 n2 lst2) n2)
       (numberp n1)
       (uint-rangep n1 32)
       (numberp n2)
       (uint-rangep n2 32)))

(defn strtok-s1p (s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
  (and (equal (mc-status s) 'running)
       (strtok-loadp s)
       (equal (mc-pc s) (add 32 (strtok-addr) 46))
       (ram-addrp (sub 32 8 (read-an 32 6 s)) (mc-mem s) 24)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (strtok-last-addr) 4 (sub 32 8 (read-an 32 6 s)) 24)
       (disjoint str1 n1 (strtok-last-addr) 4)
       (disjoint str1 n1 (sub 32 8 (read-an 32 6 s)) 24)
       (disjoint str2 n2 (sub 32 8 (read-an 32 6 s)) 24)
       (equal str2 (read-dn 32 3 s))
       (equal ch (uread-dn 8 1 s))
       (equal* (read-dn 32 1 s) (ext 8 (read-dn 8 1 s) 32))
```

```
      (equal* (read-an 32 1 s) (add 32 str1 (add 32 i1* 1)))
      (equal* (read-an 32 0 s) (add 32 str2 i2*))
      (numberp i1*)
      (nat-rangep i1* 32)
      (equal i1 (nat-to-uint i1*))
      (numberp i2*)
      (nat-rangep i2* 32)
      (equal i2 (nat-to-uint i2*))
      (lessp (slen 0 n2 lst2) n2)
      (lessp (slen i1 n1 lst1) n1)
      (lessp (slen i2 n2 lst2) n2)
      (numberp n1)
      (uint-rangep n1 32)
      (numberp n2)
      (uint-rangep n2 32)))

(defn strtok-s2p (s i1* i1 str1 n1 lst1 str2 n2 lst2 ch)
  (and (equal (mc-status s) 'running)
       (strtok-loadp s)
       (equal (mc-pc s) (add 32 (strtok-addr) 52))
       (ram-addrp (sub 32 8 (read-an 32 6 s)) (mc-mem s) 24)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (strtok-last-addr) 4 (sub 32 8 (read-an 32 6 s)) 24)
       (disjoint str1 n1 (strtok-last-addr) 4)
       (disjoint str1 n1 (sub 32 8 (read-an 32 6 s)) 24)
       (equal* (read-dn 32 1 s) (ext 8 (read-dn 8 1 s) 32))
       (equal* (read-an 32 1 s) (add 32 str1 (add 32 i1* 1)))
       (equal str2 (read-dn 32 3 s))
       (equal ch (uread-dn 8 1 s))
       (numberp i1*)
       (nat-rangep i1* 32)
       (equal i1 (nat-to-uint i1*))
       (lessp (slen i1 n1 lst1) n1)
       (lessp (slen 0 n2 lst2) n2)
       (numberp n1)
       (uint-rangep n1 32)
       (numberp n2)
       (uint-rangep n2 32)))

(defn strtok-s3p (s i1* i1 str1 n1 lst1 str2 n2 lst2 tok)
  (and (equal (mc-status s) 'running)
       (strtok-loadp s)
       (equal (mc-pc s) (add 32 (strtok-addr) 70))
       (ram-addrp (sub 32 8 (read-an 32 6 s)) (mc-mem s) 24)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (strtok-last-addr) 4 (sub 32 8 (read-an 32 6 s)) 24)
       (disjoint str1 n1 (strtok-last-addr) 4)
       (disjoint str1 n1 (sub 32 8 (read-an 32 6 s)) 24)
```

```
       (equal str2 (read-dn 32 3 s))
       (equal tok (read-dn 32 2 s))
       (equal* (read-an 32 1 s) (add 32 str1 i1*))
       (numberp i1*)
       (nat-rangep i1* 32)
       (equal i1 (nat-to-uint i1*))
       (lessp (slen i1 n1 lst1) n1)
       (lessp (slen 0 n2 lst2) n2)
       (numberp n1)
       (uint-rangep n1 32)
       (numberp n2)
       (uint-rangep n2 32)))

(defn strtok-s4p (s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch tok)
  (and (equal (mc-status s) 'running)
       (strtok-loadp s)
       (equal (mc-pc s) (add 32 (strtok-addr) 76))
       (ram-addrp (sub 32 8 (read-an 32 6 s)) (mc-mem s) 24)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (strtok-last-addr) 4 (sub 32 8 (read-an 32 6 s)) 24)
       (disjoint str1 n1 (strtok-last-addr) 4)
       (disjoint (sub 32 8 (read-an 32 6 s)) 24 str1 n1)
       (equal* (read-an 32 1 s) (add 32 str1 (add 32 i1* 1)))
       (equal* (read-an 32 0 s) (add 32 str2 i2*))
       (equal str2 (read-dn 32 3 s))
       (equal ch (uread-dn 8 1 s))
       (equal tok (read-dn 32 2 s))
       (equal* (read-dn 32 1 s) (ext 8 (read-dn 8 1 s) 32))
       (numberp i1*)
       (nat-rangep i1* 32)
       (equal i1 (nat-to-uint i1*))
       (numberp i2*)
       (nat-rangep i2* 32)
       (equal i2 (nat-to-uint i2*))
       (lessp (slen 0 n2 lst2) n2)
       (lessp (slen i1 n1 lst1) n1)
       (lessp (slen i2 n2 lst2) n2)
       (numberp n1)
       (uint-rangep n1 32)
       (numberp n2)
       (uint-rangep n2 32)))

; from the initial state to exit.  s --> sn, when str1 == NULL & last == NULL.
(prove-lemma strtok-s-sn (rewrite)
     (let ((sn (stepn s 14)))
       (implies (and (strtok-statep s str1 n1 lst1 str2 n2 lst2)
                     (equal (nat-to-uint str1) 0)
                     (equal (uread-mem (strtok-last-addr) (mc-mem s) 4) 0))
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-dn 32 0 sn) 0)
```

```
                    (equal (read-mem (strtok-last-addr) (mc-mem sn) 4)
                           (read-mem (strtok-last-addr) (mc-mem s) 4))
                    (equal (read-rn 32 15 (mc-rfile sn))
                           (add 32 (read-an 32 7 s) 4))
                    (equal (read-rn 32 14 (mc-rfile sn))
                           (read-an 32 6 s))))))

(prove-lemma strtok-s-sn-rfile (rewrite)
     (implies (and (strtok-statep s str1 n1 lst1 str2 n2 lst2)
                   (equal (nat-to-uint str1) 0)
                   (equal (uread-mem (strtok-last-addr) (mc-mem s) 4) 0)
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 14)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strtok-s-sn-mem (rewrite)
     (implies (and (strtok-statep s str1 n1 lst1 str2 n2 lst2)
                   (equal (nat-to-uint str1) 0)
                   (equal (uread-mem (strtok-last-addr) (mc-mem s) 4) 0)
                   (disjoint x k (sub 32 12 (read-sp s)) 24))
              (equal (read-mem x (mc-mem (stepn s 14)) k)
                     (read-mem x (mc-mem s) k))))

; from the initial state to s0.  s --> s0, when str1 =~ NULL.
(prove-lemma strtok-s-s0-1 ()
     (implies (and (strtok-statep s str1 n1 lst1 str2 n2 lst2)
                   (not (equal (nat-to-uint str1) 0)))
              (strtok-s0p (stepn s 6) 0 0 str1 n1 lst1 str2 n2 lst2)))

(prove-lemma strtok-s-s0-else-1 (rewrite)
     (let ((s0 (stepn s 6)))
       (implies (and (strtok-statep s str1 n1 lst1 str2 n2 lst2)
                     (not (equal (nat-to-uint str1) 0)))
                (and (equal (linked-rts-addr s0) (rts-addr s))
                     (equal (linked-a6 s0) (read-an 32 6 s))
                     (equal (read-rn 32 14 (mc-rfile s0))
                            (sub 32 4 (read-sp s)))
                     (equal (movem-saved s0 4 8 2)
                            (readm-rn 32 '(2 3) (mc-rfile s)))))))

(prove-lemma strtok-s-s0-rfile-1 (rewrite)
     (implies (and (strtok-statep s str1 n1 lst1 str2 n2 lst2)
                   (not (equal (nat-to-uint str1) 0))
                   (d4-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 6)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strtok-s-s0-mem-1 (rewrite)
     (implies (and (strtok-statep s str1 n1 lst1 str2 n2 lst2)
                   (not (equal (nat-to-uint str1) 0))
                   (disjoint x k (sub 32 12 (read-sp s)) 24))
              (equal (read-mem x (mc-mem (stepn s 6)) k)
                     (read-mem x (mc-mem s) k))))
```

```
; from s to s0.  s -> s0, when str1 == NULL and last =~ NULL.
(prove-lemma strtok-s-s0-2 ()
     (let ((last (read-mem (strtok-last-addr) (mc-mem s) 4)))
       (implies (and (strtok-statep s str1 n1 lst1 str2 n2 lst2)
                     (equal (nat-to-uint str1) 0)
                     (not (equal (nat-to-uint last) 0)))
                (strtok-s0p (stepn s 9) 0 0 last n1 lst1 str2 n2 lst2))))

(prove-lemma strtok-s-s0-else-2 (rewrite)
     (let ((s0 (stepn s 9))
           (last (read-mem (strtok-last-addr) (mc-mem s) 4)))
       (implies (and (strtok-statep s str1 n1 lst1 str2 n2 lst2)
                     (equal (nat-to-uint str1) 0)
                     (not (equal (nat-to-uint last) 0)))
                (and (equal (linked-rts-addr s0) (rts-addr s))
                     (equal (linked-a6 s0) (read-an 32 6 s))
                     (equal (read-rn 32 14 (mc-rfile s0))
                            (sub 32 4 (read-sp s)))
                     (equal (movem-saved s0 4 8 2)
                            (readm-rn 32 '(2 3) (mc-rfile s)))))))

(prove-lemma strtok-s-s0-rfile-2 (rewrite)
     (implies (and (strtok-statep s str1 n1 lst1 str2 n2 lst2)
                   (equal (nat-to-uint str1) 0)
                   (not (equal (uread-mem (strtok-last-addr) (mc-mem s) 4) 0))
                   (d4-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 9)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strtok-s-s0-mem-2 (rewrite)
     (implies (and (strtok-statep s str1 n1 lst1 str2 n2 lst2)
                   (equal (nat-to-uint str1) 0)
                   (not (equal (uread-mem (strtok-last-addr) (mc-mem s) 4) 0))
                   (disjoint x k (sub 32 12 (read-sp s)) 24))
              (equal (read-mem x (mc-mem (stepn s 9)) k)
                     (read-mem x (mc-mem s) k))))

; from s0 to s1.  s0 --> s1.
; s0 --> s1.
(prove-lemma strtok-s0-s1 ()
     (implies (strtok-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
              (strtok-s1p (stepn s 4) i1* i1 str1 n1 lst1
                          0 0 str2 n2 lst2 (get-nth i1 lst1)))))

(prove-lemma strtok-s0-s1-else (rewrite)
     (let ((s1 (stepn s 4)))
       (implies (strtok-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                (and (equal (linked-rts-addr s1) (linked-rts-addr s))
                     (equal (linked-a6 s1) (linked-a6 s))
                     (equal (read-rn oplen 14 (mc-rfile s1))
                            (read-rn oplen 14 (mc-rfile s)))
                     (equal (movem-saved s1 4 16 4)
                            (movem-saved s 4 16 4))
```

```
                    (equal (read-mem x (mc-mem s1) k)
                           (read-mem x (mc-mem s) k))))))

(prove-lemma strtok-s0-s1-rfile (rewrite)
     (implies (and (strtok-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                   (d4-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 4)))
                     (read-rn oplen rn (mc-rfile s)))))

; loop: s1 --> s1.
; base case: s1 --> s2, when lst2[i] == 0.
(prove-lemma strtok-s1-s2-base (rewrite)
     (implies (and (strtok-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                   (equal (get-nth i2 lst2) 0))
              (and (strtok-s2p (stepn s 3) i1* i1 str1 n1 lst1 str2 n2 lst2 ch)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 3)))
                          (read-an 32 6 s))
                   (equal (linked-a6 (stepn s 3)) (linked-a6 s))
                   (equal (linked-rts-addr (stepn s 3))
                          (linked-rts-addr s))
                   (equal (read-mem x (mc-mem (stepn s 3)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma strtok-s1-s2-rfile-base (rewrite)
     (implies (and (strtok-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                   (equal (get-nth i2 lst2) 0)
                   (d4-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 3)))
                     (read-rn oplen rn (mc-rfile s)))))

; base case: s1 --> s0.
(prove-lemma strtok-s1-s0-base (rewrite)
     (implies (and (strtok-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                   (equal ch (get-nth i1 lst1))
                   (not (equal (get-nth i2 lst2) 0))
                   (equal (get-nth i2 lst2) ch))
              (and (strtok-s0p (stepn s 5) (add 32 i1* 1) (add1 i1) str1 n1
                               lst1 str2 n2 lst2)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 5)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (linked-a6 (stepn s 5)) (linked-a6 s))
                   (equal (linked-rts-addr (stepn s 5))
                          (linked-rts-addr s))
                   (equal (read-mem x (mc-mem (stepn s 5)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma strtok-s1-s0-rfile-base (rewrite)
     (implies (and (strtok-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                   (not (equal (get-nth i2 lst2) 0))
                   (equal (get-nth i2 lst2) ch)
                   (d4-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 5)))
                     (read-rn oplen rn (mc-rfile s)))))
```

```
; induction case: s1 --> s1.
(prove-lemma strtok-s1-s1 (rewrite)
     (implies (and (strtok-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                   (not (equal (get-nth i2 lst2) 0))
                   (not (equal (get-nth i2 lst2) ch)))
              (and (strtok-s1p (stepn s 5) i1* i1 str1 n1 lst1 (add 32 i2* 1)
                               (add1 i2) str2 n2 lst2 ch)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 5)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (linked-a6 (stepn s 5)) (linked-a6 s))
                   (equal (linked-rts-addr (stepn s 5))
                          (linked-rts-addr s))
                   (equal (rn-saved (stepn s 5)) (rn-saved s))
                   (equal (read-mem x (mc-mem (stepn s 5)) k)
                          (read-mem x (mc-mem s) k)))))

(prove-lemma strtok-s1-s1-rfile (rewrite)
     (implies (and (strtok-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                   (not (equal (get-nth i2 lst2) 0))
                   (not (equal (get-nth i2 lst2) ch))
                   (d4-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 5)))
                     (read-rn oplen rn (mc-rfile s)))))

; put together:  s1 --> s0, when (strchr1 i2 n2 lst2 ch).
(prove-lemma strtok-s1p-info (rewrite)
     (implies (strtok-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
              (equal (lessp i2 n2) t)))

(prove-lemma strtok-s1-s0 (rewrite)
     (let ((s0 (stepn s (strtok-t0 i2 n2 lst2 ch))))
       (implies (and (strtok-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                     (equal ch (get-nth i1 lst1))
                     (strchr1 i2 n2 lst2 ch))
                (and (strtok-s0p s0 (add 32 i1* 1) (add1 i1) str1 n1 lst1
                                 str2 n2 lst2)
                     (equal (read-rn 32 14 (mc-rfile s0)) (read-an 32 6 s))
                     (equal (linked-a6 s0) (linked-a6 s))
                     (equal (linked-rts-addr s0) (linked-rts-addr s))
                     (equal (movem-saved s0 4 8 2) (movem-saved s 4 8 2))
                     (equal (read-mem x (mc-mem s0) k)
                            (read-mem x (mc-mem s) k)))))
     ((induct (strtok-induct0 s i2* i2 n2 lst2 ch))
      (disable strtok-s0p strtok-s1p)))

(prove-lemma strtok-s1-s0-rfile (rewrite)
     (implies
      (and (strtok-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
           (strchr1 i2 n2 lst2 ch)
           (d4-7a2-5p rn))
      (equal (read-rn oplen rn (mc-rfile (stepn s (strtok-t0 i2 n2 lst2 ch))))
             (read-rn oplen rn (mc-rfile s))))
     ((induct (strtok-induct0 s i2* i2 n2 lst2 ch))
      (disable strtok-s1p)))
```

```
; put together:  s1 --> s2, when (strchr1 i2 n2 lst2 ch).
(prove-lemma strtok-s1-s2 (rewrite)
     (let ((s2 (stepn s (strtok-t0 i2 n2 lst2 ch))))
       (implies (and (strtok-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
                     (not (strchr1 i2 n2 lst2 ch)))
                (and (strtok-s2p s2 i1* i1 str1 n1 lst1 str2 n2 lst2 ch)
                     (equal (read-rn 32 14 (mc-rfile s2)) (read-an 32 6 s))
                     (equal (linked-a6 s2) (linked-a6 s))
                     (equal (linked-rts-addr s2) (linked-rts-addr s))
                     (equal (movem-saved s2 4 8 2) (movem-saved s 4 8 2))
                     (equal (read-mem x (mc-mem s2) k)
                            (read-mem x (mc-mem s) k))))))
     ((induct (strtok-induct0 s i2* i2 n2 lst2 ch))
      (disable strtok-s2p strtok-s1p)))

(prove-lemma strtok-s1-s2-rfile (rewrite)
     (implies
      (and (strtok-s1p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch)
           (not (strchr1 i2 n2 lst2 ch))
           (d4-7a2-5p rn))
      (equal (read-rn oplen rn (mc-rfile (stepn s (strtok-t0 i2 n2 lst2 ch))))
             (read-rn oplen rn (mc-rfile s))))
     ((induct (strtok-induct0 s i2* i2 n2 lst2 ch))
      (disable strtok-s1p)))

(disable strtok-s1p-info)

; from s0 to s2:  s0 --> s2.
; base case: s0 --> s2.
(prove-lemma strtok-s0-s2-base (rewrite)
     (let ((s2 (stepn s (strtok-t1 n2 lst2 (get-nth i1 lst1)))))
       (implies (and (strtok-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                     (not (strchr1 0 n2 lst2 (get-nth i1 lst1))))
                (and (strtok-s2p s2 i1* i1 str1 n1 lst1 str2 n2 lst2
                                 (get-nth i1 lst1))
                     (equal (linked-rts-addr s2) (linked-rts-addr s))
                     (equal (linked-a6 s2) (linked-a6 s))
                     (equal (read-rn 32 14 (mc-rfile s2))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (movem-saved s2 4 8 2) (movem-saved s 4 8 2))
                     (equal (read-mem x (mc-mem s2) k)
                            (read-mem x (mc-mem s) k))))))
     ((use (strtok-s0-s1))
      (disable strtok-s0p strtok-s1p strtok-s2p)))

(prove-lemma strtok-s0-s2-rfile-base (rewrite)
     (let ((s2 (stepn s (strtok-t1 n2 lst2 (get-nth i1 lst1)))))
       (implies (and (strtok-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                     (not (strchr1 0 n2 lst2 (get-nth i1 lst1)))
                     (d4-7a2-5p rn))
                (equal (read-rn oplen rn (mc-rfile s2))
                       (read-rn oplen rn (mc-rfile s))))))
     ((use (strtok-s0-s1))
```

```
      (disable strtok-s0p strtok-s1p strtok-s2p strtok-t0)))

; induction case: s0 --> s0.
(prove-lemma strtok-s0-s0 (rewrite)
     (let ((s0 (stepn s (strtok-t1 n2 lst2 (get-nth i1 lst1)))))
       (implies (and (strtok-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                     (strchr1 0 n2 lst2 (get-nth i1 lst1)))
                (and (strtok-s0p s0 (add 32 i1* 1) (add1 i1) str1 n1 lst1
                                 str2 n2 lst2)
                     (equal (linked-rts-addr s0) (linked-rts-addr s))
                     (equal (linked-a6 s0) (linked-a6 s))
                     (equal (read-rn 32 14 (mc-rfile s0))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (movem-saved s0 4 8 2) (movem-saved s 4 8 2))
                     (equal (read-mem x (mc-mem s0) k)
                            (read-mem x (mc-mem s) k))))))
     ((use (strtok-s0-s1))
      (disable strtok-s0p strtok-s1p strtok-s2p strtok-t0 strchr1)))

(prove-lemma strtok-s0-s0-rfile (rewrite)
     (let ((s0 (stepn s (strtok-t1 n2 lst2 (get-nth i1 lst1)))))
       (implies (and (strtok-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                     (strchr1 0 n2 lst2 (get-nth i1 lst1))
                     (d4-7a2-5p rn))
                (equal (read-rn oplen rn (mc-rfile s0))
                       (read-rn oplen rn (mc-rfile s))))))
      ((use (strtok-s0-s1))
       (disable strtok-s0p strtok-s1p strtok-s2p strtok-t0 strchr1)))

; put together: s0 --> s2.
(prove-lemma strtok-s0p-info (rewrite)
     (implies (strtok-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
              (and (numberp i1)
                   (equal (lessp i1 n1) t))))

(prove-lemma strtok-s0-s2 ()
     (let ((s2 (stepn s (strtok-t2 i1 n1 lst1 n2 lst2))))
       (implies (strtok-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                (strtok-s2p s2 (strspn* i1* i1 n1 lst1 n2 lst2)
                            (strspn i1 n1 lst1 n2 lst2) str1 n1 lst1
                            str2 n2 lst2
                            (get-nth (strspn i1 n1 lst1 n2 lst2) lst1)))))
     ((induct (strtok-induct1 s i1* i1 n1 lst1 n2 lst2))
      (disable strtok-s0p strtok-s2p strtok-t1 strchr1)))

(disable strtok-s0p-info)

(prove-lemma strtok-s0-s2-else (rewrite)
     (let ((s2 (stepn s (strtok-t2 i1 n1 lst1 n2 lst2))))
       (implies (strtok-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                (and (equal (linked-rts-addr s2) (linked-rts-addr s))
                     (equal (linked-a6 s2) (linked-a6 s))
                     (equal (read-rn 32 14 (mc-rfile s2))
                            (read-rn 32 14 (mc-rfile s)))
```

```
                    (equal (movem-saved s2 4 8 2) (movem-saved s 4 8 2))
                    (equal (read-mem x (mc-mem s2) k)
                           (read-mem x (mc-mem s) k)))))
     ((induct (strtok-induct1 s i1* i1 n1 lst1 n2 lst2))
      (disable strtok-s0p strtok-s2p strtok-t1 strchr1)))

(prove-lemma strtok-s0-s2-rfile (rewrite)
     (let ((s2 (stepn s (strtok-t2 i1 n1 lst1 n2 lst2))))
       (implies (and (strtok-s0p s i1* i1 str1 n1 lst1 str2 n2 lst2)
                     (d4-7a2-5p rn))
                (equal (read-rn oplen rn (mc-rfile s2))
                       (read-rn oplen rn (mc-rfile s))))))
     ((induct (strtok-induct1 s i1* i1 n1 lst1 n2 lst2))
      (disable strtok-s0p strtok-s2p strtok-t1 strchr1)))

; from s2 to exit:  s2 --> sn.
(prove-lemma strtok-s2-sn-1 (rewrite)
     (let ((sn (stepn s 8)))
       (implies (and (strtok-s2p s i* i str1 n1 lst1 str2 n2 lst2 ch)
                     (equal ch 0))
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) 0)
                     (equal (read-mem (strtok-last-addr) (mc-mem sn) 4) 0)
                     (mem-lst 1 str1 (mc-mem sn) n1 lst1)
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))))))

(prove-lemma strtok-s2-sn-rfile-1 (rewrite)
     (implies (and (strtok-s2p s i* i str1 n1 lst1 str2 n2 lst2 ch)
                   (equal ch 0)
                   (d2-7a2-5p rn)
                   (leq oplen 32))
              (equal (read-rn oplen rn (mc-rfile (stepn s 8)))
                     (if (d4-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (get-vlst oplen 0 rn '(2 3)
                                 (movem-saved s 4 8 2))))))

(prove-lemma strtok-s2-sn-mem-1 (rewrite)
     (implies (and (strtok-s2p s i* i str1 n1 lst1 str2 n2 lst2 ch)
                   (equal ch 0)
                   (disjoint x k (strtok-last-addr) 4))
              (equal (read-mem x (mc-mem (stepn s 8)) k)
                     (read-mem x (mc-mem s) k))))

; from s2 to s3:  s2 --> s3.
(prove-lemma strtok-s2-s3 ()
     (implies (and (strtok-s2p s i1* i1 str1 n1 lst1 str2 n2 lst2 ch)
                   (equal ch (get-nth i1 lst1))
                   (not (equal ch 0)))
              (strtok-s3p (stepn s 4) (add 32 i1* 1) (add1 i1) str1 n1 lst1
```

```
                        str2 n2 lst2 (add 32 str1 i1*))))

(prove-lemma strtok-s2-s3-else (rewrite)
     (let ((s3 (stepn s 4)))
       (implies (and (strtok-s2p s i1* i1 str1 n1 lst1 str2 n2 lst2 ch)
                     (not (equal ch 0)))
                (and (equal (linked-rts-addr s3) (linked-rts-addr s))
                     (equal (linked-a6 s3) (linked-a6 s))
                     (equal (read-rn oplen 14 (mc-rfile s3))
                            (read-rn oplen 14 (mc-rfile s)))
                     (equal (movem-saved s3 4 8 2)
                            (movem-saved s 4 8 2))
                     (equal (read-mem x (mc-mem s3) k)
                            (read-mem x (mc-mem s) k))))))

(prove-lemma strtok-s2-s3-rfile (rewrite)
     (implies (and (strtok-s2p s i1* i1 str1 n1 lst1 str2 n2 lst2 ch)
                   (not (equal ch 0))
                   (d4-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 4)))
                     (read-rn oplen rn (mc-rfile s)))))

; from s3 to s4:  s3 --> s4.
(prove-lemma strtok-s3-s4 ()
     (implies (strtok-s3p s i1* i1 str1 n1 lst1 str2 n2 lst2 tok)
              (strtok-s4p (stepn s 3) i1* i1 str1 n1 lst1
                          0 0 str2 n2 lst2 (get-nth i1 lst1) tok)))

(prove-lemma strtok-s3-s4-else (rewrite)
     (let ((s4 (stepn s 3)))
       (implies (strtok-s3p s i1* i1 str1 n1 lst1 str2 n2 lst2 tok)
                (and (equal (linked-rts-addr s4) (linked-rts-addr s))
                     (equal (linked-a6 s4) (linked-a6 s))
                     (equal (read-rn oplen 14 (mc-rfile s4))
                            (read-rn oplen 14 (mc-rfile s)))
                     (equal (movem-saved s4 4 16 4)
                            (movem-saved s 4 16 4))
                     (equal (read-mem x (mc-mem s4) k)
                            (read-mem x (mc-mem s) k))))))

(prove-lemma strtok-s3-s4-rfile (rewrite)
     (implies (and (strtok-s3p s i1* i1 str1 n1 lst1 str2 n2 lst2 tok)
                   (d4-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 3)))
                     (read-rn oplen rn (mc-rfile s)))))

; from s4 to exit: s4 --> sn.
; case 1.
(prove-lemma strtok-s4-sn-1 (rewrite)
     (let ((sn (stepn s 13)))
       (implies
        (and (strtok-s4p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch tok)
             (equal (get-nth i2 lst2) ch)
             (not (equal ch 0)))
```

```
        (and (equal (mc-status sn) 'running)
             (equal (mc-pc sn) (linked-rts-addr s))
             (equal (read-dn 32 0 sn) tok)
             (equal (read-mem (strtok-last-addr) (mc-mem sn) 4)
                    (add 32 str1 (add 32 i1* 1)))
             (mem-lst 1 str1 (mc-mem sn) n1 (put-nth 0 i1 lst1))
             (equal (read-rn 32 14 (mc-rfile sn))
                    (linked-a6 s))
             (equal (read-rn 32 15 (mc-rfile sn))
                    (add 32 (read-an 32 6 s) 8))))))

(prove-lemma strtok-s4-sn-rfile-1 (rewrite)
     (implies
      (and (strtok-s4p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch tok)
           (equal (get-nth i2 lst2) ch)
           (not (equal ch 0))
           (d2-7a2-5p rn)
           (leq oplen 32))
      (equal (read-rn oplen rn (mc-rfile (stepn s 13)))
             (if (d4-7a2-5p rn)
                 (read-rn oplen rn (mc-rfile s))
               (get-vlst oplen 0 rn '(2 3) (movem-saved s 4 8 2))))))

(prove-lemma strtok-s4-sn-mem-1 (rewrite)
     (implies
      (and (strtok-s4p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch tok)
           (equal (get-nth i2 lst2) ch)
           (not (equal ch 0))
           (disjoint x k (strtok-last-addr) 4)
           (disjoint x k str1 n1))
      (equal (read-mem x (mc-mem (stepn s 13)) k)
             (read-mem x (mc-mem s) k))))

; case 2.
(prove-lemma strtok-s4-sn-2 (rewrite)
     (let ((sn (stepn s 14)))
       (implies
        (and (strtok-s4p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch tok)
             (equal (get-nth i2 lst2) ch)
             (equal ch 0))
        (and (equal (mc-status sn) 'running)
             (equal (mc-pc sn) (linked-rts-addr s))
             (equal (read-dn 32 0 sn) tok)
             (equal (read-mem (strtok-last-addr) (mc-mem sn) 4) 0)
             (mem-lst 1 str1 (mc-mem sn) n1 lst1)
             (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
             (equal (read-rn 32 15 (mc-rfile sn))
                    (add 32 (read-an 32 6 s) 8))))))

(prove-lemma strtok-s4-sn-rfile-2 (rewrite)
     (implies
      (and (strtok-s4p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch tok)
           (equal (get-nth i2 lst2) ch)
           (equal ch 0)
```

```
          (d2-7a2-5p rn)
          (leq oplen 32))
     (equal (read-rn oplen rn (mc-rfile (stepn s 14)))
            (if (d4-7a2-5p rn)
                (read-rn oplen rn (mc-rfile s))
              (get-vlst oplen 0 rn '(2 3) (movem-saved s 4 8 2))))))

(prove-lemma strtok-s4-sn-mem-2 (rewrite)
     (implies
      (and (strtok-s4p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch tok)
           (equal (get-nth i2 lst2) ch)
           (equal ch 0)
           (disjoint x k (strtok-last-addr) 4)
           (disjoint x k str1 n1))
      (equal (read-mem x (mc-mem (stepn s 14)) k)
             (read-mem x (mc-mem s) k))))

; from s4 to s3:  s4 --> s3.
(prove-lemma strtok-s4-s3-base (rewrite)
     (let ((s3 (stepn s 7)))
       (implies
        (and (strtok-s4p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch tok)
             (equal ch (get-nth i1 lst1))
             (not (equal (get-nth i2 lst2) ch))
             (equal (get-nth i2 lst2) 0))
        (and (strtok-s3p s3 (add 32 i1* 1) (add1 i1) str1 n1 lst1
                         str2 n2 lst2 tok)
             (equal (linked-rts-addr s3) (linked-rts-addr s))
             (equal (linked-a6 s3) (linked-a6 s))
             (equal (read-rn oplen 14 (mc-rfile s3))
                    (read-rn oplen 14 (mc-rfile s)))
             (equal (movem-saved s3 4 8 2) (movem-saved s 4 8 2))
             (equal (read-mem x (mc-mem s3) k)
                    (read-mem x (mc-mem s) k))))))

(prove-lemma strtok-s4-s3-rfile-base (rewrite)
     (implies
      (and (strtok-s4p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch tok)
           (not (equal (get-nth i2 lst2) ch))
           (equal (get-nth i2 lst2) 0)
           (d4-7a2-5p rn))
      (equal (read-rn oplen rn (mc-rfile (stepn s 7)))
             (read-rn oplen rn (mc-rfile s)))))

; from s4 to s4:  s4 --> s4.
(prove-lemma strtok-s4-s4 (rewrite)
     (let ((s4 (stepn s 6)))
       (implies
        (and (strtok-s4p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch tok)
             (not (equal (get-nth i2 lst2) ch))
             (not (equal (get-nth i2 lst2) 0)))
        (and (strtok-s4p s4 i1* i1 str1 n1 lst1 (add 32 i2* 1)
                         (add1 i2) str2 n2 lst2 ch tok)
             (equal (linked-rts-addr s4) (linked-rts-addr s))
```

```
            (equal (linked-a6 s4) (linked-a6 s))
            (equal (read-rn oplen 14 (mc-rfile s4))
                   (read-rn oplen 14 (mc-rfile s)))
            (equal (movem-saved s4 4 8 2) (movem-saved s 4 8 2))
            (equal (read-mem x (mc-mem s4) k)
                   (read-mem x (mc-mem s) k))))))

(prove-lemma strtok-s4-s4-rfile (rewrite)
     (implies
      (and (strtok-s4p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch tok)
           (not (equal (get-nth i2 lst2) ch))
           (not (equal (get-nth i2 lst2) 0))
           (d4-7a2-5p rn))
      (equal (read-rn oplen rn (mc-rfile (stepn s 6)))
             (read-rn oplen rn (mc-rfile s)))))

; put together: s4 --> sn.
(prove-lemma strtok-s4-sn (rewrite)
     (let ((sn (stepn s (strtok-t3 i2 n2 lst2 ch))))
       (implies
        (and (strtok-s4p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch tok)
             (strchr i2 n2 lst2 ch))
        (and (equal (mc-status sn) 'running)
             (equal (mc-pc sn) (linked-rts-addr s))
             (equal (read-dn 32 0 sn) tok)
             (equal (read-mem (strtok-last-addr) (mc-mem sn) 4)
                    (if (equal ch 0) 0 (add 32 str1 (add 32 i1* 1))))
             (mem-lst 1 str1 (mc-mem sn) n1 (strtok-lst0 i1 lst1 ch))
             (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
             (equal (read-rn 32 15 (mc-rfile sn))
                    (add 32 (read-an 32 6 s) 8))))))
     ((induct (strtok-induct2 s i2* i2 n2 lst2 ch))
      (disable strtok-s4p read-dn)))

(prove-lemma strtok-s4-sn-rfile (rewrite)
     (let ((sn (stepn s (strtok-t3 i2 n2 lst2 ch))))
       (implies
        (and (strtok-s4p s i* i str1 n1 lst1 i2* i2 str2 n2 lst2 ch tok)
             (strchr i2 n2 lst2 ch)
             (d2-7a2-5p rn)
             (leq oplen 32))
        (equal (read-rn oplen rn (mc-rfile sn))
               (if (d4-7a2-5p rn)
                   (read-rn oplen rn (mc-rfile s))
                 (get-vlst oplen 0 rn '(2 3) (movem-saved s 4 8 2)))))))
     ((induct (strtok-induct2 s i2* i2 n2 lst2 ch))
      (disable strtok-s4p)))

(prove-lemma strtok-s4-sn-mem (rewrite)
     (let ((sn (stepn s (strtok-t3 i2 n2 lst2 ch))))
       (implies
        (and (strtok-s4p s i* i str1 n1 lst1 i2* i2 str2 n2 lst2 ch tok)
             (strchr i2 n2 lst2 ch)
             (disjoint x k (strtok-last-addr) 4)
```

```
             (disjoint x k str1 n1))
        (equal (read-mem x (mc-mem sn) k)
               (read-mem x (mc-mem s) k))))
     ((induct (strtok-induct2 s i2* i2 n2 lst2 ch))
      (disable strtok-s4p)))

; put together: s4 --> s3.
(prove-lemma strtok-s4p-info (rewrite)
     (implies (strtok-s4p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch tok)
              (equal (lessp i2 n2) t)))

(prove-lemma strtok-s4-s3 (rewrite)
     (let ((s3 (stepn s (strtok-t3 i2 n2 lst2 ch))))
       (implies
        (and (strtok-s4p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch tok)
             (equal ch (get-nth i1 lst1))
             (not (strchr i2 n2 lst2 ch)))
        (and (strtok-s3p s3 (add 32 i1* 1) (add1 i1) str1 n1 lst1
                         str2 n2 lst2 tok)
             (equal (linked-rts-addr s3) (linked-rts-addr s))
             (equal (linked-a6 s3) (linked-a6 s))
             (equal (read-rn oplen 14 (mc-rfile s3))
                    (read-rn oplen 14 (mc-rfile s)))
             (equal (movem-saved s3 4 8 2) (movem-saved s 4 8 2))
             (equal (read-mem x (mc-mem s3) k)
                    (read-mem x (mc-mem s) k))))))
     ((induct (strtok-induct2 s i2* i2 n2 lst2 ch))
      (disable strtok-s4p strtok-s3p)))

(disable strtok-s4p-info)

(prove-lemma strtok-s4-s3-rfile (rewrite)
     (let ((s3 (stepn s (strtok-t3 i2 n2 lst2 ch))))
       (implies
        (and (strtok-s4p s i1* i1 str1 n1 lst1 i2* i2 str2 n2 lst2 ch tok)
             (not (strchr i2 n2 lst2 ch))
             (d4-7a2-5p rn))
        (equal (read-rn oplen rn (mc-rfile s3))
               (read-rn oplen rn (mc-rfile s))))))
     ((induct (strtok-induct2 s i2* i2 n2 lst2 ch))
      (disable strtok-s4p)))

; from s3 to exit:  s3 --> sn.
(prove-lemma strtok-s3-sn-base (rewrite)
     (let ((ch (get-nth i1 lst1)))
       (let ((sn (stepn s (strtok-t4 n2 lst2 ch))))
         (implies (and (strtok-s3p s i1* i1 str1 n1 lst1 str2 n2 lst2 tok)
                       (strchr 0 n2 lst2 ch))
                  (and (equal (mc-status sn) 'running)
                       (equal (mc-pc sn) (linked-rts-addr s))
                       (equal (read-dn 32 0 sn) tok)
                       (equal (read-mem (strtok-last-addr) (mc-mem sn) 4)
                              (if (equal ch 0) 0 (add 32 str1 (add 32 i1* 1))))
                       (mem-lst 1 str1 (mc-mem sn) n1 (strtok-lst0 i1 lst1 ch))
```

```
                    (equal (read-rn 32 14 (mc-rfile sn))
                           (linked-a6 s))
                    (equal (read-rn 32 15 (mc-rfile sn))
                           (add 32 (read-an 32 6 s) 8))))))
     ((use (strtok-s3-s4))
      (disable strtok-s3p strtok-s4p read-dn strtok-t3 strtok-lst0)))

(prove-lemma strtok-s3-sn-rfile-base (rewrite)
     (let ((sn (stepn s (strtok-t4 n2 lst2 (get-nth i1 lst1)))))
       (implies (and (strtok-s3p s i1* i1 str1 n1 lst1 str2 n2 lst2 tok)
                     (strchr 0 n2 lst2 (get-nth i1 lst1))
                     (d2-7a2-5p rn)
                     (leq oplen 32))
                (equal (read-rn oplen rn (mc-rfile sn))
                       (if (d4-7a2-5p rn)
                           (read-rn oplen rn (mc-rfile s))
                         (get-vlst oplen 0 rn '(2 3) (movem-saved s 4 8 2)))))))
     ((use (strtok-s3-s4))
      (disable strtok-s3p strtok-s4p strchr strtok-t3)))

(prove-lemma strtok-s3-sn-mem-base (rewrite)
     (let ((sn (stepn s (strtok-t4 n2 lst2 (get-nth i1 lst1)))))
       (implies (and (strtok-s3p s i1* i1 str1 n1 lst1 str2 n2 lst2 tok)
                     (strchr 0 n2 lst2 (get-nth i1 lst1))
                     (disjoint x k (strtok-last-addr) 4)
                     (disjoint x k str1 n1))
                (equal (read-mem x (mc-mem sn) k)
                       (read-mem x (mc-mem s) k)))))
     ((use (strtok-s3-s4))
      (disable strtok-s3p strtok-s4p)))

; from s3 to s3:  s3 --> s3.
(prove-lemma strtok-s3-s3 (rewrite)
     (let ((s3 (stepn s (strtok-t4 n2 lst2 (get-nth i1 lst1)))))
       (implies (and (strtok-s3p s i1* i1 str1 n1 lst1 str2 n2 lst2 tok)
                     (not (strchr 0 n2 lst2 (get-nth i1 lst1))))
                (and (strtok-s3p s3 (add 32 i1* 1) (add1 i1) str1 n1 lst1
                                 str2 n2 lst2 tok)
                     (equal (linked-rts-addr s3) (linked-rts-addr s))
                     (equal (linked-a6 s3) (linked-a6 s))
                     (equal (read-rn oplen 14 (mc-rfile s3))
                            (read-rn oplen 14 (mc-rfile s)))
                     (equal (movem-saved s3 4 16 4)
                            (movem-saved s 4 16 4))
                     (equal (read-mem x (mc-mem s3) k)
                            (read-mem x (mc-mem s) k)))))
     ((use (strtok-s3-s4))
      (disable strtok-s3p strtok-s4p strchr strtok-t3)))

(prove-lemma strtok-s3-s3-rfile (rewrite)
     (let ((s3 (stepn s (strtok-t4 n2 lst2 (get-nth i1 lst1)))))
       (implies (and (strtok-s3p s i1* i1 str1 n1 lst1 str2 n2 lst2 tok)
                     (not (strchr 0 n2 lst2 (get-nth i1 lst1)))
                     (d4-7a2-5p rn))
```

```
                (equal (read-rn oplen rn (mc-rfile s3))
                       (read-rn oplen rn (mc-rfile s)))))
     ((use (strtok-s3-s4))
      (disable strtok-s3p strtok-s4p strchr strtok-t3)))

; put together: s3 --> sn.
(prove-lemma strtok-s3p-info (rewrite)
     (implies (strtok-s3p s i1* i1 str1 n1 lst1 str2 n2 lst2 tok)
              (and (numberp i1)
                   (equal (lessp i1 n1) t))))

(prove-lemma strtok-s3-sn (rewrite)
     (let ((sn (stepn s (strtok-t5 i1 n1 lst1 n2 lst2))))
       (implies (strtok-s3p s i1* i1 str1 n1 lst1 str2 n2 lst2 tok)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn) tok)
                     (equal (read-mem (strtok-last-addr) (mc-mem sn) 4)
                            (strtok-last0 str1 i1* i1 n1 lst1 n2 lst2))
                     (mem-lst 1 str1 (mc-mem sn) n1
                              (strtok-lst1 i1 n1 lst1 n2 lst2))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))))))
     ((induct (strtok-induct3 s i1* i1 n1 lst1 n2 lst2))
      (disable strtok-s3p strtok-t4 strchr read-dn strtok-lst0)))

(prove-lemma strtok-s3-sn-rfile (rewrite)
     (let ((sn (stepn s (strtok-t5 i1 n1 lst1 n2 lst2))))
       (implies (and (strtok-s3p s i1* i1 str1 n1 lst1 str2 n2 lst2 tok)
                     (d2-7a2-5p rn)
                     (leq oplen 32))
                (equal (read-rn oplen rn (mc-rfile sn))
                       (if (d4-7a2-5p rn)
                           (read-rn oplen rn (mc-rfile s))
                         (get-vlst oplen 0 rn '(2 3) (movem-saved s 4 8 2)))))))
     ((induct (strtok-induct3 s i1* i1 n1 lst1 n2 lst2))
      (disable strtok-s3p strtok-t4 strchr)))

(prove-lemma strtok-s3-sn-mem (rewrite)
     (let ((sn (stepn s (strtok-t5 i1 n1 lst1 n2 lst2))))
       (implies (and (strtok-s3p s i1* i1 str1 n1 lst1 str2 n2 lst2 tok)
                     (disjoint x k (strtok-last-addr) 4)
                     (disjoint x k str1 n1))
                (equal (read-mem x (mc-mem sn) k)
                       (read-mem x (mc-mem s) k)))))
     ((induct (strtok-induct3 s i1* i1 n1 lst1 n2 lst2))
      (disable strtok-s3p strtok-t4 strchr)))

(disable strtok-s3p-info)

; from s2 to exit:  s2 --> sn.
(prove-lemma strtok-s2-sn (rewrite)
```

```
     (let ((sn (stepn s (strtok-t6 i1 n1 lst1 n2 lst2))))
       (implies (and (strtok-s2p s i1* i1 str1 n1 lst1 str2 n2 lst2 ch)
                     (equal (get-nth i1 lst1) ch))
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (equal (read-dn 32 0 sn)
                            (if (equal ch 0) 0 (add 32 str1 i1*)))
                     (equal (read-mem (strtok-last-addr) (mc-mem sn) 4)
                            (strtok-last1 str1 i1* i1 n1 lst1 n2 lst2))
                     (mem-lst 1 str1 (mc-mem sn) n1
                              (strtok-lst2 i1 n1 lst1 n2 lst2))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8)))))
     ((use (strtok-s2-s3 (ch (get-nth i1 lst1))))
      (disable strtok-s2p strtok-s3p strtok-t5 read-dn strtok-last0
               strtok-lst1)))

(prove-lemma strtok-s2-sn-rfile (rewrite)
     (let ((ch (get-nth i1 lst1))
           (sn (stepn s (strtok-t6 i1 n1 lst1 n2 lst2))))
       (implies (and (strtok-s2p s i1* i1 str1 n1 lst1 str2 n2 lst2 ch)
                     (d2-7a2-5p rn)
                     (leq oplen 32))
                (equal (read-rn oplen rn (mc-rfile sn))
                       (if (d4-7a2-5p rn)
                           (read-rn oplen rn (mc-rfile s))
                         (get-vlst oplen 0 rn '(2 3) (movem-saved s 4 8 2)))))))
     ((use (strtok-s2-s3 (ch (get-nth i1 lst1))))
      (disable strtok-s2p strtok-s3p strtok-t5)))

(prove-lemma strtok-s2-sn-mem (rewrite)
     (let ((ch (get-nth i1 lst1))
           (sn (stepn s (strtok-t6 i1 n1 lst1 n2 lst2))))
       (implies (and (strtok-s2p s i1* i1 str1 n1 lst1 str2 n2 lst2 ch)
                     (disjoint x k (strtok-last-addr) 4)
                     (disjoint x k str1 n1))
                (equal (read-mem x (mc-mem sn) k)
                       (read-mem x (mc-mem s) k))))
     ((use (strtok-s2-s3 (ch (get-nth i1 lst1))))
      (disable strtok-s2p strtok-s3p)))

; the correctness of strtok.
(prove-lemma strtok-correctness (rewrite)
     (let ((last (read-mem (strtok-last-addr) (mc-mem s) 4)))
       (let ((sn (stepn s (strtok-t str1 last n1 lst1 n2 lst2))))
         (implies (strtok-statep s str1 n1 lst1 str2 n2 lst2)
                  (and (equal (mc-status sn) 'running)
                       (equal (mc-pc sn) (rts-addr s))
                       (equal (read-dn 32 0 sn)
                              (strtok-tok str1 last n1 lst1 n2 lst2))
                       (equal (read-mem (strtok-last-addr) (mc-mem sn) 4)
                              (strtok-last str1 last n1 lst1 n2 lst2))
```

```
                    (equal (read-rn 32 14 (mc-rfile sn))
                           (read-rn 32 14 (mc-rfile s)))
                    (equal (read-rn 32 15 (mc-rfile sn))
                           (add 32 (read-sp s) 4))))))
     ((use (strtok-s-s0-1)
           (strtok-s-s0-2)
           (strtok-s0-s2 (s (stepn s 6)) (i1* 0) (i1 0))
           (strtok-s0-s2 (s (stepn s 9))
                         (str1 (read-mem (strtok-last-addr) (mc-mem s) 4))
                         (i1* 0) (i1 0)))
     (disable strtok-statep strtok-s0p strtok-s2p strspn* strspn strtok-t2
              strtok-t6 linked-rts-addr linked-a6 read-dn strtok-last1)))

(prove-lemma strtok-lst-1 (rewrite)
     (let ((last (read-mem (strtok-last-addr) (mc-mem s) 4)))
       (let ((sn (stepn s (strtok-t str1 last n1 lst1 n2 lst2))))
         (implies (and (strtok-statep s str1 n1 lst1 str2 n2 lst2)
                       (not (equal (nat-to-uint str1) 0)))
                  (mem-lst 1 str1 (mc-mem sn) n1
                           (strtok-lst n1 lst1 n2 lst2))))))
     ((use (strtok-s-s0-1)
           (strtok-s0-s2 (s (stepn s 6)) (i1* 0) (i1 0))
           (strtok-s0-s2 (s (stepn s 9))
                         (str1 (read-mem (strtok-last-addr) (mc-mem s) 4))
                         (i1* 0) (i1 0)))
      (disable strtok-statep strtok-s0p strtok-s2p strspn* strspn
               strtok-t2 strtok-t6 strtok-lst2)))

(prove-lemma strtok-lst-2 (rewrite)
     (let ((last (read-mem (strtok-last-addr) (mc-mem s) 4)))
       (let ((sn (stepn s (strtok-t str1 last n1 lst1 n2 lst2))))
         (implies (and (strtok-statep s str1 n1 lst1 str2 n2 lst2)
                       (equal (nat-to-uint str1) 0)
                       (not (equal (nat-to-uint last) 0)))
                  (mem-lst 1 last (mc-mem sn) n1
                           (strtok-lst n1 lst1 n2 lst2))))))
     ((use (strtok-s-s0-2)
           (strtok-s0-s2 (s (stepn s 6)) (i1* 0) (i1 0))
           (strtok-s0-s2 (s (stepn s 9))
                         (str1 (read-mem (strtok-last-addr) (mc-mem s) 4))
                         (i1* 0) (i1 0)))
      (disable strtok-statep strtok-s0p strtok-s2p strspn* strspn
               strtok-t2 strtok-t6 strtok-lst2)))

(prove-lemma strtok-rfile (rewrite)
     (let ((last (read-mem (strtok-last-addr) (mc-mem s) 4)))
       (let ((sn (stepn s (strtok-t str1 last n1 lst1 n2 lst2))))
         (implies (and (strtok-statep s str1 n1 lst1 str2 n2 lst2)
                       (d2-7a2-5p rn)
                       (leq oplen 32))
                  (equal (read-rn oplen rn (mc-rfile sn))
                         (read-rn oplen rn (mc-rfile s)))))))
     ((use (strtok-s-s0-1)
           (strtok-s-s0-2)
```

```
          (strtok-s0-s2 (s (stepn s 6)) (i1* 0) (i1 0))
          (strtok-s0-s2 (s (stepn s 9))
                        (str1 (read-mem (strtok-last-addr) (mc-mem s) 4))
                        (i1* 0) (i1 0)))
     (disable strtok-statep strtok-s0p strtok-s2p strspn* strspn
              strtok-t2 strtok-t6)))

(prove-lemma strtok-mem (rewrite)
     (let ((last (read-mem (strtok-last-addr) (mc-mem s) 4)))
       (let ((sn (stepn s (strtok-t str1 last n1 lst1 n2 lst2))))
         (implies (and (strtok-statep s str1 n1 lst1 str2 n2 lst2)
                       (disjoint x k (sub 32 12 (read-sp s)) 24)
                       (disjoint x k (strtok-last-addr) 4)
                       (if (equal (nat-to-uint str1) 0)
                           (disjoint x k last n1)
                         (disjoint x k str1 n1)))
                  (equal (read-mem x (mc-mem sn) k)
                         (read-mem x (mc-mem s) k))))))
     ((use (strtok-s-s0-1)
           (strtok-s-s0-2)
           (strtok-s0-s2 (s (stepn s 6)) (i1* 0) (i1 0))
           (strtok-s0-s2 (s (stepn s 9))
                         (str1 (read-mem (strtok-last-addr) (mc-mem s) 4))
                         (i1* 0) (i1 0)))
      (disable strtok-statep strtok-s0p strtok-s2p strspn* strspn
               strtok-t2 strtok-t6)))

(disable strtok-t)

; some properties of strtok.
; see the file cstring.events.
```

C.30 The strxfrm Function

```
;          Proof of the Correctness of the STRXFRM Function
#|
This is part of our effort to verify the Berkeley string library.  The
Berkeley string library is widely used as part of the Berkeley Unix OS.

This is the source code of strxfrm function in the Berkeley string library.

/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 */
/*
 * Transform src, storing the result in dst, such that
 * strcmp() on transformed strings returns what strcoll()
 * on the original untransformed strings would return.
 */
size_t
strxfrm(dst, src, n)
```

```
        register char *dst;
        register const char *src;
        register size_t n;
{
        register size_t r = 0;
        register int c;

        /*
         * Since locales are unimplemented, this is just a copy.
         */
        if (n != 0) {
                while ((c = *src++) != 0) {
                        r++;
                        if (--n == 0) {
                                while (*src++ != 0)
                                        r++;
                                break;
                        }
                        *dst++ = c;
                }
                *dst = 0;
        }
        return (r);
}
```

The MC68020 assembly code of the C function strxfrm on SUN-3 is given as follows. This binary is generated by "gcc -O".

```
0x23a0 <strxfrm>:       linkw fp,#0
0x23a4 <strxfrm+4>:     movel d2,sp@-
0x23a6 <strxfrm+6>:     moveal fp@(8),a1
0x23aa <strxfrm+10>:    moveal fp@(12),a0
0x23ae <strxfrm+14>:    movel fp@(16),d0
0x23b2 <strxfrm+18>:    clrl d1
0x23b4 <strxfrm+20>:    tstl d0
0x23b6 <strxfrm+22>:    beq 0x23d4 <strxfrm+52>
0x23b8 <strxfrm+24>:    bra 0x23cc <strxfrm+44>
0x23ba <strxfrm+26>:    addql #1,d1
0x23bc <strxfrm+28>:    subl #1,d0
0x23be <strxfrm+30>:    bne 0x23ca <strxfrm+42>
0x23c0 <strxfrm+32>:    bra 0x23c4 <strxfrm+36>
0x23c2 <strxfrm+34>:    addql #1,d1
0x23c4 <strxfrm+36>:    tstb a0@+
0x23c6 <strxfrm+38>:    bne 0x23c2 <strxfrm+34>
0x23c8 <strxfrm+40>:    bra 0x23d2 <strxfrm+50>
0x23ca <strxfrm+42>:    moveb d2,a1@+
0x23cc <strxfrm+44>:    moveb a0@+,d2
0x23ce <strxfrm+46>:    extbl d2
0x23d0 <strxfrm+48>:    bne 0x23ba <strxfrm+26>
0x23d2 <strxfrm+50>:    clrb a1@
0x23d4 <strxfrm+52>:    movel d1,d0
0x23d6 <strxfrm+54>:    movel fp@(-4),d2
0x23da <strxfrm+58>:    unlk fp
0x23dc <strxfrm+60>:    rts
```

```
The machine code of the above program is:

<strxfrm>:     0x4e56  0x0000  0x2f02  0x226e  0x0008  0x206e  0x000c  0x202e
<strxfrm+16>:  0x0010  0x4281  0x4a80  0x671c  0x6012  0x5281  0x5380  0x660a
<strxfrm+32>:  0x6002  0x5281  0x4a18  0x66fa  0x6008  0x12c2  0x1418  0x49c2
<strxfrm+48>:  0x66e8  0x4211  0x2001  0x242e  0xfffc  0x4e5e  0x4e75

'(78      86      0       0       47      2       34      110
  0       8       32      110     0       12      32      46
  0       16      66      129     74      128     103     28
  96      18      82      129     83      128     102     10
  96      2       82      129     74      24      102     250
  96      8       18      194     20      24      73      194
  102     232     66      17      32      1       36      46
  255     252     78      94      78      117)
|#

; in the logic, the above program is defined by (strxfrm-code).
(defn strxfrm-code ()
  '(78      86      0       0       47      2       34      110
    0       8       32      110     0       12      32      46
    0       16      66      129     74      128     103     28
    96      18      82      129     83      128     102     10
    96      2       82      129     74      24      102     250
    96      8       18      194     20      24      73      194
    102     232     66      17      32      1       36      46
    255     252     78      94      78      117))

; the Berkeley strxfrm returns the following value.  It seems a bug!
(defn strxfrm-n (n2 lst2 n)
  (if (zerop n)
      0
    (strlen 0 n2 lst2)))

; the computation time of the program.
(defn strxfrm-t2 (j n2 lst2)
  (if (lessp j n2)
      (if (equal (get-nth j lst2) 0)
          8
        (splus 3 (strxfrm-t2 (add1 j) n2 lst2)))
    0)
  ((lessp (difference n2 j))))

(defn strxfrm-t1 (i n2 lst2)
  (splus 7 (strxfrm-t2 (add1 i) n2 lst2)))

(defn strxfrm-t0 (i n2 lst2 n)
  (if (equal (get-nth i lst2) 0)
      8
    (if (equal (sub1 n) 0)
        (strxfrm-t1 i n2 lst2)
      (splus 7 (strxfrm-t0 (add1 i) n2 lst2 (sub1 n))))))
```

```
(defn strxfrm-t (n2 lst2 n)
  (if (zerop n)
      12
    (splus 9 (strxfrm-t0 0 n2 lst2 n))))

; two induction hints.
(defn strxfrm-induct2 (s j* j n2 lst2)
  (if (lessp j n2)
      (if (equal (get-nth j lst2) 0)
          t
        (strxfrm-induct2 (stepn s 3) (add 32 j* 1) (add1 j) n2 lst2))
    t)
  ((lessp (difference n2 j))))

(defn strxfrm-induct1 (s i* i lst1 lst2 n)
  (if (equal (get-nth i lst2) 0)
      t
    (if (equal (sub1 n) 0)
        t
      (strxfrm-induct1 (stepn s 7) (add 32 i* 1) (add1 i)
                       (put-nth (get-nth i lst2) i lst1) lst2 (sub1 n)))))

; the preconditions of the initial state.
(defn strxfrm-statep (s str1 n1 lst1 str2 n2 lst2 n)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (mc-pc s) (mc-mem s) 62)
       (mcode-addrp (mc-pc s) (mc-mem s) (strxfrm-code))
       (ram-addrp (sub 32 8 (read-sp s)) (mc-mem s) 24)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (sub 32 8 (read-sp s)) 24 str1 n1)
       (disjoint (sub 32 8 (read-sp s)) 24 str2 n2)
       (disjoint str1 n1 str2 n2)
       (equal str1 (read-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal str2 (read-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
       (equal n (uread-mem (add 32 (read-sp s) 12) (mc-mem s) 4))
       (lessp (slen 0 n2 lst2) n2)
       (leq n2 n1)
       (numberp n1)
       (numberp n2)
       (uint-rangep n1 32)
       (uint-rangep n2 32)))

; an intermediate state s0.
(defn strxfrm-s0p (s i* i str1 n1 lst1 str2 n2 lst2 n)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 44 (mc-pc s)) (mc-mem s) 62)
       (mcode-addrp (sub 32 44 (mc-pc s)) (mc-mem s) (strxfrm-code))
       (ram-addrp (sub 32 4 (read-an 32 6 s)) (mc-mem s) 24)
       (ram-addrp str1 (mc-mem s) n1)
```

```
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (sub 32 4 (read-an 32 6 s)) 24 str1 n1)
       (disjoint (sub 32 4 (read-an 32 6 s)) 24 str2 n2)
       (disjoint str1 n1 str2 n2)
       (equal* (read-an 32 1 s) (add 32 str1 i*))
       (equal* (read-an 32 0 s) (add 32 str2 i*))
       (equal n (nat-to-uint (read-dn 32 0 s)))
       (equal i* (read-dn 32 1 s))
       (equal i (nat-to-uint i*))
       (not (equal n 0))
       (lessp (slen i n2 lst2) n2)
       (leq n2 n1)
       (lessp i n2)
       (numberp n1)
       (numberp n2)
       (uint-rangep n1 32)
       (uint-rangep n2 32)))

; an intermediate state s1.
(defn strxfrm-s1p (s i* i str1 n1 lst1 j* j str2 n2 lst2)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (sub 32 36 (mc-pc s)) (mc-mem s) 62)
       (mcode-addrp (sub 32 36 (mc-pc s)) (mc-mem s) (strxfrm-code))
       (ram-addrp (sub 32 4 (read-an 32 6 s)) (mc-mem s) 24)
       (ram-addrp str1 (mc-mem s) n1)
       (mem-lst 1 str1 (mc-mem s) n1 lst1)
       (ram-addrp str2 (mc-mem s) n2)
       (mem-lst 1 str2 (mc-mem s) n2 lst2)
       (disjoint (sub 32 4 (read-an 32 6 s)) 24 str1 n1)
       (disjoint (sub 32 4 (read-an 32 6 s)) 24 str2 n2)
       (disjoint str1 n1 str2 n2)
       (equal* (read-an 32 1 s) (add 32 str1 i*))
       (equal* (read-an 32 0 s) (add 32 str2 j*))
       (equal j* (read-dn 32 1 s))
       (equal j (nat-to-uint j*))
       (lessp i n1)
       (lessp (slen j n2 lst2) n2)
       (numberp i*)
       (nat-rangep i* 32)
       (equal i (nat-to-uint i*))
       (numberp n1)
       (numberp n2)
       (uint-rangep n1 32)
       (uint-rangep n2 32)))

; from the initial state s to exit: s --> sn, when n = 0.
(prove-lemma strxfrm-s-sn (rewrite)
     (implies (and (strxfrm-statep s str1 n1 lst1 str2 n2 lst2 n)
                   (zerop n))
              (and (equal (mc-status (stepn s 12)) 'running)
                   (equal (mc-pc (stepn s 12)) (rts-addr s))
```

```
                (mem-lst 1 str1 (mc-mem (stepn s 12)) n1 lst1)
                (equal (uread-dn 32 0 (stepn s 12)) 0)
                (equal (read-rn 32 15 (mc-rfile (stepn s 12)))
                       (add 32 (read-an 32 7 s) 4))
                (equal (read-rn 32 14 (mc-rfile (stepn s 12)))
                       (read-an 32 6 s)))))

(prove-lemma strxfrm-s-sn-rfile (rewrite)
     (implies (and (strxfrm-statep s str1 n1 lst1 str2 n2 lst2 n)
                   (zerop n)
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 12)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strxfrm-s-sn-mem (rewrite)
     (implies (and (strxfrm-statep s str1 n1 lst1 str2 n2 lst2 n)
                   (zerop n)
                   (disjoint x k (sub 32 8 (read-sp s)) 24))
              (equal (read-mem x (mc-mem (stepn s 12)) k)
                     (read-mem x (mc-mem s) k))))

; from the initial state to s0: s --> s0, when n =- 0.
(prove-lemma strxfrm-s-s0 ()
     (implies (and (strxfrm-statep s str1 n1 lst1 str2 n2 lst2 n)
                   (not (zerop n)))
              (strxfrm-s0p (stepn s 9) 0 0 str1 n1 lst1 str2 n2 lst2 n)))

(prove-lemma strxfrm-s-s0-else (rewrite)
     (implies (and (strxfrm-statep s str1 n1 lst1 str2 n2 lst2 n)
                   (not (zerop n)))
              (and (equal (linked-rts-addr (stepn s 9)) (rts-addr s))
                   (equal (linked-a6 (stepn s 9)) (read-an 32 6 s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 9)))
                          (sub 32 4 (read-sp s)))
                   (equal (rn-saved (stepn s 9)) (read-dn 32 2 s)))))

(prove-lemma strxfrm-s-s0-rfile (rewrite)
     (implies (and (strxfrm-statep s str1 n1 lst1 str2 n2 lst2 n)
                   (not (zerop n))
                   (d3-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 9)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strxfrm-s-s0-mem (rewrite)
     (implies (and (strxfrm-statep s str1 n1 lst1 str2 n2 lst2 n)
                   (not (zerop n))
                   (disjoint x k (sub 32 8 (read-sp s)) 24))
              (equal (read-mem x (mc-mem (stepn s 9)) k)
                     (read-mem x (mc-mem s) k))))

; from s1 to exit: s1 --> sn.  By induction.
; base case:  s1 --> sn, when lst2[i] == 0.
(prove-lemma strxfrm-s1-sn-base (rewrite)
```

```
     (implies (and (strxfrm-s1p s i* i str1 n1 lst1 j* j str2 n2 lst2)
                   (equal (get-nth j lst2) 0))
              (and (equal (mc-status (stepn s 8)) 'running)
                   (equal (mc-pc (stepn s 8)) (linked-rts-addr s))
                   (mem-lst 1 str1 (mc-mem (stepn s 8)) n1 (put-nth 0 i lst1))
                   (equal (uread-dn 32 0 (stepn s 8)) j)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 8)))
                          (linked-a6 s))
                   (equal (read-rn 32 15 (mc-rfile (stepn s 8)))
                          (add 32 (read-an 32 6 s) 8)))))

(prove-lemma strxfrm-s1-sn-rfile-base (rewrite)
     (implies (and (strxfrm-s1p s i* i str1 n1 lst1 j* j str2 n2 lst2)
                   (equal (get-nth j lst2) 0)
                   (leq oplen 32)
                   (d2-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 8)))
                     (if (d3-7a2-5p rn)
                         (read-rn oplen rn (mc-rfile s))
                       (head (rn-saved s) oplen)))))

(prove-lemma strxfrm-s1-sn-mem-base (rewrite)
     (implies (and (strxfrm-s1p s i* i str1 n1 lst1 j* j str2 n2 lst2)
                   (equal (get-nth j lst2) 0)
                   (disjoint x k str1 n1))
              (equal (read-mem x (mc-mem (stepn s 8)) k)
                     (read-mem x (mc-mem s) k))))

; induction case:  s1 --> s1.
(prove-lemma strxfrm-s1-s1 (rewrite)
     (implies (and (strxfrm-s1p s i* i str1 n1 lst1 j* j str2 n2 lst2)
                   (not (equal (get-nth j lst2) 0)))
              (and (strxfrm-s1p (stepn s 3) i* i str1 n1 lst1
                                (add 32 j* 1) (add1 j) str2 n2 lst2)
                   (equal (read-rn 32 14 (mc-rfile (stepn s 3)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (linked-a6 (stepn s 3)) (linked-a6 s))
                   (equal (linked-rts-addr (stepn s 3)) (linked-rts-addr s))
                   (equal (read-mem x (mc-mem (stepn s 3)) k)
                          (read-mem x (mc-mem s) k))
                   (equal (rn-saved (stepn s 3)) (rn-saved s)))))

(prove-lemma strxfrm-s1-s1-rfile (rewrite)
     (implies (and (strxfrm-s1p s i* i str1 n1 lst1 j* j str2 n2 lst2)
                   (not (equal (get-nth j lst2) 0))
                   (d3-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 3)))
                     (read-rn oplen rn (mc-rfile s)))))

; put together: s1 --> sn.
(prove-lemma strxfrm-s1p-info (rewrite)
     (implies (strxfrm-s1p s i* i str1 n1 lst1 j* j str2 n2 lst2)
              (equal (lessp j n2) t)))
```

```
(prove-lemma strxfrm-s1-sn (rewrite)
     (let ((sn (stepn s (strxfrm-t2 j n2 lst2))))
       (implies (strxfrm-s1p s i* i str1 n1 lst1 j* j str2 n2 lst2)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (mem-lst 1 str1 (mc-mem sn) n1 (put-nth 0 i lst1))
                     (equal (uread-dn 32 0 sn) (strlen j n2 lst2))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))))))
     ((induct (strxfrm-induct2 s j* j n2 lst2))
      (disable strxfrm-s1p uread-dn)))

(prove-lemma strxfrm-s1-sn-rfile (rewrite)
     (implies
      (and (strxfrm-s1p s i* i str1 n1 lst1 j* j str2 n2 lst2)
           (leq oplen 32)
           (d2-7a2-5p rn))
      (equal (read-rn oplen rn (mc-rfile (stepn s (strxfrm-t2 j n2 lst2))))
             (if (d3-7a2-5p rn)
                 (read-rn oplen rn (mc-rfile s))
               (head (rn-saved s) oplen))))
      ((induct (strxfrm-induct2 s j* j n2 lst2))
       (disable strxfrm-s1p)))

(prove-lemma strxfrm-s1-sn-mem (rewrite)
     (implies (and (strxfrm-s1p s i* i str1 n1 lst1 j* j str2 n2 lst2)
                   (disjoint x k str1 n1))
              (equal (read-mem x (mc-mem (stepn s (strxfrm-t2 j n2 lst2))) k)
                     (read-mem x (mc-mem s) k)))
     ((induct (strxfrm-induct2 s j* j n2 lst2))
      (disable strxfrm-s1p)))

(disable strxfrm-s1p-info)

; from s0 to exit: s0 --> sn.  By induction.
; base case 1. s0 --> sn, when lst2[i] = 0.
(prove-lemma strxfrm-s0-sn-base1 (rewrite)
     (let ((sn (stepn s 8)))
       (implies (and (strxfrm-s0p s i* i str1 n1 lst1 str2 n2 lst2 n)
                     (equal (get-nth i lst2) 0))
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (mem-lst 1 str1 (mc-mem sn) n1 (put-nth 0 i lst1))
                     (equal (uread-dn 32 0 sn) i)
                     (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))))))

(prove-lemma strxfrm-s0-sn-rfile-base1 (rewrite)
     (implies (and (strxfrm-s0p s i* i str1 n1 lst1 str2 n2 lst2 n)
                   (equal (get-nth i lst2) 0)
                   (leq oplen 32)
```

```
                (d2-7a2-5p rn))
           (equal (read-rn oplen rn (mc-rfile (stepn s 8)))
                  (if (d3-7a2-5p rn)
                      (read-rn oplen rn (mc-rfile s))
                    (head (rn-saved s) oplen)))))

(prove-lemma strxfrm-s0-sn-mem-base1 (rewrite)
     (implies (and (strxfrm-s0p s i* i str1 n1 lst1 str2 n2 lst2 n)
                   (equal (get-nth i lst2) 0)
                   (disjoint x k str1 n1))
          (equal (read-mem x (mc-mem (stepn s 8)) k)
                 (read-mem x (mc-mem s) k))))

; base case 2:  s0 --> s1 --> sn, when lst2[i] =- 0 and n-1 == 0.
; s0 --> s1.
(prove-lemma strxfrm-s0-s1 ()
     (implies (and (strxfrm-s0p s i* i str1 n1 lst1 str2 n2 lst2 n)
                   (not (equal (get-nth i lst2) 0))
                   (equal (sub1 n) 0))
              (strxfrm-s1p (stepn s 7) i* i str1 n1 lst1
                           (add 32 i* 1) (add1 i) str2 n2 lst2)))

(prove-lemma strxfrm-s0-s1-else (rewrite)
     (implies (and (strxfrm-s0p s i* i str1 n1 lst1 str2 n2 lst2 n)
                   (not (equal (get-nth i lst2) 0))
                   (equal (sub1 n) 0))
              (and (equal (linked-rts-addr (stepn s 7))
                          (linked-rts-addr s))
                   (equal (linked-a6 (stepn s 7)) (linked-a6 s))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 7)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (rn-saved (stepn s 7)) (rn-saved s)))))

(prove-lemma strxfrm-s0-s1-rfile (rewrite)
     (implies (and (strxfrm-s0p s i* i str1 n1 lst1 str2 n2 lst2 n)
                   (not (equal (get-nth i lst2) 0))
                   (equal (sub1 n) 0)
                   (d3-7a2-5p rn))
              (equal (read-rn oplen rn (mc-rfile (stepn s 7)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strxfrm-s0-s1-mem (rewrite)
     (implies (and (strxfrm-s0p s i* i str1 n1 lst1 str2 n2 lst2 n)
                   (not (equal (get-nth i lst2) 0))
                   (equal (sub1 n) 0)
                   (disjoint x k str1 n1))
              (equal (read-mem x (mc-mem (stepn s 7)) k)
                     (read-mem x (mc-mem s) k))))

; s0 --> sn.
(prove-lemma strxfrm-s0-sn-base2 (rewrite)
     (let ((sn (stepn s (strxfrm-t1 i n2 lst2))))
       (implies (and (strxfrm-s0p s i* i str1 n1 lst1 str2 n2 lst2 n)
                     (not (equal (get-nth i lst2) 0))
```

```
                    (equal (sub1 n) 0))
               (and (equal (mc-status sn) 'running)
                    (equal (mc-pc sn) (linked-rts-addr s))
                    (mem-lst 1 str1 (mc-mem sn) n1 (put-nth 0 i lst1))
                    (equal (uread-dn 32 0 sn) (strlen (add1 i) n2 lst2))
                    (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
                    (equal (read-rn 32 15 (mc-rfile sn))
                           (add 32 (read-an 32 6 s) 8)))))
     ((use (strxfrm-s0-s1))
      (disable strxfrm-s0p strxfrm-s1p uread-dn)))

(prove-lemma strxfrm-s0-sn-rfile-base2 (rewrite)
     (implies
      (and (strxfrm-s0p s i* i str1 n1 lst1 str2 n2 lst2 n)
           (not (equal (get-nth i lst2) 0))
           (equal (sub1 n) 0)
           (leq oplen 32)
           (d2-7a2-5p rn))
      (equal (read-rn oplen rn (mc-rfile (stepn s (strxfrm-t1 i n2 lst2))))
             (if (d3-7a2-5p rn)
                 (read-rn oplen rn (mc-rfile s))
               (head (rn-saved s) oplen))))
     ((use (strxfrm-s0-s1))
      (disable strxfrm-s0p strxfrm-s1p)))

(prove-lemma strxfrm-s0-sn-mem-base2 (rewrite)
     (implies (and (strxfrm-s0p s i* i str1 n1 lst1 str2 n2 lst2 n)
                   (not (equal (get-nth i lst2) 0))
                   (equal (sub1 n) 0)
                   (disjoint x k str1 n1))
              (equal (read-mem x (mc-mem (stepn s (strxfrm-t1 i n2 lst2))) k)
                     (read-mem x (mc-mem s) k)))
     ((use (strxfrm-s0-s1))
      (disable strxfrm-s0p strxfrm-s1p)))

; induction case: s0 --> s0, when lst2[i] =~ 0 and n-1 =~ 0.
(prove-lemma strxfrm-s0-s0 (rewrite)
     (implies (and (strxfrm-s0p s i* i str1 n1 lst1 str2 n2 lst2 n)
                   (not (equal (get-nth i lst2) 0))
                   (not (equal (sub1 n) 0)))
              (and (strxfrm-s0p (stepn s 7) (add 32 i* 1) (add1 i) str1
                                n1 (put-nth (get-nth i lst2) i lst1)
                                str2 n2 lst2 (sub1 n))
                   (equal (read-rn 32 14 (mc-rfile (stepn s 7)))
                          (read-rn 32 14 (mc-rfile s)))
                   (equal (linked-a6 (stepn s 7)) (linked-a6 s))
                   (equal (linked-rts-addr (stepn s 7)) (linked-rts-addr s))
                   (equal (rn-saved (stepn s 7)) (rn-saved s)))))

(prove-lemma strxfrm-s0-s0-rfile (rewrite)
     (implies (and (strxfrm-s0p s i* i str1 n1 lst1 str2 n2 lst2 n)
                   (not (equal (get-nth i lst2) 0))
                   (not (equal (sub1 n) 0))
                   (d3-7a2-5p rn))
```

```
              (equal (read-rn oplen rn (mc-rfile (stepn s 7)))
                     (read-rn oplen rn (mc-rfile s)))))

(prove-lemma strxfrm-s0-s0-mem (rewrite)
     (implies (and (strxfrm-s0p s i* i str1 n1 lst1 str2 n2 lst2 n)
                   (not (equal (get-nth i lst2) 0))
                   (not (equal (sub1 n) 0))
                   (disjoint x k str1 n1))
              (equal (read-mem x (mc-mem (stepn s 7)) k)
                     (read-mem x (mc-mem s) k)))))

; put together: s0 --> sn.
(prove-lemma strxfrm-s0p-info (rewrite)
     (implies (strxfrm-s0p s i* i str1 n1 lst1 str2 n2 lst2 n)
              (equal (lessp i n2) t)))

(prove-lemma strxfrm-s0-sn (rewrite)
     (let ((sn (stepn s (strxfrm-t0 i n2 lst2 n))))
       (implies (strxfrm-s0p s i* i str1 n1 lst1 str2 n2 lst2 n)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (linked-rts-addr s))
                     (mem-lst 1 str1 (mc-mem sn) n1 (strxfrm1 i lst1 lst2 n))
                     (equal (uread-dn 32 0 sn) (strlen i n2 lst2))
                     (equal (read-rn 32 14 (mc-rfile sn)) (linked-a6 s))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 6 s) 8))))))
     ((induct (strxfrm-induct1 s i* i lst1 lst2 n))
      (disable strxfrm-s0p strxfrm-t1 uread-dn)))

(prove-lemma strxfrm-s0-sn-rfile (rewrite)
     (implies
      (and (strxfrm-s0p s i* i str1 n1 lst1 str2 n2 lst2 n)
           (leq oplen 32)
           (d2-7a2-5p rn))
      (equal (read-rn oplen rn (mc-rfile (stepn s (strxfrm-t0 i n2 lst2 n))))
             (if (d3-7a2-5p rn)
                 (read-rn oplen rn (mc-rfile s))
               (head (rn-saved s) oplen))))
     ((induct (strxfrm-induct1 s i* i lst1 lst2 n))
      (disable strxfrm-s0p strxfrm-t1)))

(prove-lemma strxfrm-s0-sn-mem (rewrite)
     (implies (and (strxfrm-s0p s i* i str1 n1 lst1 str2 n2 lst2 n)
                   (disjoint x k str1 n1))
              (equal (read-mem x (mc-mem (stepn s (strxfrm-t0 i n2 lst2 n))) k)
                     (read-mem x (mc-mem s) k)))
     ((induct (strxfrm-induct1 s i* i lst1 lst2 n))
      (disable strxfrm-s0p strxfrm-t1)))

(disable strxfrm-s0p-info)

; the correctness of strxfrm.
(prove-lemma strxfrm-correctness (rewrite)
     (let ((sn (stepn s (strxfrm-t n2 lst2 n))))
```

```
       (implies (strxfrm-statep s str1 n1 lst1 str2 n2 lst2 n)
                (and (equal (mc-status sn) 'running)
                     (equal (mc-pc sn) (rts-addr s))
                     (equal (read-rn 32 14 (mc-rfile sn))
                            (read-rn 32 14 (mc-rfile s)))
                     (equal (read-rn 32 15 (mc-rfile sn))
                            (add 32 (read-an 32 7 s) 4))
                     (implies (and (d2-7a2-5p rn)
                                   (leq oplen 32))
                              (equal (read-rn oplen rn (mc-rfile sn))
                                     (read-rn oplen rn (mc-rfile s))))
                     (implies (and (disjoint x k str1 n1)
                                   (disjoint x k (sub 32 8 (read-sp s)) 24))
                              (equal (read-mem x (mc-mem sn) k)
                                     (read-mem x (mc-mem s) k)))
                     (equal (uread-dn 32 0 sn) (strxfrm-n n2 lst2 n))
                     (mem-lst 1 str1 (mc-mem sn) n1 (strxfrm lst1 lst2 n)))))
     ((use (strxfrm-s-s0))
      (disable strxfrm-statep strxfrm-s0p linked-rts-addr linked-a6
               uread-dn strxfrm-t0)))

(disable strxfrm-t)

; some properties of strxfrm.
; see file cstring.events.
```

C.31 Theorems About the String Functions

```
; some general theorems.  They are used to establish properties of string
; functions.
(prove-lemma ilessp-lessp (rewrite)
     (implies (and (numberp x)
                   (numberp y))
              (equal (ilessp x y) (lessp x y)))
     ((enable ilessp)))

(prove-lemma idifference-numberp (rewrite)
     (equal (numberp (idifference x y))
            (not (ilessp x y)))
     ((enable idifference iplus ilessp integerp)))

(prove-lemma idifference-equal-0 (rewrite)
     (equal (equal (idifference x y) 0)
            (equal (fix-int x) (fix-int y)))
     ((enable idifference iplus integerp)))

(enable get-nth-0)
(enable put-nth-0)

; the upper bound of strlen.
(prove-lemma strlen-ub ()
     (implies (stringp i n lst)
```

```
             (lessp (strlen i n lst) n)))

; the lower bound of strlen.
(prove-lemma strlen-lb (rewrite)
     (not (lessp (strlen i n lst) i)))

(prove-lemma strlen-01 (rewrite)
     (and (equal (strlen i 0 lst) i)
          (equal (strlen i 1 lst)
                 (if (and (lessp i 1)
                          (not (equal (get-nth 0 lst) 0)))
                     1 i))
          (not (equal (strlen 1 n lst) 0))))

(prove-lemma strlen-ii (rewrite)
     (equal (strlen i i lst) i)
     ((expand (strlen i i lst))))

(prove-lemma strlen-ii1 (rewrite)
     (equal (strlen i (add1 i) lst)
            (if (equal (get-nth i lst) 0) i (add1 i))))

; some properties of strlen.
; lst[j] =~ 0, if i <= j < strlen(lst).
(prove-lemma strlen-non0p ()
     (implies (and (leq i j)
                   (lessp j (strlen i n lst)))
              (not (equal (get-nth j lst) 0))))

; lst[strlen] == 0!
(prove-lemma strlen-0p (rewrite)
     (implies (stringp i n lst)
              (equal (get-nth (strlen i n lst) lst) 0)))

; some properties of strcpy.
(prove-lemma strcpy-get-1 (rewrite)
     (implies (lessp j i)
              (equal (get-nth j (strcpy i lst1 n2 lst2))
                     (get-nth j lst1))))

; strlen(strcpy(lst1, lst2)) == strlen(lst2).
(prove-lemma strcpy-strlen (rewrite)
     (equal (strlen i n2 (strcpy i lst1 n2 lst2))
            (strlen i n2 lst2)))

;  if i <= j <= strlen(lst2), lst1[j] == lst2[j].
(prove-lemma strcpy-cpy (rewrite)
     (implies (and (stringp i n2 lst2)
                   (leq i j)
                   (leq j (strlen i n2 lst2)))
              (equal (get-nth j (strcpy i lst1 n2 lst2))
                     (get-nth j lst2)))
     ((expand (strcpy 0 lst1 n2 lst2))))
```

```
;  if lst2 is a string, then lst1' is also a string.
(prove-lemma strcpy-stringp (rewrite)
     (implies (stringp i n2 lst2)
              (stringp i n2 (strcpy i lst1 n2 lst2))))

; some properties of strcmp.
; (strcmp lst lst) == 0.
(prove-lemma strcmp-id (rewrite)
     (equal (strcmp i n lst lst) 0))

; (strcmp lst1 lst2) < 0 ==> (strcmp lst2 lst1) >= 0.
(prove-lemma strcmp-antisym (rewrite)
     (equal (negativep (strcmp i n lst1 lst2))
            (lessp 0 (strcmp i n lst2 lst1)))
     ((enable idifference iplus)))

; the transitivity of strcmp.
(prove-lemma strcmp-trans-1 (rewrite)
     (implies (and (lst-of-chrp lst1)
                   (lst-of-chrp lst2)
                   (lst-of-chrp lst3)
                   (numberp (strcmp i n lst1 lst2))
                   (numberp (strcmp i n lst2 lst3)))
              (numberp (strcmp i n lst1 lst3))))

; (strcpy lst1 lst2) == lst2.
(prove-lemma strcmp-strcpy (rewrite)
     (equal (strcmp i n lst2 (strcpy i lst1 n lst2)) 0))

; if (strcmp lst1 lst2) == 0, all i <= j < (strlen lst1) lst1[j] == lst2[j].
(prove-lemma strcmp-thm1 ()
     (implies (and (lst-of-chrp lst1)
                   (lst-of-chrp lst2)
                   (equal (strcmp i n lst1 lst2) 0)
                   (leq i j)
                   (lessp j (strlen i n lst1)))
              (equal (get-nth j lst1) (get-nth j lst2)))
     ((enable idifference iplus)))

(defn-sk strcmp-sk (n1 lst1 lst2)
  (exists j
          (and (forall i (implies (lessp i j)
                                  (equal (get-nth i lst1) (get-nth i lst2))))
               (equal (strcmp 0 n1 lst1 lst2)
                      (idifference (get-nth j lst1) (get-nth j lst2))))))

(defn strcmp-j (i n1 lst1 lst2)
  (if (lessp i n1)
      (if (equal (get-nth i lst1) (get-nth i lst2))
          (if (equal (get-nth i lst1) (null))
              (fix i)
            (strcmp-j (add1 i) n1 lst1 lst2))
        (fix i))
    (fix i))
```

```
  ((lessp (difference n1 i))))

(prove-lemma strcmp-j-1 (rewrite)
     (implies (and (lessp i (strcmp-j i1 n1 lst1 lst2))
                   (leq i1 i))
              (equal (get-nth i lst1) (get-nth i lst2)))
     ((enable get-nth-0)))

(prove-lemma strcmp-j-2 (rewrite)
     (implies (not (equal (strcmp i1 n1 lst1 lst2) 0))
              (equal (strcmp i1 n1 lst1 lst2)
                     (idifference (get-nth (strcmp-j i1 n1 lst1 lst2) lst1)
                                  (get-nth (strcmp-j i1 n1 lst1 lst2) lst2))))
     ((enable get-nth-0)))

(prove-lemma strcmp-thm2 ()
     (implies (not (equal (strcmp 0 n1 lst1 lst2) 0))
              (strcmp-sk n1 lst1 lst2))
     ((use (strcmp-sk (j (strcmp-j 0 n1 lst1 lst2))))))

(disable strcmp-j-1)
(disable strcmp-j-2)

; some properties of strcat.
(prove-lemma strcpy1-get-1 (rewrite)
     (implies (lessp j i1)
              (equal (get-nth j (strcpy1 i1 lst1 i2 n2 lst2))
                     (get-nth j lst1))))

(defn strlen1 (i n lst)
  (if (lessp i n)
      (if (equal (get-nth i lst) (null))
          0
        (add1 (strlen1 (add1 i) n lst)))
    0)
  ((lessp (difference n i))))

(prove-lemma strlen1-strlen (rewrite)
     (equal (strlen1 i n lst)
            (difference (strlen i n lst) i)))

(disable strlen1-strlen)

(prove-lemma strcpy1-get-2 (rewrite)
     (implies (lessp (plus i1 (strlen1 i2 n2 lst2)) j)
              (equal (get-nth j (strcpy1 i1 lst1 i2 n2 lst2))
                     (get-nth j lst1)))
     ((induct (strcpy1 i1 lst1 i2 n2 lst2))))

(prove-lemma get-nth-plus-0 (rewrite)
     (implies (not (numberp j))
              (equal (get-nth (plus i j) lst)
                     (get-nth i lst)))
     ((enable get-nth)))
```

```
(prove-lemma plus-sub1-add1 (rewrite)
     (equal (sub1 (plus x (add1 y)))
            (plus x y)))

(prove-lemma strcpy1-get-3 (rewrite)
     (implies (and (leq i1 j)
                   (lessp j (plus i1 (strlen1 i2 n2 lst2))))
              (equal (get-nth j (strcpy1 i1 lst1 i2 n2 lst2))
                     (get-nth (plus i2 (difference j i1)) lst2)))
     ((expand (strcpy1 0 lst1 i2 n2 lst2))))

; all 0 <= j < (strlen lst1) lst1'[j] == lst1[j].
(prove-lemma strcat-get-1 (rewrite)
     (implies (lessp j (strlen 0 n1 lst1))
              (equal (get-nth j (strcat n1 lst1 n2 lst2))
                     (get-nth j lst1))))

; all (strlen lst1) <= j < (strlen lst1)+(strlen lst2) lst1'[j] == lst2[j].
(prove-lemma strcat-get-2 (rewrite)
     (implies (and (stringp 0 n1 lst1)
                   (leq (strlen 0 n1 lst1) j)
                   (lessp j (plus (strlen 0 n1 lst1) (strlen 0 n2 lst2))))
              (equal (get-nth j (strcat n1 lst1 n2 lst2))
                     (get-nth (difference j (strlen 0 n1 lst1)) lst2)))
     ((use (strcpy1-get-3 (i1 (strlen 0 n1 lst1)) (i2 0)))
      (enable strlen1-strlen)))

; some properties of strchr.
; lst[strchr] == ch,  if strchr returns non-f.
(prove-lemma strchr-thm1 (rewrite)
     (implies (strchr i n lst ch)
              (equal (get-nth (strchr i n lst ch) lst)
                     ch)))

; all i <= j < strchr, lst[j] =~ ch.  i.e. strchr is the first one.
(prove-lemma strchr-thm2 ()
     (implies (and (leq i j)
                   (lessp j (strchr i n lst ch)))
              (not (equal (get-nth j lst) ch))))

; all i <= j < strlen, lst[j] =~ ch, if strchr returns f.
(prove-lemma strchr-thm3 ()
     (implies (and (not (strchr i n lst ch))
                   (leq i j)
                   (lessp j (strlen i n lst)))
              (not (equal (get-nth j lst) ch))))

; ch is not equal to the null character, if strchr returns f.
(prove-lemma strchr-thm4 ()
     (implies (and (stringp i n lst)
                   (not (strchr i n lst ch)))
              (not (equal ch 0))))
```

```
; strcpy and strchr.
(prove-lemma strcpy-strchr (rewrite)
     (equal (strchr i n (strcpy i lst1 n lst2) ch)
            (strchr i n lst2 ch)))

; some properties of memset.
; lemmas about memset1.
(prove-lemma memset1-get-1 (rewrite)
     (implies (lessp j i)
              (equal (get-nth j (memset1 i n lst ch))
                     (get-nth j lst))))

; all i <= j < i+n, lst'[j] == ch.
(prove-lemma memset1-thm1 (rewrite)
     (implies (and (leq i j)
                   (lessp j (plus i n)))
              (equal (get-nth j (memset1 i n lst ch))
                     ch))
     ((induct (memset1 i n lst ch))))

(prove-lemma memset1-thm2 (rewrite)
     (implies (and (leq (plus i n) j)
                   (not (zerop n)))
              (equal (get-nth j (memset1 i n lst ch))
                     (get-nth j lst)))
     ((induct (memset1 i n lst ch))))

; all i <= j < n, lst'[j] == ch.
(prove-lemma memset-thm1 (rewrite)
     (implies (and (leq i j)
                   (lessp j n))
              (equal (get-nth j (memset n lst ch))
                     ch)))

; all j >= n, lst'[j] == lst[j].
(prove-lemma memset-thm2 (rewrite)
     (implies (and (leq n j)
                   (numberp n))
              (equal (get-nth j (memset n lst ch))
                     (get-nth j lst))))

; some properties of memchr.
(prove-lemma memchr1-thm1 (rewrite)
     (implies (memchr1 i n lst ch)
              (equal (get-nth (memchr1 i n lst ch) lst)
                     ch)))

; lst[memchr] == ch,  if memchr returns non-f.
(prove-lemma memchr-thm1 (rewrite)
     (implies (memchr n lst ch)
              (equal (get-nth (memchr n lst ch) lst)
                     ch)))

(prove-lemma memchr1-thm2 ()
```

```
     (implies (and (leq i j)
                   (lessp j (memchr1 i n lst ch)))
              (not (equal (get-nth j lst) ch))))

; all j < memchr, lst[j] =~ ch.  i.e. memchr is the first one.
(prove-lemma memchr-thm2 ()
     (implies (lessp j (memchr n lst ch))
              (not (equal (get-nth j lst) ch)))
     ((use (memchr1-thm2 (i 0)))))

(prove-lemma memchr1-thm3 ()
     (implies (and (not (memchr1 i n lst ch))
                   (leq i j)
                   (lessp j (plus i n)))
              (not (equal (get-nth j lst) ch)))
     ((induct (memchr1 i n lst ch))))

; all j < n, lst[j] =~ ch, if memchr returns f.
(prove-lemma memchr-thm3 ()
     (implies (and (not (memchr n lst ch))
                   (lessp j n))
              (not (equal (get-nth j lst) ch)))
     ((use (memchr1-thm3 (i 0)))))

; some properties of strrchr.
(prove-lemma strrchr-la1 (rewrite)
     (implies (numberp j)
              (strrchr i n lst ch j)))

(prove-lemma strrchr-strchr-la (rewrite)
     (implies (not (strchr i n lst ch))
              (equal (strrchr i n lst ch j) j)))

(prove-lemma strrchr-la2 ()
     (implies (strchr i n lst ch)
              (equal (get-nth (strrchr i n lst ch j) lst)
                     ch))
     ((induct (strrchr i n lst ch j))
      (expand (strrchr i n lst (get-nth i lst) j))))

; strrchr finds ch in the string.
(prove-lemma strrchr-thm1 (rewrite)
     (implies (strrchr i n lst ch f)
              (equal (get-nth (strrchr i n lst ch f) lst)
                     ch))
     ((use (strrchr-la2 (j f)))))

(prove-lemma strchr-get (rewrite)
     (implies (and (leq i j)
                   (lessp j (strlen i n lst)))
              (strchr i n lst (get-nth j lst))))

(prove-lemma strrchr-la3 ()
     (implies (and (strchr i n lst ch)
```

```
                (lessp (strrchr i n lst ch k) j)
                (lessp j (strlen i n lst)))
           (not (equal (get-nth j lst) ch)))
     ((expand (strrchr i n lst (get-nth i lst) k))))

; strrchr returns the last one.
(prove-lemma strrchr-thm2 ()
     (implies (and (strrchr i n lst ch f)
                (lessp (strrchr i n lst ch f) j)
                (lessp j (strlen i n lst)))
           (not (equal (get-nth j lst) ch)))
     ((use (strrchr-la3 (k f)))))

; no means none.
(prove-lemma strrchr-thm3 ()
     (implies (and (not (strrchr i n lst ch k))
                (leq i j)
                (lessp j (strlen i n lst)))
           (not (equal (get-nth j lst) ch))))

; ch is not equal to the null character, if strrchr returns f.
(prove-lemma strrchr-thm4 ()
     (implies (and (stringp i n lst)
                (not (strrchr i n lst ch k)))
           (not (equal ch 0))))

; some properties of memcmp.
(prove-lemma memcmp1-id (rewrite)
     (equal (memcmp1 i n lst lst) 0))

(prove-lemma memcmp-id (rewrite)
     (equal (memcmp n lst lst) 0))

(prove-lemma memcmp1-thm1 ()
     (implies (and (lst-of-chrp lst1)
                (lst-of-chrp lst2)
                (equal (memcmp1 i n lst1 lst2) 0)
                (leq i j)
                (lessp j (plus i n)))
           (equal (get-nth j lst1) (get-nth j lst2)))
     ((induct (memcmp1 i n lst1 lst2))
      (enable idifference iplus)))

(prove-lemma memcmp-thm1 ()
     (implies (and (lst-of-chrp lst1)
                (lst-of-chrp lst2)
                (equal (memcmp n lst1 lst2) 0)
                (lessp j n))
           (equal (get-nth j lst1) (get-nth j lst2)))
     ((use (memcmp1-thm1 (i 0)))))

(defn-sk memcmp-sk (n lst1 lst2)
  (exists j
          (and (forall i (implies (lessp i j)
```

```
                                (equal (get-nth i lst1) (get-nth i lst2))))
              (equal (memcmp n lst1 lst2)
                     (idifference (get-nth j lst1) (get-nth j lst2))))))

(defn memcmp-j (i n lst1 lst2)
  (if (equal (get-nth i lst1) (get-nth i lst2))
      (if (equal (sub1 n) 0)
          (fix i)
        (memcmp-j (add1 i) (sub1 n) lst1 lst2))
    (fix i)))

(prove-lemma memcmp-j-1 (rewrite)
     (implies (and (lessp i (memcmp-j i1 n1 lst1 lst2))
                   (leq i1 i))
              (equal (get-nth i lst1) (get-nth i lst2)))
     ((enable get-nth-0)))

(prove-lemma memcmp-j-2 (rewrite)
     (implies (not (equal (memcmp1 i1 n1 lst1 lst2) 0))
              (equal (memcmp1 i1 n1 lst1 lst2)
                     (idifference (get-nth (memcmp-j i1 n1 lst1 lst2) lst1)
                                  (get-nth (memcmp-j i1 n1 lst1 lst2) lst2))))
     ((enable get-nth-0)))

(prove-lemma memcmp-thm2 ()
     (implies (not (equal (memcmp n lst1 lst2) 0))
              (memcmp-sk n lst1 lst2))
     ((use (memcmp-sk (j (memcmp-j 0 n lst1 lst2))))))

(disable memcmp-j-1)
(disable memcmp-j-2)

; some properties of strncat.
(prove-lemma strcpy2-get-1 (rewrite)
     (implies (lessp j i1)
              (equal (get-nth j (strcpy2 i1 lst1 i2 n2 lst2))
                     (get-nth j lst1))))

(prove-lemma strlen1-ii (rewrite)
     (equal (strlen1 i i lst) 0)
     ((expand (strlen1 i i lst))))

(prove-lemma strcpy2-get-2 (rewrite)
     (implies (and (leq i1 j)
                   (lessp j (plus i1 (strlen1 i2 (plus i2 n2) lst2))))
              (equal (get-nth j (strcpy2 i1 lst1 i2 n2 lst2))
                     (get-nth (difference (plus i2 j) i1) lst2)))
     ((induct (strcpy2 i1 lst1 i2 n2 lst2))
      (expand (strcpy2 i1 lst1 i2 n2 lst2)
              (strcpy2 0 lst1 i2 n2 lst2))
      (enable plus-add1-1)))

; all j < (strlen lst1), lst'[j] == lst1[j].
(prove-lemma strncat-get-1 (rewrite)
```

```
     (implies (lessp j (strlen 0 n1 lst1))
              (equal (get-nth j (strncat n1 lst1 n2 lst2))
                     (get-nth j lst1))))

; all (strlen lst1) <= j < (strlen lst1)+(strlen lst2),
; lst'[j] == lst2[j-(strlen lst1)].
(prove-lemma strncat-get-2 (rewrite)
     (implies (and (stringp 0 n1 lst1)
                   (leq (strlen 0 n1 lst1) j)
                   (lessp j (plus (strlen 0 n1 lst1) (strlen 0 n2 lst2))))
              (equal (get-nth j (strncat n1 lst1 n2 lst2))
                     (get-nth (difference j (strlen 0 n1 lst1)) lst2)))
     ((use (strcpy2-get-2 (i1 (strlen 0 n1 lst1)) (i2 0)))
      (enable strlen1-strlen)))

; some properties of strncpy.
(prove-lemma zero-list1-la (rewrite)
     (implies (lessp j i)
              (equal (get-nth j (zero-list1 i n lst))
                     (get-nth j lst))))

(prove-lemma zero-list1-get (rewrite)
     (implies (and (leq i j)
                   (lessp j (sub1 (plus i n))))
              (equal (get-nth j (zero-list1 i n lst)) 0))
     ((induct (zero-list1 i n lst))))

(prove-lemma zero-list-get (rewrite)
     (implies (and (leq i j)
                   (lessp j (plus i n)))
              (equal (get-nth j (zero-list i n lst)) 0))
     ((use (zero-list1-get (i (add1 i)) (lst (put-nth 0 i lst))))))

(prove-lemma strncpy1-get (rewrite)
     (implies (lessp j i)
              (equal (get-nth j (strncpy1 i n lst1 lst2))
                     (get-nth j lst1))))

(disable zero-list)

(prove-lemma strncpy1-strlen ()
     (equal (strlen i (plus i n) (strncpy1 i n lst1 lst2))
            (strlen i (plus i n) lst2))
     ((induct (strncpy1 i n lst1 lst2))
      (enable plus-add1-1)))

(prove-lemma strcpy-0 (rewrite)
     (equal (strncpy 0 lst1 lst2) lst1))

; the length of (strncpy lst1 lst2) equals the length of lst2.
(prove-lemma strncpy-strlen (rewrite)
     (equal (strlen 0 n (strncpy n lst1 lst2))
            (if (zerop n)
                (strlen 0 n lst1)
```

```
              (strlen 0 n lst2)))
     ((use (strncpy1-strlen (i 0)))))

(prove-lemma strncpy1-cpy (rewrite)
     (implies (and (leq i j)
                   (lessp j (strlen i (plus i n) lst2)))
              (equal (get-nth j (strncpy1 i n lst1 lst2))
                     (get-nth j lst2)))
     ((induct (strncpy1 i n lst1 lst2))
      (expand (strncpy1 i n lst1 lst2)
              (strncpy1 0 n lst1 lst2))
      (enable plus-add1-1)))

; all j < (strlen lst2), lst'[j] == lst2[j].
(prove-lemma strncpy-cpy (rewrite)
     (implies (lessp j (strlen 0 n lst2))
              (equal (get-nth j (strncpy n lst1 lst2))
                     (get-nth j lst2)))
     ((use (strncpy1-cpy (i 0)))))

(prove-lemma strncpy1-0s (rewrite)
     (implies (and (leq (strlen i (plus i n) lst2) j)
                   (lessp j (plus i n)))
              (equal (get-nth j (strncpy1 i n lst1 lst2)) 0))
     ((induct (strncpy1 i n lst1 lst2))
      (enable plus-add1-1)))

; all (strlen lst2) <= j < n, lst'[j] == 0.
(prove-lemma strncpy-0s (rewrite)
     (implies (and (leq (strlen 0 n lst2) j)
                   (lessp j n))
              (equal (get-nth j (strncpy n lst1 lst2)) 0))
     ((use (strncpy1-0s (i 0)))))

; some properties of strncmp.
(prove-lemma strncmp1-id (rewrite)
     (equal (strncmp1 i n lst lst) 0))

(prove-lemma strncmp-id (rewrite)
     (equal (strncmp n lst lst) 0))

(prove-lemma strncmp1-thm1 ()
     (implies (and (lst-of-chrp lst1)
                   (lst-of-chrp lst2)
                   (equal (strncmp1 i n lst1 lst2) 0)
                   (leq i j)
                   (lessp j (strlen i (plus i n) lst1)))
              (equal (get-nth j lst1) (get-nth j lst2)))
     ((induct (strncmp1 i n lst1 lst2))
      (enable idifference iplus plus-add1-1)))

(prove-lemma strncmp-thm1 ()
     (implies (and (lst-of-chrp lst1)
                   (lst-of-chrp lst2)
```

```
                (equal (strncmp n lst1 lst2) 0)
                (lessp j (strlen 0 n lst1)))
           (equal (get-nth j lst1) (get-nth j lst2)))
    ((use (strncmp1-thm1 (i 0)))))

(defn strncmp-j (i n lst1 lst2)
  (if (equal (get-nth i lst1) (get-nth i lst2))
      (if (equal (get-nth i lst1) 0)
          (fix i)
        (if (equal (sub1 n) 0)
            (fix i)
          (strncmp-j (add1 i) (sub1 n) lst1 lst2)))
    (fix i)))

(defn-sk strncmp-sk (n lst1 lst2)
  (exists j
          (and (forall i (implies (lessp i j)
                                  (equal (get-nth i lst1) (get-nth i lst2))))
               (equal (strncmp n lst1 lst2)
                      (idifference (get-nth j lst1) (get-nth j lst2))))))

(prove-lemma strncmp-j-1 (rewrite)
    (implies (and (lessp i (strncmp-j i1 n1 lst1 lst2))
                  (leq i1 i))
             (equal (get-nth i lst1) (get-nth i lst2)))
    ((enable get-nth-0)))

(prove-lemma strncmp-j-2 (rewrite)
    (implies (not (equal (strncmp1 i1 n1 lst1 lst2) 0))
             (equal (strncmp1 i1 n1 lst1 lst2)
                    (idifference (get-nth (strncmp-j i1 n1 lst1 lst2) lst1)
                                 (get-nth (strncmp-j i1 n1 lst1 lst2) lst2))))
    ((enable get-nth-0)))

(prove-lemma strncmp-thm2 ()
    (implies (not (equal (strncmp n lst1 lst2) 0))
             (strncmp-sk n lst1 lst2))
    ((use (strncmp-sk (j (strncmp-j 0 n lst1 lst2))))))

(disable strncmp-j-1)
(disable strncmp-j-2)

; some properties of strpbrk.
(prove-lemma strpbrk-thm1 (rewrite)
    (let ((j (strpbrk i1 n1 lst1 n2 lst2)))
      (implies j
               (strchr1 0 n2 lst2 (get-nth j lst1)))))

(prove-lemma strpbrk-thm2 ()
    (implies (and (leq i1 j)
                  (lessp j (strpbrk i1 n1 lst1 n2 lst2)))
             (not (strchr1 0 n2 lst2 (get-nth j lst1)))))

(prove-lemma strpbrk-thm3 ()
```

```
     (implies (and (not (strpbrk i1 n1 lst1 n2 lst2))
                   (leq i1 j)
                   (lessp j (strlen i1 n1 lst1)))
              (not (strchr1 0 n2 lst2 (get-nth j lst1)))))

; strchr1, which is used to specify several string functions, is a variant
; of strchr.  strchr1 does not have the 'thm4' of strchr.
(prove-lemma strchr1-thm1 (rewrite)
     (implies (strchr1 i n lst ch)
              (equal (get-nth (strchr1 i n lst ch) lst)
                     ch)))

(prove-lemma strchr1-thm2 ()
     (implies (and (leq i j)
                   (lessp j (strchr1 i n lst ch)))
              (not (equal (get-nth j lst) ch))))

(prove-lemma strchr1-thm3 ()
     (implies (and (not (strchr1 i n lst ch))
                   (leq i j)
                   (lessp j (strlen i n lst)))
              (not (equal (get-nth j lst) ch))))

; some properties of strcspn.
(prove-lemma strcspn-thm1 (rewrite)
     (let ((j (strcspn i1 n1 lst1 n2 lst2)))
       (implies j
                (strchr 0 n2 lst2 (get-nth j lst1)))))

(prove-lemma strcspn-thm2 ()
     (implies (and (leq i1 j)
                   (lessp j (strcspn i1 n1 lst1 n2 lst2)))
              (not (strchr 0 n2 lst2 (get-nth j lst1)))))

(prove-lemma strcspn-thm3 ()
     (implies (and (not (strcspn i1 n1 lst1 n2 lst2))
                   (leq i1 j)
                   (lessp j (strlen i1 n1 lst1)))
              (not (strchr 0 n2 lst2 (get-nth j lst1)))))

; some properties of strspn.
(prove-lemma strspn-thm1 (rewrite)
     (let ((j (strspn i1 n1 lst1 n2 lst2)))
       (implies j
                (not (strchr1 0 n2 lst2 (get-nth j lst1))))))

(prove-lemma strspn-thm2 ()
     (implies (and (leq i1 j)
                   (lessp j (strspn i1 n1 lst1 n2 lst2)))
              (strchr1 0 n2 lst2 (get-nth j lst1))))

(prove-lemma strspn-thm3 ()
     (implies (and (not (strspn i1 n1 lst1 n2 lst2))
                   (leq i1 j)
```

```
                (lessp j (strlen i1 n1 lst1)))
           (strchr1 0 n2 lst2 (get-nth j lst1))))

; a generalized defn for strncmp.
(defn strncmp2 (i n lst1 j lst2)
  (if (zerop n)
      0
    (if (equal (get-nth i lst1) (get-nth j lst2))
        (if (equal (get-nth i lst1) 0)
            0
          (strncmp2 (add1 i) (sub1 n) lst1 (add1 j) lst2))
      (idifference (get-nth i lst1) (get-nth j lst2)))))

(prove-lemma strncmp2-non-numberp (rewrite)
     (implies (not (numberp i))
              (equal (strncmp2 i n lst1 j lst2)
                     (strncmp2 0 n lst1 j lst2))))

(prove-lemma strncmp2-0 (rewrite)
     (equal (strncmp2 i 0 lst1 j lst2) 0))

(prove-lemma strncmp1-strncmp2 (rewrite)
     (implies (not (zerop n))
              (equal (strncmp1 i n lst1 lst2)
                     (strncmp2 i n lst1 i lst2))))

(prove-lemma strncmp-strncmp2 (rewrite)
     (equal (strncmp n lst1 lst2)
            (strncmp2 0 n lst1 0 lst2)))

(prove-lemma strncmp2-mcdr-1 (rewrite)
     (equal (strncmp2 i1 n (mcdr j lst1) i2 lst2)
            (strncmp2 (plus i1 j) n lst1 i2 lst2))
     ((enable plus-add1-1)))

(prove-lemma strncmp2-mcdr-2 (rewrite)
     (equal (strncmp2 i1 n lst1 i2 (mcdr j lst2))
            (strncmp2 i1 n lst1 (plus i2 j) lst2))
     ((enable plus-add1-1)))

; a lemma of strlen and mcdr.
(prove-lemma strlen-cdr (rewrite)
     (implies (numberp i)
              (equal (strlen i (sub1 n) (mcdr 1 lst))
                     (sub1 (strlen (add1 i) n lst)))))

(prove-lemma strlen-mcdr (rewrite)
     (implies (numberp i)
              (equal (strlen i n (mcdr k lst))
                     (difference (strlen (plus i k) (plus k n) lst) k)))
     ((enable plus-add1-1)))

; some properties of strstr.
; a lemma for strstr-thm1.
```

```
(prove-lemma strstr1-thm1 (rewrite)
     (let ((j (strstr1 i n1 lst1 n2 lst2 len)))
       (implies (and (numberp j)
                     (equal n (add1 len)))
                (equal (strncmp2 j n lst1 0 lst2) 0)))
     ((disable strchr1 strncmp mcdr)))

; all 0 <= i < (strlen lst2), lst1[j+i] == lst2[i], if strstr == j =- 0.
(prove-lemma strstr-thm1 (rewrite)
     (let ((j (strstr n1 lst1 n2 lst2)))
       (implies (numberp j)
                (equal (strncmp (strlen 0 n2 lst2) (mcdr j lst1) lst2) 0))))

; a few lemmas about strchr1.
(prove-lemma strchr1-nth (rewrite)
     (implies (and (leq i j)
                   (lessp j (strlen i n lst)))
              (numberp (strchr1 i n lst (get-nth j lst)))))

(prove-lemma strlen-strchr1-nth (rewrite)
     (implies (and (leq i j)
                   (lessp j (strlen i n lst)))
              (equal (strlen (add1 (strchr1 i n lst (get-nth j lst))) n lst)
                     (strlen i n lst))))

(prove-lemma strlen-strchr1 (rewrite)
     (implies (numberp (strchr1 i n lst 0))
              (not (lessp (strchr1 i n lst 0) (strlen i n lst)))))

(prove-lemma strchr1-ch0 (rewrite)
     (not (strchr1 i n lst 0)))

(prove-lemma strchr1-nth-first (rewrite)
     (implies (and (numberp (strchr1 i n lst (get-nth j lst)))
                   (leq i j))
              (not (lessp j (strchr1 i n lst (get-nth j lst))))))

; a lemma for strstr-thm2.
(prove-lemma strstr1-thm2 ()
     (implies (and (lst-of-chrp lst1)
                   (lst-of-chrp lst2)
                   (leq i j)
                   (lessp j (strstr1 i n1 lst1 n2 lst2 len))
                   (equal n (add1 len)))
              (not (equal (strncmp2 j n lst1 0 lst2) 0)))
     ((induct (strstr1 i n1 lst1 n2 lst2 len))
      (disable strncmp mcdr)))

; all j < (strstr lst1 lst2), (strncmp (mcdr j lst1) lst2) =- 0.
(prove-lemma strstr-thm2 ()
    (implies (and (lst-of-chrp lst1)
                  (lst-of-chrp lst2)
                  (lessp j (strstr n1 lst1 n2 lst2))
                  (not (zerop n2)))
```

```
            (not (equal (strncmp (strlen 0 n2 lst2) (mcdr j lst1) lst2)
                        0)))
    ((use (strstr1-thm2 (i 0)
                        (len (sub1 (strlen 0 n2 lst2)))
                        (n (strlen 0 n2 lst2))))))

; a lemma for strstr-thm3.
(prove-lemma strstr1-thm3 (rewrite)
     (implies (and (lst-of-chrp lst1)
                   (lst-of-chrp lst2)
                   (not (strstr1 i n1 lst1 n2 lst2 len))
                   (leq i j)
                   (lessp j (strlen i n1 lst1))
                   (equal n (add1 len)))
              (not (equal (strncmp2 j n lst1 0 lst2) 0)))
     ((induct (strstr1 i n1 lst1 n2 lst2 len))
      (disable strncmp mcdr)))

; all j < (strlen lst1), (strncmp (mcdr j lst1) lst2) =~ 0.
(prove-lemma strstr-thm3 (rewrite)
     (implies (and (lst-of-chrp lst1)
                   (lst-of-chrp lst2)
                   (not (strstr n1 lst1 n2 lst2))
                   (lessp j (strlen 0 n1 lst1))
                   (not (zerop n2)))
              (not (equal (strncmp (strlen 0 n2 lst2) (mcdr j lst1) lst2) 0)))
     ((use (strstr1-thm3 (i 0)
                         (len (sub1 (strlen 0 n2 lst2)))
                         (n (strlen 0 n2 lst2))))))

; some properties of strtok.
(prove-lemma strtok-thm1 (rewrite)
     (let ((i (strspn 0 n1 lst1 n2 lst2)))
       (implies (and (not (equal (nat-to-uint s1) 0))
                     (equal (get-nth i lst1) 0))
                (and (equal (strtok-tok s1 last n1 lst1 n2 lst2) 0)
                     (equal (strtok-last s1 last n1 lst1 n2 lst2) 0)
                     (equal (strtok-lst n1 lst1 n2 lst2) lst1))))))

(prove-lemma strtok-thm2 (rewrite)
     (let ((i (strspn 0 n1 lst1 n2 lst2)))
       (implies (and (not (equal (nat-to-uint s1) 0))
                     (not (equal (get-nth i lst1) 0))
                     (equal (get-nth (strcspn i n1 lst1 n2 lst2) lst1) 0)
                     (stringp 0 n1 lst1)
                     (numberp n1))
                (and (equal (strtok-tok s1 last n1 lst1 n2 lst2)
                            (add 32 s1 (strspn* 0 0 n1 lst1 n2 lst2)))
                     (equal (strtok-last s1 last n1 lst1 n2 lst2) 0)
                     (equal (strtok-lst n1 lst1 n2 lst2) lst1))))))

(prove-lemma strspn-strchr1 (rewrite)
     (implies (strspn i1 n1 lst1 n2 lst2)
              (not (strchr1 0 n2 lst2
```

```
                        (get-nth (strspn i1 n1 lst1 n2 lst2) lst1)))))

(prove-lemma strchr-strchr1 (rewrite)
     (implies (not (equal ch 0))
              (equal (strchr i n lst ch) (strchr1 i n lst ch))))

(prove-lemma strspn-true (rewrite)
     (implies (and (stringp i n1 lst1)
                   (stringp 0 n2 lst2))
              (strspn i n1 lst1 n2 lst2))
     ((induct (strspn i n1 lst1 n2 lst2))
      (enable slen) (disable slen-rec)))

(prove-lemma strtok-thm3 (rewrite)
     (let ((i* (strspn* 0 0 n1 lst1 n2 lst2))
           (i (strspn 0 n1 lst1 n2 lst2)))
       (implies (and (not (equal (nat-to-uint s1) 0))
                     (not (equal (get-nth i lst1) 0))
                     (not (equal (get-nth (strcspn i n1 lst1 n2 lst2) lst1) 0))
                     (stringp 0 n1 lst1)
                     (stringp 0 n2 lst2)
                     (numberp n1))
                (and (equal (strtok-tok s1 last n1 lst1 n2 lst2)
                            (add 32 s1 (strspn* 0 0 n1 lst1 n2 lst2)))
                     (equal (strtok-last s1 last n1 lst1 n2 lst2)
                            (add 32 s1 (add 32 (strcspn* i* i n1 lst1 n2 lst2)
                                            1)))
                     (equal (get-nth (strcspn i n1 lst1 n2 lst2)
                                     (strtok-lst n1 lst1 n2 lst2))
                       0))))
     ((disable strspn* strspn strchr strchr1)))

(prove-lemma strtok-thm4 (rewrite)
     (and (equal (strtok-tok 0 0 n1 lst1 n2 lst2) 0)
          (equal (strtok-last 0 0 n1 lst1 n2 lst2) 0)))

(prove-lemma strtok-thm5 (rewrite)
     (let ((i (strspn 0 n1 lst1 n2 lst2)))
       (implies (and (not (equal (nat-to-uint last) 0))
                     (not (equal (get-nth i lst1) 0))
                     (equal (get-nth (strcspn i n1 lst1 n2 lst2) lst1) 0)
                     (stringp 0 n1 lst1)
                     (numberp n1))
                (and (equal (strtok-tok 0 last n1 lst1 n2 lst2)
                            (add 32 last (strspn* 0 0 n1 lst1 n2 lst2)))
                     (equal (strtok-last 0 last n1 lst1 n2 lst2) 0)
                     (equal (strtok-lst n1 lst1 n2 lst2) lst1)))))

(prove-lemma strtok-thm6 (rewrite)
     (let ((i* (strspn* 0 0 n1 lst1 n2 lst2))
           (i (strspn 0 n1 lst1 n2 lst2)))
       (implies (and (not (equal (nat-to-uint last) 0))
                     (not (equal (get-nth i lst1) 0))
                     (not (equal (get-nth (strcspn i n1 lst1 n2 lst2) lst1) 0))
```

```
                    (stringp 0 n1 lst1)
                    (stringp 0 n2 lst2)
                    (numberp n1))
               (and (equal (strtok-tok 0 last n1 lst1 n2 lst2)
                           (add 32 last (strspn* 0 0 n1 lst1 n2 lst2)))
                    (equal (strtok-last 0 last n1 lst1 n2 lst2)
                           (add 32 last
                                (add 32 (strcspn* i* i n1 lst1 n2 lst2) 1)))
                    (equal (get-nth (strcspn i n1 lst1 n2 lst2)
                                    (strtok-lst n1 lst1 n2 lst2))
                      0))))
     ((disable strspn* strspn strchr strchr1)))

(disable strchr-strchr1)

; some properties of memmove.
; memmove is equivalent to lstmov.
(prove-lemma mmov1-lst-cpy (rewrite)
     (equal (mmov1-lst i lst1 lst2 n)
            (lstcpy i lst1 i lst2 n)))

(prove-lemma mmovn-mmov1-lst (rewrite)
     (equal (mmovn-lst h lst1 lst2 i n)
            (mmov1-lst i lst1 lst2 (times h n))))

(prove-lemma lstcpy-nonnumberp (rewrite)
     (implies (not (numberp i1))
              (and (equal (lstcpy i1 lst1 i2 lst2 n)
                          (lstcpy 0 lst1 i2 lst2 n))
                   (equal (lstcpy i2 lst1 i1 lst2 n)
                          (lstcpy i2 lst1 0 lst2 n))))
     ((enable get-nth put-nth)))

(prove-lemma lstcpy-lstcpy-0 (rewrite)
     (equal (lstcpy h1 (lstcpy 0 lst1 0 lst2 h1) h1 lst2 h2)
            (lstcpy 0 lst1 0 lst2 (plus h1 h2)))
     ((use (lstcpy-lstcpy (j1 h1) (j2 h1) (i1 0) (i2 0)))
      (disable lstcpy-lstcpy)))

(prove-lemma mmov1-lst1-cpy (rewrite)
     (implies (leq n i)
              (equal (mmov1-lst1 i lst1 lst2 n)
                     (lstcpy (difference i n) lst1 (difference i n) lst2 n)))
     ((enable lstcpy-cpy1)))

(prove-lemma lstcpy-lstcpy-1 (rewrite)
     (implies (and (equal j1 (plus h1 i1))
                   (equal j2 (plus h1 i2)))
              (equal (lstcpy i1 (lstcpy j1 lst1 j2 lst2 h2) i2 lst2 h1)
                     (lstcpy i1 lst1 i2 lst2 (plus h1 h2))))
     ((induct (lstcpy i1 lst1 i2 lst2 h1))
      (enable put-nth-0 get-nth-0)))

(prove-lemma times-lessp-dual (rewrite)
```

```
     (implies (lessp x y)
              (equal (lessp x (times y z)) (not (zerop z)))))

(prove-lemma mmovn-lst1-mmov1-lst (rewrite)
     (implies (leq (times h n) i)
              (equal (mmovn-lst1 h lst1 lst2 i n)
                     (mmov1-lst (difference i (times h n)) lst1 lst2
                                (times h n))))
     ((induct (mmovn-lst1 h lst1 lst2 i n))))

(prove-lemma quotient-shrink-fast-1 (rewrite)
     (equal (lessp x (times y (quotient x y))) f))

(prove-lemma memmove-mmov1-lst (rewrite)
     (equal (memmove str1 str2 n lst1 lst2)
            (if (zerop n)
                lst1
              (if (equal (nat-to-uint str1) (nat-to-uint str2))
                  lst2
                (mmov1-lst 0 lst1 lst2 n))))
     ((use (remainder-quotient
            (x (difference (plus n (remainder (nat-to-uint str1) 4)) 4))
            (y 4))
           (remainder-quotient
            (x (difference n (remainder (plus n (nat-to-uint str1)) 4)))
            (y 4)))
      (disable remainder-quotient remainder quotient plus difference)))

; for k < i, lstcpy(lst1,lst2)[k] == lst1[k].
(prove-lemma lstcpy-get-1 (rewrite)
     (implies (lessp k i)
              (equal (get-nth k (lstcpy i lst1 j lst2 n))
                     (get-nth k lst1))))

; for i <= i1 < i+n, lstcpy(lst1,lst2)[i1] == lst2[i1].
(prove-lemma lstcpy-get-2 (rewrite)
     (implies (and (leq i i1)
                   (lessp i1 (plus i n)))
              (equal (get-nth i1 (lstcpy i lst1 j lst2 n))
                     (get-nth (difference (plus j i1) i) lst2)))
     ((induct (lstcpy i lst1 j lst2 n))
      (enable get-nth-0 put-nth-0 plus-add1-1)))

; for 0 <= i < n, mmov1-lst(lst1,lst2)[i] == lst2[i].
(prove-lemma mmov1-lst-thm1 (rewrite)
     (implies (lessp i1 n)
              (equal (get-nth i1 (mmov1-lst 0 lst1 lst2 n))
                     (get-nth i1 lst2)))
     ((use (lstcpy-get-2 (i 0) (j 0)))
      (enable get-nth-0)))

; for 0 <= i < n, memmove(lst1,lst2,n)[i] == lst2[i].
(prove-lemma memmove-thm1 (rewrite)
     (implies (lessp i n)
```

```
        (equal (get-nth i (memmove str1 str2 n lst1 lst2))
               (get-nth i lst2)))
((enable get-nth-0)
 (disable memmove)))
```

BIBLIOGRAPHY

[1] Gordon Bell and Allen Newell. The PMS and ISP descriptive systems for computer structures. In *Proceedings of the Spring Joint Computer Conference*. AFIPS Press, 1970.

[2] William Bevier. *A Verified Operating System Kernel*. PhD thesis, University of Texas at Austin, 1987.

[3] William Bevier, Warren Hunt, J Strother Moore, and William Young. Special issue on system verification. *Journal of Automated Reasoning*, 5(4), 1989.

[4] Jonathan Bowen. The formal specification of a microprocessor instruction set, technical monograph PRG-60. Technical report, Oxford University, January 1986.

[5] R. S. Boyer and J S. Moore. A verification condition generator for FORTRAN. In R. S. Boyer and J S. Moore, editors, *The Correctness Problem in Computer Science*. Academic Press, London, 1981.

[6] Robert S. Boyer and J Strother Moore. Proving theorems about LISP functions. *Journal of the ACM*, 22(1), 1975.

[7] Robert S. Boyer and J Strother Moore. *A Computational Logic*. Academic Press, New York, 1979.

[8] Robert S. Boyer and J Strother Moore. Program verification. *Journal of Automated Reasoning*, 1(1):17–23, 1985.

[9] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[10] Robert S. Boyer and J Strother Moore. MJRTY - a fast majority vote algorithm. In Robert S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 105–117. Kluwer Academic, 1991.

[11] Robert S. Boyer and Yuan Yu. Automated correctness proofs of machine code programs for a commercial microprocessor. Technical Report TR-91-33, Computer Sciences Department, University of Texas at Austin, 1991.

[12] Robert S. Boyer and Yuan Yu. A formal specification of some user mode instructions for the motorola 68020. Technical Report TR-92-04, Computer Sciences Department, University of Texas at Austin, 1992.

[13] D.L. Clutterbuck and B.A. Carré. The verification of low-level code. *IEE Software Engineering Journal*, May 1988.

[14] Avra Cohn. A proof of correctness of the Viper microprocessor: The first level. Technical Report 104, University of Cambridge, January 1987.

[15] J. V. Cook. Verification of the C/30 microcode using the State Delta Verification System (SDVS). In *13th National Computer Security Conference*, volume 1, pages 20–31, 1990.

[16] D. Good, et al. Report on the language GYPSY version 2.0. Technical Report ICSCA-CMP-10, Institute for Computing Science and Computer Applications, University of Texas at Austin, 1978.

[17] A. Falkoff, K. Iverson, and E. Sussenguth. A formal description of system/360. *IBM Systems Journal*, 3(3):198–262, 1964.

[18] James R. Farr. A formal specification of the Transputer instruction set. Master's thesis, Oxford, September 1987.

[19] Robert W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics, American Mathematical Society*, pages 19–32, Providence, Rhode Island, 1967.

[20] Herman H. Goldstine and John von Neumann. Planning and coding problems for an electronic computing instrument. In *John von Neumann, Collected Works*, volume V, pages 34–235. Pergamon Press, Oxford, 1961.

[21] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*. Springer-Verlag, New York, 1979.

[22] Mike Gordon. LCF-LSM, a system for specifying and verifying hardware. Technical Report TR 41, Computer Laboratory, University of Cambridge, September 1983.

[23] C.A.R. Hoare. An axiomatic basis for computer programming. *The Communication of ACM*, 12(10):576–583, 1969.

[24] Warren A. Hunt and B. Brock. A formal HDL and its use in the FM9001 verification. In *Proceedings of the Royal Society*, 1992.

[25] S. Igarashi, R.L. London, and D.C. Luckham. Automatic program verification I: A logical basis and its implementation. Technical Report ISI/RR-73-11, Information Science Institute, USC, 1973.

[26] I.M. O'Neill, et al. The formal verification of safety-critical assembly code. In *Safety of Computer Control System 1988*. Pergamon Press, November 1988.

[27] Kenneth E. Iverson. *A Programming Language*. John Wiley and Sons, New York, 1962.

[28] ISO Committee JTC1/SC22/WG14. *ISO/IEC Standard 9899:1990*. International Standards Organization, Geneva, 1990.

[29] Matt Kaufmann. A user's manual for an interactive enhancement to the Boyer-Moore theorem prover. Technical Report CLI-19, Computational Logic, Inc., May 1988.

[30] Matt Kaufmann. DEFN-SK: An extension of the Boyer-Moore theorem prover to handle first-order quantifiers. Technical Report CLI-43, Computational Logic, Inc., 1989.

[31] Matt Kaufmann. An integer library for Nqthm. Technical Report CLI Internal 182, Computational Logic, Inc., March 1990.

[32] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, Englewood Cliff, New Jersey, 1988.

[33] J. C. King. *A Program Verifier*. PhD thesis, Carnegie-Mellon University, 1969.

[34] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, Reading, Massachusetts, 1981.

[35] Tim Leonard. Specification of computer architectures: A survey and annotated bibliography. Technical Report 188, University of Cambridge, January 1990.

[36] W. D. Maurer. An IBM 370 assembly language verifier. In *Proceedings of the 16th Annual technical Symposium on Systems and Software: Operational Reliability and Performance Assurance*. ACM, June 1974.

[37] W. D. Maurer. Some correctness principles for machine language program and microprocessors. In *Proceedings of the Seventh Annual Workshop on Microprogramming*, Palo Alto, CA, 1974.

[38] John McCarthy. Computer programs for checking mathematical proofs. In *Recursive Function Theory, Proceedings of a Symposium in Pure Mathematics*, volume V, pages 219–227, Providence, Rhode Island, 1962. American Mathematical Society.

[39] John McCarthy. Towards a mathematical science of computation. In *Proceedings of IFIP Congress*, pages 21–28, 1962.

[40] John McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Mathematical Aspects of Computer Science, Proc. Symp. Appl. Math., American Mathematical Society*, volume XIX, Providence, Rhode Island, 1967.

[41] Motorola, Inc. *MC68020 32-bit Microprocessor User's Manual.* Prentice Hall, New Jersey, 1989.

[42] Ministry of Defense (Britain). Interim defense standard 00-55, requirements for the procurement of safety critical software in defense equipment. Technical report, Directorate of Standardization, Ministry of Defense, Kentigern House 65, Brown Street, Glasgow G2 8EX, Great Britain, 1989.

[43] P. J. Plauger. Private communication.

[44] P. J. Plauger. *The Standard C Library.* Prentice Hall, New Jersey, 1992.

[45] Wolfgang Polak. *Compiler Specification and Verification.* Springer-Verlag, Berlin, 1981.

[46] Phillip Rose. A partial specification of the M68000 microprocessor. Master's thesis, Oxford, September 1987.

[47] D.P. Siewiorek, Gordon Bell, and Allen Newell. *Computer Structures: Principles and examples.* McGraw-Hill, 1982.

[48] Richard L. Sites. *Alpha Architecture Reference Manual.* Digital Press, Bedford, Mass., 1992.

[49] SPARC International, Inc. *The SPARC Architecture Manual, Version 8.* SPARC International, Inc., Menlo Park, California, 1991.

[50] Chris Torek. Private communication.

[51] Alan M. Turing. On checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69. Univ. Math. Laboratory, Cambridge, 1949.

[52] Warren A. Hunt, Jr. *FM8501: A Verified Microprocessor.* PhD thesis, University of Texas at Austin, 1985.

[53] The ANSI Committee X3J11. *ANSI Standard X3.159-1989.* American National Standards Institute, New York, 1989.

Index

a-mapping, 166
abs, 146
abs-lessp-int-rangep, 237
absolute-long, 163
absolute-short, 163
add, 26, 30, 32, 39–41, 49, 51, 52, 56, 59, 60, 63, 65, 70, 72, 76–78, 80–84, 87–89, 102, 105, 106
add, 145, 247
add-0, 245
add-adder, 247, 249
add-addr-modep1, 168
add-addr-modep2, 168
add-addx-c, 277, 291
add-addx-v, 285, 291
add-associativity, 39
add-associativity, 246
add-bcs, 284
add-bmi, 288
add-bmi-crock1, 234, 291
add-bmi-crock2, 234, 291
add-bvs, 286
add-c, 167, 278
add-c-bitp, 270
add-cancel, 248
add-cancel0, 248
add-commutativity, 246, 329
add-commutativity1, 246, 329
add-commutativity2, 248
add-cvznx, 167
add-effect, 167
add-equal-cancel-1, 275
add-evenp, 227
add-fringe, 274
add-group, 170
add-group-h, 330
add-head, 247
add-ins1, 168
add-ins2, 168
add-int, 40
add-int, 250
add-leq, 246
add-mapping, 168
add-n, 167, 327
add-n-bitp, 270
add-nat-la, 243, 245
add-nat-rangep, 246
add-neg-0, 274
add-neg-adder, 247, 251
add-neg-cancel, 274
add-non-numberp, 292, 329
add-of-0, 245
add-plus, 292, 293
add-sub, 247
add-sub-cancel, 247
add-tree, 274
add-tree-add-fringe, 275
add-tree-append, 275
add-tree-bagdiff, 275
add-tree-delete, 276
add-tree-delete-cond, 275
add-tree-delete-equal, 275
add-tree-nat-rangep, 275
add-uint, 40
add-uint, 244
add-uintxx, 353
add-uintxxx, 353, 477
add-v, 167, 327
add-v-bitp, 270
add-z, 167, 327
add-z-bitp, 270
add1-int-rangep, 391, 420
add1-int-rangepxx, 393, 421
adda-addr-modep, 168
adda-ins, 168
adder, 26
adder, 145, 249
adder-associativity, 245
adder-cancel, 248
adder-commutativity, 245
adder-commutativity1, 245
adder-head, 246
adder-int, 249
adder-int-bridge, 249
adder-int-end, 233, 249
adder-lognot, 228
adder-nat-la, 243, 245
adder-nat-rangep, 246
adder-shift-carry, 245
adder-uint, 244
addi-addr-modep, 217
addi-ins, 217
addi-subgroup, 217
addq-addr-modep, 208
addq-ins, 209
addr-disp, 160
addr-index, 163
addr-index-bd, 161
addr-index-disp, 161
addr-index1, 163

addr-index2, 162
addr-index2-mem, 330
addr-index3, 162
addr-indirect, 160
addr-modep, 33
addr-modep, 165
addr-postinc, 160
addr-predec, 160
addx-c, 169, 278
addx-c-bitp, 270
addx-c-la, 277, 291
addx-cvznx, 169
addx-effect, 169
addx-ins1, 169
addx-ins2, 169
addx-n, 169, 327
addx-n-bitp, 270
addx-v, 169, 327
addx-v-bitp, 270
addx-v-crock1, 234, 286
addx-v-crock2, 234, 286
addx-v-la, 285, 291
addx-z, 169, 327
addx-z-bitp, 270
addy-y, 249
addy-y1, 249
AKCL Common Lisp compiler, ii
alterable-addr-modep, 165
an-direct, 159
an-direct-modep, 166
and-addr-modep1, 173
and-addr-modep2, 173
and-cvznx, 173
and-effect, 173
and-group, 176
and-group-h, 331
and-ins1, 173
and-ins2, 174
and-mapping, 174
and-n, 173, 328
and-n-bitp, 271
and-not-zerop-tree, 125
and-not-zerop-tree-delete, 126, 128
and-z, 173, 327
and-z-bitp, 271
and-z-commutativity, 353
andi-addr-modep, 216
andi-ins, 216
andi-subgroup, 216
andi-to-ccr-ins, 216
app, 25–27, 30
app, 145, 255
app-0, 229
app-associativity, 254
app-cancel, 265
app-head-tail, 254
app-nat-rangep, 254
append-len, 311
asl, 27
asl, 146, 327
asl-0, 232
asl-asl, 232
asl-beq, 283
asl-bmi, 289
asl-bvs, 287
asl-c, 184, 328
asl-c-bitp, 272
asl-cvznx, 185
asl-effect, 185
asl-int, 230
asl-int-crock1, 230
asl-int-crock2, 230
asl-n, 185, 328
asl-n-bitp, 272
asl-nat-rangep, 232
asl-v, 184, 328
asl-v-bitp, 272
asl-x, 185
asl-z, 185, 328
asl-z-bitp, 272
asr, 27
asr, 146, 327
asr-0, 232
asr-asr, 232
asr-beq, 283
asr-bmi, 289
asr-c, 185, 328
asr-c-bitp, 272
asr-cvznx, 186
asr-effect, 186
asr-int, 231
asr-int-crock, 231, 232
asr-n, 185, 328
asr-n-bitp, 272
asr-nat-rangep, 232
asr-x, 185
asr-z, 185, 328
asr-z-bitp, 272

b, 28, 30
b, 141
b-and, 26, 33, 34
b-and, 142
b-eor, 26
b-eor, 143
b-equal, 143
b-nand, 143
b-nor, 143

b-not, 26, 33, 34
b-not, 142
b-or, 26, 33, 34
b-or, 143
b0, 34
b0, 142
b0p, 26
b0p, 142
b1, 34
b1, 142
b1p, 142
bagdiff, 116
bagint, 116
bcar, 25, 26
bcar, 143, 254
bcar-1, 253
bcar-2, 253
bcar-app, 254
bcar-head, 253
bcar-lessp, 254
bcar-nonnumberp, 253
bcar-replace, 254
bcc-group, 207
bcc-group-h, 331
bcc-ra-sr, 207
bcdr, 25, 26
bcdr, 143, 254
bcdr-1, 253
bcdr-2, 253
bcdr-lessp, 254
bcdr-nonnumberp, 253
bchg-addr-modep, 213
bchg-effect, 213
bchg-ins, 213
bclr-addr-modep, 214
bclr-effect, 214
bclr-ins, 214
bcs, 44
bcs, 206, 285
bd-sz, 162
beq, 44
beq, 206, 284
Berkeley Unix C string library, ii, 90, 91, 107
between-ileq, 87
between-ileq, 278, 326
between-ileq-la, 326
bf-subgroup, 189
bge, 44
bge, 206, 290
bge-v0, 289
bgt, 45
bgt, 206, 291
bhi, 206, 279
Big Endian, 153
Binary Search, 370
binary tree, 149
bit, 224
bit-group, 218
bit-group-h, 332
bit-ins, 215
bitn, 25, 26, 39
bitn, 144, 255
bitn-0, 255
bitn-0-1, 255
bitn-app, 255
bitn-bitp, 270
bitn-head, 255
bitn-lognot, 255
bitn-replace, 255
bitn-tail, 39
bitn-tail, 255
bitp, 142
bitp-fix-bit, 269
bits, 25
bits, 144
ble, 206, 291
ble-bgt, 291
bls, 44
bls, 206, 279
bls-bhi, 279
blt, 206, 290
blt-bge, 290
blt-v0, 289
bmi, 44
bmi, 206, 291
boolean, 123
Boyer-Moore Majority Voting, 47, 413
Boyer-Moore Theorem Proving System, ii, 1, 19
branch-cc, 206
branch0, 148
branch1, 149
bridge-to-subbagp-implies-plus-tree-geq, 120
bs-pc, 164
bs-register, 163
bsearch, 59, 60
bsearch, 372
bsearch-code, 59
bsearch-code, 372
bsearch-correctness, 60
bsearch-correctness, 377
bsearch-crock, 375
bsearch-found, 60
bsearch-found, 379
bsearch-induct, 373
bsearch-not-found, 60
bsearch-not-found, 380

bsearch-s-s0, 374
bsearch-s-s0-mem, 374
bsearch-s-s0-rfile, 374
bsearch-s-s0p, 374
bsearch-s0-s0-1, 375
bsearch-s0-s0-2, 376
bsearch-s0-s0-rfile1, 376
bsearch-s0-sn, 376
bsearch-s0-sn-base1, 374
bsearch-s0-sn-base2, 375
bsearch-s0-sn-rfile, 377
bsearch-s0-sn-rfile-base1, 374
bsearch-s0p, 373, 376
bsearch-statep, 58, 60
bsearch-statep, 373, 376
bsearch-t, 59–61
bsearch-t, 373, 377
bsearch-t-ubound, 61
bsearch-t-ubound, 380
bsearch-t-ubound-la, 380
bsearch1, 372
bsearch1-found, 378
bsearch1-not-found, 379
bsearch1-not-found-1, 378
bsearch1-not-found-2, 379
bsearch1-not-found-2-lemma, 379
bsearch1-s0-s0-rfile2, 376
bsearch1-s0-sn-rfile-base2, 375
bsearch1-t, 59
bsearch1-t, 372
bsearch1-t-0, 380
bsearch1-t-crock, 380
bsearch1-t-ubound, 380
bset-addr-modep, 214
bset-effect, 214
bset-ins, 214
bsr-ins, 207
bsz, 141
btst-addr-modep, 214
btst-ins, 214
bv-adder, 224, 226, 227
bv-adder-bridge, 227
bv-adder-len, 225
bv-adder-listp, 225
bv-adder-nat, 227
bv-bitn, 224
bv-bitn-bitn, 227
bv-bitn-not, 226
bv-head, 224
bv-head-len, 225
bv-head-listp, 225
bv-head-nat, 226
bv-len, 224
bv-len-listp, 225
bv-mbit, 224
bv-mbit-bitn, 227
bv-not, 224
bv-not-lognot, 227
bv-sized-to-nat, 225
bv-sized-to-nat-head, 226
bv-sized-to-nat-to-bv-sized, 226
bv-tail, 224
bv-to-lst, 311
bv-to-lst-len, 311
bv-to-lst-proper-lstp, 311
bv-to-nat, 225
bv-to-nat-rangep, 226
bv-to-nat-to-bv, 226
bv-to-nat-to-bv-sized, 226
bvs, 44
bvs, 203, 287
bxxx-num, 213
bxxx-opd, 213
bxxx-oplen, 213
byte-an-direct-modep, 33
byte-an-direct-modep, 166
byte-read, 30
byte-read, 153, 266
byte-read-nat-rangep, 259
byte-read-write, 266
byte-read-write-mem, 268
byte-read-write-mem-end, 268, 269
byte-read-write-mem-lemma, 269
byte-read=read-mem-1, 310
byte-readp, 152, 266
byte-readp-ram0, 296
byte-readp-ram1, 296
byte-readp-ram2, 297
byte-readp-ram3, 297
byte-readp-rom0, 294
byte-readp-rom1, 294
byte-readp-rom2, 294
byte-readp-rom3, 295
byte-write, 30
byte-write, 155, 266
byte-write-app, 266
byte-write-maintain-byte-readp, 266
byte-write-maintain-byte-writep, 266
byte-write-maintain-pc-byte-readp, 266
byte-write-maintain-pc-read-memp, 267
byte-write-maintain-ram-addrp, 292
byte-write-maintain-read-memp, 267
byte-write-maintain-rom-addrp, 291
byte-write-maintain-write-memp, 267
byte-write-mcode-addrp, 300
byte-write-write, 266
byte-write-write-la, 266
byte-write=write-mem-1, 310

byte-writep, 155, 266
byte-writep->readp, 265
byte-writep-ram0, 295
byte-writep-ram1, 295
byte-writep-ram2, 295
byte-writep-ram3, 295
byte-writep=write-memp-1, 310

cadr-eval, 120
cancel-add-neg, 276
cancel-difference-plus, 121
cancel-equal-add, 39
cancel-equal-add, 276
cancel-equal-plus, 121
cancel-equal-times, 128
cancel-lessp-plus, 122
cancel-lessp-times, 127
cand-cnt, 71–73
cand-cnt, 416
cand-cnt-0, 428
cand-cnt-1, 428
cand-cnt-induct, 418
cand-cnt-lemma, 429
cand-cnt-rec, 429, 430
cand-cnt-t, 70, 71, 73
cand-cnt-t, 417
cand-cnt-t-0, 431
cand-cnt-t-1, 431
cand-cnt-t-ubound, 73
cand-cnt-t-ubound, 431
ccr-c, 156, 260
ccr-n, 156, 261
ccr-v, 156, 260
ccr-x, 156, 261
ccr-z, 156, 260
cd-divides-gcd, 437
cdr-mcdr, 311
clr-addr-modep, 192
clr-cvznx, 192
clr-effect, 192
clr-ins, 192
clr-subgroup, 193
cmp-addr-modep, 210
cmp-cvznx, 210
cmp-group, 212
cmp-group-h, 331
cmp-ins, 210
cmpa-addr-modep, 210
cmpa-ins, 210
cmpi-addr-modep, 218
cmpi-ins, 218
cmpi-mapping, 218
cmpi-subgroup, 218
cmpm-ins, 211
cmpm-mapping, 211
comp, 80, 81
comp-correctness, 80
comp-loadp, 81
comp-statep, 80, 81
comp-t, 80, 81
compiler verification, 6
cond-field, 155
cons-key-lst, 220
control-addr-modep, 165
Berkeley Unix copyright notice, 91
correctness-of-cancel-add-neg, 277
correctness-of-cancel-difference-plus, 122
correctness-of-cancel-equal-add, 39
correctness-of-cancel-equal-add, 276
correctness-of-cancel-equal-plus, 121
correctness-of-cancel-equal-times, 129
correctness-of-cancel-lessp-plus, 122
correctness-of-cancel-lessp-times, 127
count-lst, 66
count-lst, 411
count-lst-0, 411
count-lst-eq, 412
count-lst-put-1, 411
count-lst-qpart-aux, 412
count-lst-qsort, 66
count-lst-qsort, 413
count-lst-qsort-la, 412
count-lst-swap, 412
count-lst-swap-1, 411
count-lst-swap-rec, 412
count-swapii, 411
count-transwap, 411
count-transwap-0, 411
current-ins, 32
current-ins, 220
cvznx, 29, 33
cvznx, 156, 261

d, 154
d-bit-subgroup, 215
d-mapping, 33
d-mapping, 166
d2-7a2-5p, 49, 52, 56, 60, 65, 72, 77, 80, 82–84, 87, 89, 102, 106
d2-7a2-5p, 336
d2-7a3-5p, 337
d3-7a2-5p, 336
d4-7a2-5p, 337
d4-7a4-5p, 337
d4-7a5p, 337
d5-7a2-5p, 337
d5-7a4-5p, 337
d6-7a2-5p, 337

d_rn, 32, 33
data-addr-modep, 165
dbcc-ins, 208
dbcc-loop, 208
delete, 115
delete-commutativity, 116
delete-non-member, 116
dh, 200
difference-0, 118
difference-app-1s, 231
difference-cancel-1, 351
difference-cancel-4294967292, 352
difference-cancel-4294967295, 352
difference-difference1, 118
difference-difference2, 118
difference-equal-cancel-0, 119
difference-equal-cancel-1, 119
difference-int-rangep, 237
difference-is-1, 351
difference-lessp, 118
difference-lessp-cancel, 119
difference-lessp-cancel-1, 352
difference-lessp-cancel-2, 353
difference-lessp1, 119
difference-nat-rangep, 258
difference-plus-cancel-add1, 118
difference-plus-cancel0, 118
difference-plus-cancel1, 118
difference-plus1, 118
difference-plus2, 118
difference-sub1, 118
difference-sub1-sub1, 118
difference-x-x, 118
difference=0, 119
disjoint, 43, 53, 56, 59, 60, 63, 65, 69, 72, 77, 82, 83, 85, 87–89, 102, 103, 105, 106
disjoint, 223, 329
disjoint-0, 301, 303
disjoint-1, 301, 304
disjoint-1-int, 349
disjoint-1-uint, 349
disjoint-10, 302, 305
disjoint-11, 303, 305, 307
disjoint-11-asl, 307
disjoint-12, 303, 305
disjoint-13, 303, 305
disjoint-14, 303, 305
disjoint-15, 303, 305
disjoint-2, 301, 304, 306, 349, 350
disjoint-2-asl, 306
disjoint-2-int, 349
disjoint-2-uint, 350
disjoint-3, 302, 304, 306
disjoint-3-asl, 306
disjoint-3-int, 349
disjoint-3-uint, 350
disjoint-4, 302, 304
disjoint-5, 302, 304
disjoint-5-uint, 350
disjoint-6, 302, 304
disjoint-6-uint, 350
disjoint-7, 302, 304
disjoint-7-uint, 350
disjoint-8, 302, 304
disjoint-9, 302, 304, 306
disjoint-9-asl, 306
disjoint-commutativity, 303, 322
disjoint-deduction0, 307
disjoint-deduction1, 307
disjoint-deduction2, 308
disjoint-deduction3, 308
disjoint-head, 300
disjoint-la0, 301, 329
disjoint-la1, 301, 329
disjoint-la2, 301, 329
disjoint-la3, 301, 329
disjoint-la4, 447
disjoint-leq, 315, 322
disjoint-leq-uint, 322, 477
disjoint-leq1, 315, 322
disjoint-leq1-uint, 322, 477
disjoint-x-x, 307
disjoint0, 223, 329
disjoint0-deduction0, 307
disjoint0-deduction1, 307
disjoint0-deduction2, 307
disjoint0-head, 300
disjoint0-la0, 300, 329
disjoint0-la1, 300, 329
disjoint0-la2, 301, 329
disjoint0-leq, 315, 322
disjoint0-x-x, 307
div, 179, 199
divs, 178, 198, 243
divs-beq, 282
divs-cvznx, 178
divs-n, 178, 328
divs-n-bitp, 271
divs-overflow, 243
divs-v, 178, 243
divs-z, 178, 328
divs-z-bitp, 271
divsl, 198
divu, 179, 199
divu-beq, 282
divu-cvznx, 179
divu-n, 179, 328
divu-n-bitp, 271

divu-overflow, 243
divu-v, 179, 243
divu-z, 179, 328
divu-z-bitp, 271
divul, 199
dl, 200
dn-direct, 159
dn-direct-modep, 166
dq, 199
dr, 199

effec-addr, 31
effec-addr, 165
effec-addr-mem, 330
embedded assembly code, 87
Endian, 153
eof, 338
eor, 211
eor-cvznx, 211
eor-effect, 211
eor-ins, 211
eor-mapping, 211
eor-n, 211, 328
eor-n-bitp, 273
eor-z, 211, 328
eor-z-bitp, 273
eori-ins, 217
eori-subgroup, 218
eori-to-ccr-ins, 217
equal*, 78
equal*, 336
equal*-reflex, 336
equal-1-eval, 128
equal-1-times-tree-delete, 128
equal-iff, 123, 124
eval, 120, 126–128, 275
evenp, 32, 51, 55, 59, 63, 69, 76, 81, 83, 84, 87, 88, 102, 104
evenp, 220, 228
execute-ins, 32
execute-ins, 219
exg-arar-ins, 175
exg-drar-ins, 176
exg-drdr-ins, 175
exp, 20, 24–27, 38, 41, 53, 57, 61, 63
exp, 129
exp-exp, 130
exp-lessp, 130
exp-of-0, 129
exp-of-1, 129
exp-of-2-0, 130
exp-of-2-1, 130
exp-plus, 129
exp-times, 130
exp2-leq, 252
exp2-lessp, 233
exp2-lessp-crock, 241
ext, 26
ext, 145, 260
ext-0, 259
ext-beq-int-0, 283
ext-beq-int-1, 284
ext-beq-uint, 283
ext-bmi, 289
ext-cvznx, 191
ext-effect, 191
ext-equal, 265
ext-equal-0, 265
ext-ins, 195
ext-int, 233
ext-lemma, 259
ext-n, 191, 328
ext-n-bitp, 273
ext-nat-rangep, 260
ext-subgroup, 197
ext-z, 191, 328
ext-z-bitp, 273
extb-ins, 191

f1, 49
f2, 49
fix-bit, 25, 29
fix-bit, 142, 327
fix-bit-bitp, 270
fix-bv, 224
fix-bv-adder, 227
fix-int, 146
fix-int-integerp, 238
fn, 49
fn*-correctness, 442
fn-name, 21, 22
foo, 86, 87, 89
foo, 439, 450
foo-code, 86–88
foo-code, 439, 450
foo-correctness, 87, 89
foo-correctness, 440, 450
foo-s-sn, 440
foo-snp, 440
foo-statep, 86–89
foo-statep, 439, 450
foo-t, 86, 87, 89
foo-t, 439, 450
functional parameter, 79, 440

gcd, 52, 53, 76, 78
gcd, 356
gcd-addr, 75, 76, 78

gcd-code, 51
gcd-code, 355
gcd-correctness, 52
gcd-correctness, 359
gcd-example, 221
gcd-induct, 356
gcd-is-cd, 53
gcd-is-cd, 360
gcd-loadp, 76
gcd-nonzero, 436
gcd-s-s0, 357
gcd-s-s0-mem, 357
gcd-s-s0-rfile, 357
gcd-s0-s0-1, 358
gcd-s0-s0-2, 358
gcd-s0-s0-rfile-1, 359
gcd-s0-s0-rfile-2, 359
gcd-s0-sn, 359
gcd-s0-sn-base-1, 357
gcd-s0-sn-base-2, 357
gcd-s0-sn-base-rfile-1, 357
gcd-s0-sn-base-rfile-2, 358
gcd-s0-sn-rfile, 359
gcd-s0p, 356
gcd-statep, 51–53
gcd-statep, 356
gcd-t, 52–54, 76
gcd-t, 356, 360
gcd-t-shrink-1, 360
gcd-t-shrink-2, 361
gcd-t-ub, 54
gcd-t-ub, 361
gcd-t-ubound, 53
gcd-t-ubound, 361
gcd-t1, 52
gcd-t1, 356
gcd-t1-t2, 361
gcd-t2, 361
gcd-t2-ub, 361
gcd-the-greatest, 53
gcd-the-greatest, 360
gcd3, 77
gcd3, 432
gcd3-addr, 75, 76, 78
gcd3-code, 75, 76
gcd3-code, 432
gcd3-correctness, 77
gcd3-correctness, 437
gcd3-is-cd, 77
gcd3-is-cd, 437
gcd3-load, 76
gcd3-load, 432
gcd3-loadp, 76, 78
gcd3-s-s0, 434
gcd3-s-s0-mem, 434
gcd3-s-s0-rfile, 434
gcd3-s0-s1, 435
gcd3-s0-s1-mem, 435
gcd3-s0-s1-rfile, 435
gcd3-s0p, 77, 78
gcd3-s0p, 433
gcd3-s1-s2, 435
gcd3-s1-s2-mem, 435
gcd3-s1-s2-rfile, 435
gcd3-s1p, 77, 78
gcd3-s1p, 433
gcd3-s2-s3, 436
gcd3-s2-s3-mem, 436
gcd3-s2-s3-rfile, 436
gcd3-s2p, 434
gcd3-s3-sn, 436
gcd3-s3-sn-mem, 437
gcd3-s3-sn-rfile, 436
gcd3-s3p, 434
gcd3-statep, 76–78
gcd3-statep, 433
gcd3-t, 76, 77
gcd3-t, 433, 437
gcd3-t0, 76, 77
gcd3-t0, 432, 437
gcd3-t1, 76, 77
gcd3-t1, 432, 437
gcd3-t2, 76
gcd3-t2, 433, 437
gcd3-t3, 433, 437
gcd3-t4, 76
gcd3-t4, 433, 437
gcd3-the-greatest, 77
gcd3-the-greatest, 438
get-lst, 310
get-lst-cdr, 312
get-lst-int, 335
get-lst-integerp, 336
get-lst-mcar, 312
get-lst-mcdr, 312
get-lst-of-chrp, 346
get-nth, 29, 59, 60, 64–66, 70, 71, 103
get-nth, 151, 338
get-nth-0, 261, 676
get-nth-append, 312
get-nth-nil, 262
get-nth-plus-0, 679
get-nth-tail-lst, 298
get-put, 262
get-swap, 338
get-swap-1, 407
get-vals, 310
get-vals-0, 316, 317

get-vals-append, 313
get-vals-cdr, 312
get-vals-len, 311
get-vals-mcar, 312
get-vals-mcdr, 312
get-vals-proper-lstp, 311
get-vlst, 263, 264
get-vlst-member, 263, 264
get-vlst-readm-rn, 263
Gnu C compiler, ii, 2, 15, 47, 85, 90
Gnu debugger, 15
Greatest Common Divisor, 50, 354
gt, 83, 84
gt, 441, 442
gt-code, 83, 84
gt-code, 441
gt-correctness, 83
gt-s-sn, 442
gt-statep, 83
gt-statep, 442
gt-t, 83, 84
gt-t, 441, 442

h-invariant, 330
halt, 31, 33
halt, 156
hardware verification, 7
head, 25, 27–29, 42, 43
head, 143, 254
head-0, 228
head-app, 253
head-app-cancel, 265
head-app-head-tail, 254
head-byte-pc-read, 257
head-byte-read, 257
head-byte-readp, 257
head-byte-write, 257
head-byte-writep, 257
head-ext, 260
head-head, 251
head-int-crock, 241, 242
head-lemma, 229
head-leq, 229
head-lessp, 228
head-mem-ilst, 309
head-mem-lst, 309
head-nat-rangep, 259
head-of-0, 228
head-pc-byte-readp, 257
head-pc-read, 256
head-pc-read-mem, 257
head-pc-read-memp, 257
head-pc-readp, 256
head-plus-cancel, 251
head-plus-cancel0, 251
head-plus-head, 251
head-read, 256
head-read-mem, 257
head-read-memp, 257
head-read-rn, 262
head-readm-mem, 258
head-readp, 256
head-recursion, 265
head-replace, 252
head-sub1-lessp, 269
head-write, 257
head-write-mem, 257
head-write-memp, 257
head-writep, 256

i-data, 180
i-is, 162
idata-modep, 166
idifference, 44
idifference, 147, 351, 383
idifference-equal-0, 676
idifference-int-rangep, 351
idifference-int-rangep1, 235
idifference-int-rangep2, 235
idifference-integerp, 238
idifference-negativep, 239
idifference-numberp, 676
idifference-x-x, 239
ileq, 66
ileq, 147
ilessp, 44, 45, 55, 59, 64, 65, 83
ilessp, 146, 364, 372, 383, 442
ilessp-crock1, 234, 291
ilessp-crock2, 235, 291
ilessp-entails-ileq, 239
ilessp-lessp, 372, 383, 676
ilessp-reflex, 239
ilessp-trans, 408
immediate, 164
immediate-mem, 330
index-j, 607
index-j-la, 609
index-n, 80
index-n, 293, 308
index-n-0, 308
index-n-deduction0, 308
index-n-deduction1, 308
index-n-deduction2, 308
index-n-x-x, 308
index-register, 160
index-rn, 160
ineg, 147
ineg-integerp, 238

ineg-iplus, 239
initial-j-int-rangep, 365
int-equal, 378
int-rangep, 40, 44, 59
int-rangep, 148, 243
int-rangep-la, 235
int-rangep-of-0, 237
int-rangep-plus-1, 392
int-to-nat, 40
int-to-nat, 148, 237
int-to-nat-0, 236
int-to-nat-rangep, 236
int-to-nat-to-int, 236
int-to-nat=0, 237, 284
integerp, 60
integerp, 146, 240, 364
integerp-fix-int, 238
integerp-minus0, 238
iplus, 40, 41
iplus, 147, 240, 364, 375, 383
iplus-0, 239
iplus-1--1, 239
iplus-associativity, 239
iplus-commutativity, 238
iplus-commutativity1, 239
iplus-int-rangep1, 235
iplus-int-rangep2, 235
iplus-integerp, 238
iplus-with-carry-negativep, 233
iplus-with-carry-non-negativep, 234
iquot, 177, 327
iquot-int, 242
iquot-nat-rangep, 259
iquotient, 147, 240, 364, 375, 383
iquotient-int-rangep, 237
iquotient-integerp, 238
iquotient-wrt--1, 240
iquotient-wrt-1, 240
iquotient=0, 282, 284
ir-scaled, 161
iread-an, 335
iread-dn, 53, 55, 57, 60, 72, 77, 78, 80, 82, 84, 85, 87, 89
iread-dn, 335
iread-mem, 51, 52, 63, 70, 76, 78, 81, 83, 84, 87–89
iread-mem, 335
iread-mem-get0, 314
irem, 178, 327
irem-int, 242
irem-nat-rangep, 259
iremainder, 147, 240
iremainder-int-rangep, 238
iremainder-integerp, 238
iremainder-wrt--1, 240
iremainder-wrt-1, 240
isqrt, 56, 57
isqrt, 363
isqrt->isqrt1, 369
isqrt-code, 55
isqrt-code, 363
isqrt-correctness, 56
isqrt-correctness, 368
isqrt-induct, 364
isqrt-logic-correctness, 57
isqrt-logic-correctness, 369
isqrt-no-overflow, 364
isqrt-s-exit, 365
isqrt-s-exit-mem, 365
isqrt-s-exit-rfile, 365
isqrt-s-s0, 365
isqrt-s-s0-mem, 366
isqrt-s-s0-rfile, 365
isqrt-s0-exit, 367
isqrt-s0-exit-base, 366
isqrt-s0-exit-mem, 367
isqrt-s0-exit-rfile, 367
isqrt-s0-s0, 366
isqrt-s0-s0-mem, 367
isqrt-s0-s0-rfile, 366
isqrt-s0p, 364, 367
isqrt-s0p-j, 367
isqrt-statep, 55, 56
isqrt-statep, 364, 367
isqrt-t, 55–57
isqrt-t, 363, 368
isqrt-t->isqrt1-t, 370
isqrt-t-ubound, 57
isqrt-t-ubound, 370
isqrt-t-ubound-la, 370
isqrt-t1-ubound, 370
isqrt1, 56
isqrt1, 363
isqrt1-lower-bound, 368
isqrt1-s0-exit-mem-base, 366
isqrt1-s0-exit-rfile-base, 366
isqrt1-t, 55, 56
isqrt1-t, 363
isqrt1-t-la-0, 369
isqrt1-t-la-1, 369
isqrt1-t-la-2, 369
isqrt1-upper-bound, 368
itimes, 147, 240
itimes-0, 239
itimes-associativity, 239
itimes-commutativity, 239
itimes-equal-0, 239
itimes-equal-cancellation, 240

itimes-integerp, 238
itimes-sign, 240
izerop, 146

j-int-rangep, 364
j-nonzerop, 364
jmp-addr-modep, 204
jmp-ins, 204
jmp-mapping, 204
jsr-addr-modep, 204
jsr-ins, 204

key-field, 220

l, 27, 28, 32
l, 141
lea-addr-modep, 191
lea-ins, 191
lea-subgroup, 192
lemma-name, 21
len, 60, 84
len, 152
leq-trans, 378
lessp-1-times-tree-delete, 126, 128
lessp-1-times-tree-delete-end, 126
lessp-app-1s, 231
lessp-cancel-4294967292, 352
lessp-cancel-4294967295, 352
lessp-of-1, 117
lessp-plus-exp2, 137
lessp-plus-times-exp2, 137
lessp-read-mem-1, 346
lessp-slen-mcdr, 347
lessp-slen-mcdr-0, 347
lessp-sub1, 117
lessp-times-exp-1s, 231
lessp-times-lessp, 360
link, 202
link-mapping, 202
linked-a6, 335
linked-rts-addr, 335
listp-eval, 121
load-lst-mem, 221
log, 24, 38, 54
log, 139
log-equal-0, 139
log-exp, 139
log-leq, 140
log-mono, 370
log-of-0, 139
log-of-1, 139
log-quotient-exp, 139
log-times-exp, 139
log-times-exp-1, 139
logand, 26
logand, 144
logand-beq-uint, 281
logand-commutativity, 255
logand-eor-beq-uint, 281
logand-logeor, 256
logand-logor, 256
logand-nat-rangep, 258
logand-or-beq-uint, 281
logand-uint, 256
logand-uint-la, 256
logeor, 26
logeor, 144
logeor-beq-int-0, 280
logeor-beq-int-1, 280
logeor-beq-uint, 280
logeor-commutativity, 256
logeor-equal-0, 255
logeor-nat-rangep, 259
lognot, 26
lognot, 144, 228
lognot-0, 228
lognot-cancel, 228
lognot-int, 228
lognot-lognot, 228
lognot-nat-rangep, 228
logor, 26
logor, 144
logor-beq-int-0, 280
logor-beq-int-1, 280
logor-beq-uint, 280
logor-commutativity, 256
logor-equal-0, 255
logor-nat-rangep, 258
long-read, 154
long-readp, 153
lsl, 27
lsl, 145, 327
lsl-0, 232
lsl-1-bcs, 284
lsl-beq, 283
lsl-c, 182, 328
lsl-c-0, 284
lsl-c-bitp, 272
lsl-cvznx, 183
lsl-effect, 183
lsl-lsl, 232
lsl-n, 183, 328
lsl-n-bitp, 272
lsl-nat-rangep, 232
lsl-uint, 229
lsl-x, 183
lsl-z, 182, 328
lsl-z-bitp, 272

lsr, 27
lsr, 146, 327
lsr-0, 232
lsr-1-bcs, 285
lsr-beq, 283
lsr-c, 183, 328
lsr-c-0, 285
lsr-c-bitp, 272
lsr-cvznx, 183
lsr-effect, 184
lsr-lsr, 232
lsr-n, 183, 328
lsr-n-bitp, 272
lsr-nat-rangep, 232
lsr-uint, 229
lsr-x, 183
lsr-z, 183, 328
lsr-z-bitp, 272
lst-eq, 410
lst-int, 335
lst-integerp, 60
lst-integerp, 335
lst-numberp, 298, 299
lst-numberp0, 298
lst-of-chrp, 106
lst-of-chrp, 346
lst-to-bv, 311
lstcpy, 323
lstcpy-0, 323
lstcpy-add1, 324
lstcpy-cpy1, 324, 325
lstcpy-get-1, 694
lstcpy-get-2, 694
lstcpy-lstcpy, 324
lstcpy-lstcpy-0, 693
lstcpy-lstcpy-1, 693
lstcpy-nonnumberp, 693
lstcpy-put-nth, 324
lstcpy1, 324
lstmcpy, 324
lstmcpy-cpy, 324
lsz, 141

m-mapping, 167
main, 368
main-trick, 368
make-bt, 149
make-map, 220
map-update, 220
mapping, 167
mapping-h, 330
max-code, 81, 84
max-code, 444
max-comp, 81, 82
max-correctness, 82
max-correctness, 447
max-disjoint, 82
max-disjointness, 445
max-fn*, 444
max-gt, 84, 85
max-gt, 448
max-gt-correctness, 84
max-gt-correctness, 448
max-gt-statep, 84
max-gt-statep, 448
max-gt-statep-s0, 448
max-gt-t, 84
max-gt-t, 448
max-s-s0, 445
max-s-s0-else, 445
max-s-s0-mem, 445
max-s-s0-rfile, 445
max-s0-s1, 445
max-s0-s1-else, 446
max-s0-s1-mem, 446
max-s0-s1-rfile, 446
max-s0p, 444
max-s1-sn-1, 446
max-s1-sn-2, 447
max-s1-sn-rfile-1, 446
max-s1-sn-rfile-2, 447
max-s1p, 445
max-sp, 81
max-sp, 444
max-state, 81
max-state, 444
max-statep, 81, 82
max-t, 81, 82
max-t, 444
max-t0, 81, 84
max-t0, 444, 447
mbit, 25, 33, 34
mbit, 144
mbit-means-lessp, 233, 278
mbit-means-negativep, 285, 291
mc-ccr, 28
mc-ccr, 152, 261
mc-ccr-rangep, 261
mc-ccr-rewrite, 261
mc-haltp, 11, 31, 32
mc-haltp, 152
mc-instate, 31, 32
mc-instate, 166
mc-instate-mem, 330
mc-mem, 28, 31, 32, 45, 49, 51, 53, 55, 57, 59, 60, 63, 65, 69, 70, 72, 76–78, 80–85, 87, 89, 102, 103, 105, 106
mc-mem, 152, 261

mc-mem-rewrite, 261
mc-pc, 27, 32, 48, 51, 52, 55, 56, 59, 60, 63, 65, 69, 72, 76–78, 80–84, 87–89, 102, 105, 106
mc-pc, 152, 261
mc-pc-rangep, 261
mc-pc-rewrite, 261
mc-rfile, 27, 49, 52, 53, 56, 60, 72, 80–85, 87, 89, 102, 103, 106
mc-rfile, 152, 261
mc-rfile-rewrite, 261
mc-state, 11, 27
mc-state, 152, 261
mc-status, 27, 48, 51, 52, 55, 56, 59, 60, 63, 65, 69, 72, 76–78, 80–84, 87–89, 102, 105, 106
mc-status, 152, 261
mc-status-rewrite, 261
MC68020, ii
mcar, 311
mcar-mcar, 312
mcar-nth, 312
mcdr, 106
mcdr, 311
mcdr-listp-len, 311
mcdr-mcdr, 312
mcdr-nth, 312
mcode-addrp, 51, 55, 59, 63, 69, 76, 81, 83, 84, 87, 88, 102, 105
mcode-addrp, 292, 334
mean-bounds, 372, 391
mean-difference-1, 351
mean-difference-1-1, 369
mean-difference-2, 352
mean-lessp, 337
mean-lessp-1, 369
mean-lessp-lemma, 337
mem-asl-effect, 186
mem-asl-ins, 186
mem-asr-effect, 186
mem-asr-ins, 186
mem-ilst, 59, 63, 65, 69
mem-ilst, 308, 336
mem-ilst-int-rangep, 309
mem-ilst-integerp, 309
mem-ilst-lst-integerp, 336
mem-indirect, 161
mem-induct0, 267
mem-induct1, 297
mem-induct2, 313
mem-induct3, 316
mem-induct4, 316
mem-lsl-effect, 184
mem-lsl-ins, 184
mem-lsr-effect, 184
mem-lsr-ins, 184
mem-lst, 102, 103, 105
mem-lst, 308, 336
mem-lst-get-lst, 313
mem-lst-get-lst0, 313
mem-lst-get-vals, 314
mem-lst-get-vals0, 313
mem-lst-integerp, 336
mem-lst-len, 322, 323
mem-lst-lessp, 309
mem-lst-mcar, 322
mem-lst-mcar-1, 322
mem-lst-mcar-2, 322
mem-lst-mcdr, 323
mem-lst-mcdr-0, 323
mem-lst-mcdr-1, 323
mem-lst-mcdr-uint, 323
mem-lst-mcdr-uint-1, 323
mem-lst-nat-rangep, 308
mem-lst-non-numberp, 309, 310
mem-lst-numberp, 309
mem-lst-of-chrp, 346
mem-lst-plus, 322
mem-lst-put-lst, 316
mem-lst-put-vals, 316
mem-lst-same, 309, 310
mem-postindex, 161
mem-preindex, 162
mem-rol-effect, 182
mem-rol-ins, 182
mem-ror-effect, 182
mem-ror-ins, 182
mem-roxl-effect, 188
mem-roxl-ins, 188
mem-roxr-effect, 188
mem-roxr-ins, 188
member-delete, 116
member-implies-numberp, 120
member-implies-plus-tree-greatereqp, 119
member1, 378
member1-member2, 379
member2, 379
member2-lemma, 379
member2-member, 379
memchr, 95, 338, 451
memchr*, 343, 457
memchr*-memchr1, 456
memchr-bounds, 348
memchr-code, 452
memchr-correctness, 456
memchr-induct, 452
memchr-non-zerop, 457
memchr-non-zerop-la, 456

memchr-s-s0, 454
memchr-s-s0-else, 454
memchr-s-s0-mem, 454
memchr-s-s0-rfile, 454
memchr-s-sn, 453
memchr-s-sn-mem, 453
memchr-s-sn-rfile, 453
memchr-s0-s0, 455
memchr-s0-s0-rfile, 455
memchr-s0-sn, 455
memchr-s0-sn-base1, 454
memchr-s0-sn-base2, 454
memchr-s0-sn-rfile, 456
memchr-s0-sn-rfile-base1, 454
memchr-s0-sn-rfile-base2, 455
memchr-s0p, 453
memchr-statep, 452
memchr-t, 452, 456
memchr-t1, 452
memchr-thm1, 681
memchr-thm2, 682
memchr-thm3, 682
memchr1, 338
memchr1-thm1, 681
memchr1-thm2, 681
memchr1-thm3, 682
memcmp, 93, 338, 457
memcmp-code, 458
memcmp-correctness, 463
memcmp-id, 683
memcmp-induct, 459
memcmp-j, 684
memcmp-j-1, 684
memcmp-j-2, 684
memcmp-s-s0, 460
memcmp-s-s0-else, 460
memcmp-s-s0-mem, 461
memcmp-s-s0-rfile, 460
memcmp-s-sn, 460
memcmp-s-sn-mem, 460
memcmp-s-sn-rfile, 460
memcmp-s0-s0, 462
memcmp-s0-s0-rfile, 462
memcmp-s0-sn, 462
memcmp-s0-sn-base1, 461
memcmp-s0-sn-base2, 461
memcmp-s0-sn-rfile, 462
memcmp-s0-sn-rfile-base1, 461
memcmp-s0-sn-rfile-base2, 461
memcmp-s0p, 459
memcmp-statep, 459
memcmp-t, 459, 463
memcmp-t1, 458
memcmp-thm1, 683
memcmp-thm2, 684
memcmp1, 338
memcmp1-id, 683
memcmp1-thm1, 683
memcpy, 92, 463, 468
memcpy-code, 467
memcpy-correctness, 468
memcpy-memmove-statep, 468
memcpy-statep, 468
memcpy-t, 468, 469
memmove, 102, 103
memmove, 90, 92, 98, 107, 342, 469
memmove-0, 342
memmove-1, 342
memmove-3, 342
memmove-4, 342
memmove-code, 102
memmove-code, 473
memmove-correctness, 102
memmove-correctness, 514
memmove-mmov1-lst, 694
memmove-s-s0-1, 478
memmove-s-s0-2, 479
memmove-s-s0-3, 480
memmove-s-s0-else-1, 479
memmove-s-s0-else-2, 480
memmove-s-s0-else-3, 481
memmove-s-s0-mem-1, 479
memmove-s-s0-mem-2, 480
memmove-s-s0-mem-3, 481
memmove-s-s0-rfile-1, 479
memmove-s-s0-rfile-2, 480
memmove-s-s0-rfile-3, 481
memmove-s-s1, 482
memmove-s-s1-else, 482
memmove-s-s1-mem, 482
memmove-s-s1-rfile, 482
memmove-s-s2, 483
memmove-s-s2-else, 483
memmove-s-s2-mem, 483
memmove-s-s2-rfile, 483
memmove-s-s3-1, 496
memmove-s-s3-2, 497
memmove-s-s3-3, 498
memmove-s-s3-else-1, 496
memmove-s-s3-else-2, 497
memmove-s-s3-else-3, 498
memmove-s-s3-mem-1, 497
memmove-s-s3-mem-2, 498
memmove-s-s3-mem-3, 499
memmove-s-s3-rfile-1, 496
memmove-s-s3-rfile-2, 498
memmove-s-s3-rfile-3, 499
memmove-s-s4, 499

memmove-s-s4-else, 500
memmove-s-s4-mem, 500
memmove-s-s4-rfile, 500
memmove-s-s5, 500
memmove-s-s5-else, 501
memmove-s-s5-mem, 501
memmove-s-s5-rfile, 501
memmove-s-sn-1, 477
memmove-s-sn-2, 478
memmove-s-sn-mem-1, 478
memmove-s-sn-mem-2, 478
memmove-s-sn-rfile-1, 477
memmove-s-sn-rfile-2, 478
memmove-s0-s0, 485
memmove-s0-s0-mem, 485
memmove-s0-s0-rfile, 485
memmove-s0-s1, 483
memmove-s0-s1-else, 484
memmove-s0-s1-mem, 484
memmove-s0-s1-rfile, 484
memmove-s0-s2, 484
memmove-s0-s2-else, 484
memmove-s0-s2-mem, 485
memmove-s0-s2-rfile, 485
memmove-s0-sn, 495
memmove-s0-sn-base-1, 492
memmove-s0-sn-base-2, 493
memmove-s0-sn-base-3, 494
memmove-s0-sn-mem, 495
memmove-s0-sn-mem-base-1, 493
memmove-s0-sn-mem-base-2, 493
memmove-s0-sn-mem-base-3, 494
memmove-s0-sn-rfile, 495
memmove-s0-sn-rfile-base-1, 492
memmove-s0-sn-rfile-base-2, 493
memmove-s0-sn-rfile-base-3, 494
memmove-s0-sn-t, 495
memmove-s0-sn-t0, 493
memmove-s0-sn-t1, 494
memmove-s0p, 474
memmove-s0p-info, 495, 496
memmove-s1-s1, 486
memmove-s1-s1-mem, 487
memmove-s1-s1-rfile, 487
memmove-s1-s2, 486
memmove-s1-s2-else, 486
memmove-s1-s2-mem, 486
memmove-s1-s2-rfile, 486
memmove-s1-sn, 491
memmove-s1-sn-base-1, 489
memmove-s1-sn-base-2, 490
memmove-s1-sn-induct, 491
memmove-s1-sn-mem, 492
memmove-s1-sn-mem-base-1, 490
memmove-s1-sn-mem-base-2, 491
memmove-s1-sn-rfile, 491
memmove-s1-sn-rfile-base-1, 490
memmove-s1-sn-rfile-base-2, 490
memmove-s1-sn-t, 491
memmove-s1-sn-t0, 490
memmove-s1p, 474
memmove-s1p-info, 491, 492
memmove-s2-s2, 488
memmove-s2-s2-mem, 488
memmove-s2-s2-rfile, 488
memmove-s2-sn, 488
memmove-s2-sn-base, 487
memmove-s2-sn-induct, 488
memmove-s2-sn-mem, 489
memmove-s2-sn-mem-base, 487
memmove-s2-sn-rfile, 489
memmove-s2-sn-rfile-base, 487
memmove-s2-sn-t, 488
memmove-s2p, 475
memmove-s2p-info, 488, 489
memmove-s3-s3, 503
memmove-s3-s3-mem, 503
memmove-s3-s3-rfile, 503
memmove-s3-s4, 501
memmove-s3-s4-else, 502
memmove-s3-s4-mem-base, 502
memmove-s3-s4-rfile-base, 502
memmove-s3-s5, 502
memmove-s3-s5-else, 502
memmove-s3-s5-mem, 503
memmove-s3-s5-rfile, 502
memmove-s3-sn, 513
memmove-s3-sn-base-1, 510
memmove-s3-sn-base-2, 511
memmove-s3-sn-base-3, 512
memmove-s3-sn-mem, 513
memmove-s3-sn-mem-base-1, 510
memmove-s3-sn-mem-base-2, 511
memmove-s3-sn-mem-base-3, 512
memmove-s3-sn-rfile, 513
memmove-s3-sn-rfile-base-1, 510
memmove-s3-sn-rfile-base-2, 511
memmove-s3-sn-rfile-base-3, 512
memmove-s3-sn-t, 513
memmove-s3-sn-t0, 511
memmove-s3-sn-t1, 512
memmove-s3p, 475
memmove-s3p-info, 513, 514
memmove-s4-s4, 504
memmove-s4-s4-mem, 505
memmove-s4-s4-rfile, 504
memmove-s4-s5, 503
memmove-s4-s5-else, 504

memmove-s4-s5-mem, 504
memmove-s4-s5-rfile, 504
memmove-s4-sn, 509
memmove-s4-sn-base-1, 507
memmove-s4-sn-base-2, 508
memmove-s4-sn-induct, 509
memmove-s4-sn-mem, 510
memmove-s4-sn-mem-base-1, 508
memmove-s4-sn-mem-base-2, 508
memmove-s4-sn-rfile, 509
memmove-s4-sn-rfile-base-1, 507
memmove-s4-sn-rfile-base-2, 508
memmove-s4-sn-t, 509
memmove-s4-sn-t0, 508
memmove-s4p, 476
memmove-s4p-info, 509, 510
memmove-s5-s5, 505
memmove-s5-s5-mem, 506
memmove-s5-s5-rfile, 506
memmove-s5-sn, 506
memmove-s5-sn-base, 505
memmove-s5-sn-induct, 506
memmove-s5-sn-mem, 507
memmove-s5-sn-mem-base, 505
memmove-s5-sn-rfile, 507
memmove-s5-sn-rfile-base, 505
memmove-s5-sn-t, 506
memmove-s5p, 477
memmove-s5p-info, 506, 507
memmove-statep, 102
memmove-statep, 474
memmove-t, 102
memmove-t, 514, 515
memmove-thm1, 103
memmove-thm1, 694
memory, 149
memory-addr-modep, 165
memory-shift-rotate, 188
memset, 98, 338, 515
memset-code, 516
memset-correctness, 520
memset-induct, 517
memset-s-s0, 518
memset-s-s0-else, 518
memset-s-s0-mem, 518
memset-s-s0-rfile, 518
memset-s-sn, 517
memset-s-sn-mem, 518
memset-s-sn-rfile, 518
memset-s0-s0, 519
memset-s0-s0-mem, 519
memset-s0-s0-rfile, 519
memset-s0-sn, 519
memset-s0-sn-base, 518
memset-s0-sn-mem, 520
memset-s0-sn-mem-base, 519
memset-s0-sn-rfile, 520
memset-s0-sn-rfile-base, 519
memset-s0p, 517
memset-statep, 517
memset-t, 516, 521
memset-t1, 516
memset-thm1, 681
memset-thm2, 681
memset1, 338
memset1-get-1, 681
memset1-thm1, 681
memset1-thm2, 681
Microcode verification, 7
microcontroller, 111
minus-integerp, 238
misc-group, 205
misc-group-h, 333
mjrty-cand, 71–73
mjrty-cand, 416
mjrty-cand-0, 428
mjrty-cand-1, 428
mjrty-cand-induct, 417
mjrty-cand-lemma, 429
mjrty-cand-rec, 429, 430
mjrty-cand-t, 70, 73
mjrty-cand-t, 417
mjrty-cand-t-0, 430
mjrty-cand-t-1, 430
mjrty-cand-t-ubound, 73
mjrty-cand-t-ubound, 430
mjrty-code, 69
mjrty-code, 415
mjrty-correctness, 72
mjrty-correctness, 427
mjrty-k, 70–72
mjrty-k, 416
mjrty-k-0, 428
mjrty-k-1, 428
mjrty-k-lemma, 428, 429
mjrty-k-rec, 429, 430
mjrty-lemma1, 429
mjrty-lemma2, 429
mjrty-lemma2-induct, 429
mjrty-p, 71–73
mjrty-p, 416
mjrty-s-s0, 419
mjrty-s-s0-mem, 419
mjrty-s-s0-rfile, 419
mjrty-s0-s0-1, 421
mjrty-s0-s0-2, 421
mjrty-s0-s0-3, 421
mjrty-s0-s0-rfile-1, 422

mjrty-s0-s0-rfile-2, 422
mjrty-s0-s0-rfile-3, 422
mjrty-s0-s1, 424
mjrty-s0-s1-base1, 423
mjrty-s0-s1-base2, 423
mjrty-s0-s1-rfile, 424
mjrty-s0-s1-rfile-base1, 424
mjrty-s0-s1-rfile-base2, 424
mjrty-s0-sn, 422
mjrty-s0-sn-base-1, 419
mjrty-s0-sn-base-2, 420
mjrty-s0-sn-rfile, 423
mjrty-s0-sn-rfile-base-1, 420
mjrty-s0-sn-rfile-base-2, 420
mjrty-s0p, 418
mjrty-s1-s1-1, 426
mjrty-s1-s1-2, 426
mjrty-s1-s1-rfile-1, 426
mjrty-s1-s1-rfile-2, 426
mjrty-s1-sn, 427
mjrty-s1-sn-1, 425
mjrty-s1-sn-2, 425
mjrty-s1-sn-rfile, 427
mjrty-s1-sn-rfile-1, 425
mjrty-s1-sn-rfile-2, 425
mjrty-s1p, 418
mjrty-sn-t, 70, 73
mjrty-sn-t, 417
mjrty-sn-t-ubound, 73
mjrty-sn-t-ubound, 430
mjrty-statep, 69, 72
mjrty-statep, 418
mjrty-statep-info, 427
mjrty-t, 70, 72, 73
mjrty-t, 417, 428
mjrty-t-crock, 430
mjrty-t-ubound, 73
mjrty-t-ubound, 431
mjrty-thm1, 73
mjrty-thm1, 430
mjrty-thm2, 73
mjrty-thm2, 430
mmov1-lst, 325
mmov1-lst-0, 325
mmov1-lst-cpy, 693
mmov1-lst-thm1, 694
mmov1-lst1, 325
mmov1-lst1-0, 325
mmov1-lst1-cpy, 693
mmovn-lst, 325
mmovn-lst-0, 325
mmovn-lst1, 325
mmovn-lst1-0, 325
mmovn-lst1-mmov1-lst, 694
mmovn-mmov1-lst, 693
mod-eq, 223
mod-eq-lemma, 223
mod32-eq, 223, 326
mod32-eq-deduction0, 325
mod32-eq-deduction1, 326
mod32-eq-deduction2, 326
mod32-eq-deduction3, 326
mode-signal, 28, 33
mode-signal, 142
modn-eq, 264
modn-eq-equal, 265
modn-lst, 321
modn-readm-rn, 321
move-addr-modep, 189
move-beq-ext, 279
move-beq-int-0, 279
move-beq-int-1, 279
move-beq-uint, 279
move-bgt, 290
move-ble, 290
move-bmi, 287
move-cvznx, 190
move-effect, 190
move-from-ccr-addr-modep, 192
move-from-ccr-effect, 192
move-from-ccr-ins, 192
move-group, 191
move-group-h, 332
move-h, 332
move-ins, 190
move-mapping, 190
move-n, 190, 328
move-n-bitp, 273
move-to-ccr-addr-modep, 193
move-to-ccr-ins, 194
move-z, 190, 328
move-z-bitp, 273
movea-addr-modep, 190
movea-ins, 190
movem-ea-rn-addr-modep, 200
movem-ea-rn-ins, 201
movem-ea-rn-subgroup, 201
movem-len, 196
movem-pre-rnlst, 196
movem-pre-rnlst-len, 332
movem-predec, 196
movem-rn-ea-addr-modep, 196
movem-rn-ea-ins, 196
movem-rnlst, 196
movem-rnlst-len, 332
movem-saved, 335
movep-ins, 213
movep-read, 212

movep-readp, 212
movep-to-mem, 212
movep-to-reg, 213
movep-write, 212
movep-write-h, 332
movep-writep, 212
moveq-ins, 209
movn-lst, 325
mul, 174, 176, 200
mul-div, 200
muls, 174, 175, 200, 242, 282, 286–288, 327
muls-crock, 241, 242
muls-cvznx, 175
muls-n, 175, 328
muls-n-bitp, 271
muls-nat-rangep, 259
muls-v, 175, 328
muls-v-bitp, 271
muls-z, 175, 328
muls-z-bitp, 271
mulu, 174, 175, 199, 240, 241, 281, 286, 327
mulu-cvznx, 174
mulu-n, 174, 328
mulu-n-bitp, 271
mulu-nat-rangep, 259
mulu-v, 174, 328
mulu-v-bitp, 271
mulu-z, 174, 328
mulu-z-bitp, 271

n-member, 223
name, 21
nat-plus-rangep, 258
nat-rangep, 40, 44, 45, 76, 105
nat-rangep, 148, 259
nat-rangep-0, 258
nat-rangep-la, 348, 349
nat-rangep-of-0, 258
nat-rangep-ub, 258
nat-to-bv, 225
nat-to-bv-sized, 225
nat-to-bv-sized-head, 225, 226
nat-to-bv-sized-la0, 225
nat-to-bv-sized-len, 225
nat-to-bv-sized-sized-to-nat, 226
nat-to-bv-sized-to-nat, 226
nat-to-bv-to-nat, 226
nat-to-int, 40, 41, 44, 45
nat-to-int, 148, 237
nat-to-int-0, 236
nat-to-int-integerp, 236
nat-to-int-rangep, 236
nat-to-int-to-nat, 236
nat-to-int=, 237, 291
nat-to-uint, 40, 44, 102
nat-to-uint, 148, 329
nat-to-uint-rangep, 236
nat-to-uint-to-nat, 236
neg, 223, 274
neg-add, 274
neg-addr-modep, 193
neg-cancel, 273
neg-head, 273
neg-ins, 193
neg-nat-rangep, 259
neg-neg, 273
neg-subgroup, 194
negativep-guts0, 238
negp, 146
negx-addr-modep, 193
negx-ins, 193
negx-subgroup, 193
nop-ins, 202
nop-subgroup, 203
not-addr-modep, 204
not-cvznx, 204
not-effect, 205
not-ins, 205
not-n, 204, 328
not-n-bitp, 273
not-subgroup, 205
not-z, 204, 328
not-z-bitp, 273
Nqthm, ii, 1, 2, 10, 19
null, 338
numberp-eval, 119, 120, 125, 275
numberp-integerp, 238

op-len, 155
op-sz, 31
op-sz, 151
opcode-field, 219
open-sublst-ileq, 406
open-sublst-ileq-la0, 406
open-sublst-ileq-la1, 406
open-sublst-ileq-la2, 406
open-sublst-ileq-la3, 406
open-sublst-ileq-la4, 406
open-sublst-ileq-la5, 406
open-sublst-ileq-la6, 406
operand, 32
operand, 158
opmode-field, 155
or-addr-modep1, 177
or-addr-modep2, 177
or-cvznx, 176
or-effect, 177
or-group, 180

or-group-h, 331
or-ins1, 177
or-ins2, 177
or-mapping, 177
or-n, 176, 328
or-n-bitp, 271
or-z, 176, 328
or-z-bitp, 271
orderedp, 60
orderedp, 378
orderedp-ordered, 378
orderedp1, 410
ori-addr-modep, 215
ori-ins, 215
ori-subgroup, 216
ori-to-ccr-ins, 215

p-disjoint, 80, 81
p-disjointness, 49, 80
p-disjointness, 442
p-f1, 48, 49
p-f2, 48, 49
p-fn, 48, 49
p-sem-eq, 49
p-statep, 17, 48, 49
p-t, 17, 48, 49
pc-byte-read, 30
pc-byte-read, 154, 266
pc-byte-read-mcode0, 297
pc-byte-read-mcode1, 297
pc-byte-read-mcode2, 298
pc-byte-read-mcode3, 298
pc-byte-read-nat-rangep, 259
pc-byte-read-write, 266
pc-byte-read-write-mem, 268
pc-byte-read=pc-read-mem-1, 310
pc-byte-readp, 153, 266
pc-byte-readp->byte-readp, 265
pc-byte-readp-rom0, 293
pc-byte-readp-rom1, 293
pc-byte-readp-rom2, 293
pc-byte-readp-rom3, 293
pc-disp, 163
pc-index, 164
pc-index-disp, 164
pc-index1, 164
pc-long-read, 154
pc-long-readp, 153
PC-Nqthm, 22
pc-odd-signal, 28, 32
pc-odd-signal, 142
pc-read, 151, 264
pc-read-mem, 30, 76, 105
pc-read-mem, 154, 328
pc-read-mem-byte-write, 268
pc-read-mem-mcode0, 298
pc-read-mem-mcode1, 299
pc-read-mem-mcode2, 299
pc-read-mem-mcode3, 299
pc-read-mem-nat-rangep, 267
pc-read-mem-write-mem, 268
pc-read-mem-writem-mem, 321
pc-read-memp, 153, 328
pc-read-memp->read-memp, 267
pc-read-memp-la0, 292, 329
pc-read-memp-la1, 292, 329
pc-read-memp-la2, 292, 329
pc-read-memp-la3, 292, 329
pc-read-memp-rom0, 293
pc-read-memp-rom1, 294
pc-read-memp-rom2, 294
pc-read-memp-rom3, 294
pc-read-write, 264
pc-readp, 150
pc-readp->readp, 264
pc-signal, 28, 32
pc-signal, 142
pc-word-read, 154
pc-word-readp, 32
pc-word-readp, 153
pea-addr-modep, 194
pea-ins, 194
pea-subgroup, 195
plus-0, 352
plus-add1, 117
plus-add1-1, 117, 327
plus-add1-sub1, 247
plus-associativity, 117
plus-commutativity, 117
plus-commutativity1, 117
plus-difference, 322, 323
plus-equal-0, 117
plus-equal-cancel, 117
plus-equal-cancel0, 117
plus-fringe, 119
plus-lessp-cancel-0, 117
plus-lessp-cancel-1, 117
plus-lessp-cancel-2, 122
plus-lessp-cancel-add1, 117
plus-numberp, 244, 245
plus-sub1-add1, 680
plus-times-equal, 264, 265
plus-times-lessp, 315
plus-times-sub1, 352, 477
plus-to-iplus, 233
plus-tree, 119
plus-tree-bagdiff, 120
plus-tree-delete, 120

plus-tree-plus-fringe, 120
plus2-times-sub1, 352
plus3-times-sub1, 352
post-effect, 159
postinc-modep, 166
pre-effect, 159
predec-modep, 166
processor specification, 7
proper-lstp, 311
protection, 149
push-sp, 158
put-commutativity, 324
put-commute, 402
put-get, 262
put-get-lst-is-cpy, 324
put-get-vals-is-cpy, 325
put-lst, 310
put-lst-int, 335
put-lst-of-chrp, 346
put-nth, 29
put-nth, 151, 338
put-nth-0, 261, 676
put-nth-len, 262
put-put, 262
put-vals, 310
put-vals-append, 313

q, 28
q, 141
qlast, 63–65, 67
qlast, 384
qlast-0, 397
qlast-aux, 65
qlast-aux, 384
qlast-aux-0, 396
qlast-aux-lb, 384
qlast-aux-swap, 407
qlast-aux-ub, 384
qlast-lb, 384
qlast-swap, 407
qlast-ub, 384
qpart, 63–65
qpart, 383, 384
qpart-aux, 64, 65
qpart-aux, 383
qpart-aux-ct, 394
qpart-aux-mem, 395
qpart-aux-qpartx, 405, 408
qpart-aux-rfile, 394
qpart-aux-swap, 408
qpart-aux-t, 63, 64
qpart-aux-t, 384
qpart-equal, 407
qpart-ilessp, 408
qpart-ilessp-closed-1, 409
qpart-ilessp-closed-2, 409
qpart-ilessp-closed-3, 410
qpart-ilessp-la1, 407
qpart-ilessp-la1-la, 406
qpart-ilessp-la2, 407
qpart-induct, 394
qpart-swap, 408
qpart-t, 64
qpart-t, 385, 395
qpartx, 405
qpartx-get-1, 405
qpartx-get-2, 405
qpartx-ilessp-1, 405
qpartx-ilessp-2, 406
qsort, 63–66
qsort, 384
qsort--1, 401
qsort-10, 64
qsort-10, 385, 401
qsort-3, 64
qsort-3, 385, 401
qsort-5, 64
qsort-5, 385, 401
qsort-base, 391
qsort-base-mem, 391
qsort-base-mem-sk, 391
qsort-base-rfile, 391
qsort-code, 62, 63
qsort-code, 383
qsort-correctness, 65
qsort-correctness, 402
qsort-correctness-la, 401
qsort-get-1, 407
qsort-get-2, 408
qsort-get3, 409
qsort-ilessp-1, 409
qsort-ilessp-2, 410
qsort-induct, 385
qsort-ordered, 410
qsort-orderedp1, 66
qsort-orderedp1, 410
qsort-orderedp1-la, 410
qsort-qpartx, 408
qsort-s-s0, 392
qsort-s-s0-mem, 392
qsort-s-s0-rfile, 392
qsort-s-s1, 394
qsort-s-s1-rfile, 394
qsort-s0-s0-1, 393
qsort-s0-s0-2, 393
qsort-s0-s0-mem-1, 393
qsort-s0-s0-mem-2, 394
qsort-s0-s0-rfile-1, 393

qsort-s0-s0-rfile-2, 393
qsort-s0-s1, 392
qsort-s0-s1-mem, 392
qsort-s0-s1-rfile, 392
qsort-s0p, 387
qsort-s1-crock, 397
qsort-s1-crock1, 397, 398
qsort-s1-crock2, 398
qsort-s1-crock2-crock, 398
qsort-s1-s, 397
qsort-s1-s2, 398
qsort-s1-s2-mem, 398, 399
qsort-s1p, 388
qsort-s2-s3, 395
qsort-s2-s3-mem, 395
qsort-s2-s3-rfile, 395
qsort-s2-s3-rfile-la, 395
qsort-s2p, 388
qsort-s3-crock, 399
qsort-s3-crock1, 400
qsort-s3-crock2, 400
qsort-s3-crock2-crock, 400
qsort-s3-s, 399
qsort-s3-s-la, 399
qsort-s3-s4, 400
qsort-s3-s4-mem, 400, 401
qsort-s3p, 389
qsort-s4-s5, 396
qsort-s4-s5-mem, 396
qsort-s4-s5-rfile, 396
qsort-s4-s5-rfile-la, 396
qsort-s4p, 390
qsort-s5p, 390
qsort-sk, 390
qsort-sk-1, 390, 391
qsort-sk-2, 391
qsort-sk-s-s1, 395
qsort-sk-s-s2, 399
qsort-sk-s-s3, 395
qsort-sk-s-s4, 401
qsort-sk-s-s5, 396
qsort-sp, 387
qsort-statep, 63, 65
qsort-statep, 386
qsort-statep-sp, 387
qsort-swap, 408
qsort-t, 63–65
qsort-t, 385
qsort-t--1, 401
qstack, 65, 66
qstack, 385, 386
qstack-0, 397
qstack-la0, 385
qstack-la1, 385, 397, 399
qstack-la2, 386, 399, 401
qstack-la3, 397
qstack-la4, 397
qstack-ubound, 66
qstack-ubound, 386
qstack-ubound-la-1, 67
qstack-ubound-la-1, 386
qstack-ubound-la-2, 67
qstack-ubound-la-2, 386
qsz, 142
Quick Sort, ii, 47, 61, 381
quot, 179, 327
quot-nat, 243
quot-nat-rangep, 259
quot2-sub12-induct, 137
quotient, 364, 375
quotient-0, 130
quotient-1, 131
quotient-2x, 131
quotient-2x-add1, 131
quotient-add1, 135
quotient-by-2, 368
quotient-crock, 136
quotient-diff, 230, 232
quotient-diff-la, 230
quotient-difference, 134
quotient-difference-plus1, 134
quotient-difference-plus2, 134
quotient-difference-times1, 134
quotient-difference-times2, 134
quotient-distributes-times2-add1, 136
quotient-equal-0, 131
quotient-exit, 130, 327
quotient-exp, 136
quotient-exp-lessp, 231, 232
quotient-generalize, 131
quotient-int-rangep, 237
quotient-leq, 131
quotient-lessp, 131
quotient-lessp-linear, 131
quotient-mono-1, 369
quotient-nat-rangep, 258
quotient-plus-add1, 133
quotient-plus-plus, 133
quotient-plus-times-exp2-1, 251
quotient-plus-times-exp2-2, 252
quotient-plus-times1, 133
quotient-plus-times2, 133
quotient-plus1, 133
quotient-plus2, 133
quotient-quotient, 137
quotient-shrink-fast, 135
quotient-shrink-fast-1, 694
quotient-sub1, 134

quotient-times, 133
quotient-times-cancel, 38
quotient-times-cancel, 136
quotient-times-exp2-1, 138
quotient-times-exp2-2, 138
quotient-times-exp2-3, 138
quotient-times-exp2-4, 138
quotient-times-lessp, 137
quotient-x-x, 131
quotient2-induct, 139

RAM, 149
ram-addrp, 51, 55, 59, 63, 69, 76, 78, 81, 83, 84, 87, 88, 102, 105
ram-addrp, 223, 296
ram-addrp-3, 353
ram-addrp-4, 353
ram-addrp-la1, 296
ram-addrp-la2, 296
read, 151
read->pc-read-mem, 299, 300
read-an, 49, 52, 56, 78, 87, 89
read-an, 157
read-dn, 33, 103, 106
read-dn, 157
read-lst, 298, 299
read-lst0, 298
read-mem, 30, 42, 43, 45, 49, 53, 57, 59, 60, 63, 65, 70, 72, 77, 80, 82, 84, 85, 87, 89, 102, 103, 105, 106
read-mem, 153, 329
read-mem-byte-write, 268
read-mem-byte-write-end, 268, 269
read-mem-ilst, 314
read-mem-ilst-asl, 318
read-mem-ilst-int, 314
read-mem-int-16, 351
read-mem-int-8, 351
read-mem-lst, 314
read-mem-lst-asl, 318
read-mem-lst-int, 314
read-mem-lst-la, 313
read-mem-lst0, 314
read-mem-mcode1-int, 299
read-mem-mcode2, 299
read-mem-mcode3, 300
read-mem-nat-rangep, 267
read-mem-non-numberp, 309, 310
read-mem-plus, 326, 327
read-memp, 31
read-memp, 153, 328
read-memp-ram0, 297
read-memp-ram1, 297
read-memp-ram1-asl, 319
read-memp-ram1-int, 297
read-memp-ram1-uint, 349
read-memp-ram2, 297
read-memp-ram3, 297
read-memp-rom0, 295
read-memp-rom1, 295
read-memp-rom1-asl, 320
read-memp-rom1-int, 295
read-memp-rom2, 295
read-memp-rom3, 295
read-rn, 29, 42, 49, 52, 53, 56, 60, 65, 72, 77, 80–85, 87, 89, 103, 106
read-rn, 152, 262
read-rn-0, 263, 264
read-rn-equal*, 336
read-rn-int-16, 350
read-rn-int-8, 350, 351
read-rn-nat-rangep, 259
read-signal, 28, 31
read-signal, 142
read-sp, 51, 53, 55, 56, 59, 60, 63, 65, 69, 70, 72, 76–78, 80–85, 87–89, 102, 103, 105, 106
read-sp, 158
read-write, 264
read-write-mem-end, 42, 43
read-write-mem-end, 268, 269
read-write-mem1, 43
read-write-mem1, 269
read-write-mem2, 43
read-write-mem2, 269
read-write-rn, 42
read-write-rn, 262
read-write-rn-end, 262
read-writem-mem, 321
read-writem-rn, 263
read-writem-rn-end, 263
reading memory, 153
readm-mem, 195
readm-mem-lst, 309
readm-rn, 195, 329
readm-rn-len, 263
readm-write-mem, 321
readm-write-rn, 263
readm-writem-mem, 322
readmp, 201
readp, 150
register-asl-ins, 185
register-asr-ins, 186
register-lsl-ins, 183
register-lsr-ins, 184
register-rol-ins, 181
register-ror-ins, 181
register-roxl-ins, 187

register-roxr-ins, 188
register-shift-rotate, 189
rem, 179, 327
rem-nat, 242
rem-nat-rangep, 259
remainder, 364, 375
remainder-0, 130
remainder-1, 130
remainder-2x, 131
remainder-2x-add1, 131
remainder-add1, 135
remainder-crock, 136
remainder-diff, 230, 232
remainder-diff-la, 229
remainder-difference, 133, 134
remainder-difference-remainder1, 135
remainder-difference-times1, 133
remainder-difference-times2, 133
remainder-distributes-times2-add1, 136
remainder-exit, 130, 327
remainder-exp, 136
remainder-int-rangep, 237
remainder-leq, 246
remainder-lessp, 131
remainder-lessp-linear, 131
remainder-plus-add1, 132
remainder-plus-cancel, 136
remainder-plus-cancel0, 136
remainder-plus-difference1, 132
remainder-plus-difference2, 132
remainder-plus-plus, 132
remainder-plus-plus-times1, 132
remainder-plus-plus-times2, 132
remainder-plus-remainder, 135
remainder-plus-remainder1, 38
remainder-plus-remainder1, 135
remainder-plus-remainder2, 135
remainder-plus-times-exp2-1, 251
remainder-plus-times-exp2-2, 251
remainder-plus-times1, 132
remainder-plus-times2, 132
remainder-plus1, 132
remainder-plus2, 132
remainder-quotient, 135
remainder-quotient-elim, 135
remainder-quotient-exp2, 252
remainder-remainder, 360
remainder-remainder-2, 138
remainder-remainder-exp2, 138
remainder-shrink-half, 360
remainder-sub1, 134
remainder-times, 132
remainder-times-exp2-1, 137
remainder-times-exp2-2, 137
remainder-times-exp2-3, 137
remainder-times-exp2-4, 138
remainder-trans, 437, 438
remainder-wrt-2, 132
remainder-x-x, 131
remainder2-plus-times-exp2, 251
replace, 29, 42
replace, 145, 255
replace-0, 229
replace-associativity, 253
replace-head, 253
replace-leq, 253
replace-leq1, 253
replace-nat-rangep, 254
replace-reflex, 252
reserved-signal, 28
reserved-signal, 142
rn-saved, 335
rol, 27
rol, 180
rol-c, 180, 328
rol-c-bitp, 271
rol-cvznx, 181
rol-effect, 181
rol-n, 181, 328
rol-n-bitp, 271
rol-z, 180, 328
rol-z-bitp, 271
ROM, 149
rom-addrp, 45, 51, 55, 59, 63, 69, 76, 81, 83, 84, 87, 88, 102, 105
rom-addrp, 223, 294
rom-addrp-la1, 294
rom-addrp-la2, 294
ror, 27
ror, 180
ror-c, 181, 328
ror-c-bitp, 271
ror-cvznx, 181
ror-effect, 181
ror-n, 181, 328
ror-n-bitp, 272
ror-z, 181, 328
ror-z-bitp, 272
roxl, 186
roxl-c, 187, 328
roxl-c-bitp, 272
roxl-cvznx, 187
roxl-effect, 187
roxl-n, 187, 328
roxl-n-bitp, 272
roxl-z, 187, 328
roxl-z-bitp, 272
roxr, 187

roxr-c, 187, 328
roxr-c-bitp, 273
roxr-cvznx, 188
roxr-effect, 188
roxr-n, 187, 328
roxr-n-bitp, 273
roxr-z, 187, 328
roxr-z-bitp, 273
rtd-ins, 203
rtd-mapping, 202
rtr-ins, 203
rts-addr, 48, 52, 56, 60, 65, 72, 77, 78, 80, 82–84, 87, 89, 102, 106
rts-addr, 335
rts-ins, 203

s, 154, 180, 189, 332
s-bit-subgroup, 215
s_mode, 31, 33
s_rn, 31, 33
scc-addr-modep, 208
scc-effect, 208
scc-group, 209
scc-group-h, 331
scc-ins, 208
set-c, 156
set-cvznx, 29
set-cvznx, 156, 261
set-cvznx-c, 260
set-cvznx-ccr, 260
set-cvznx-n, 260
set-cvznx-nat-rangep, 260
set-cvznx-v, 260
set-cvznx-x, 260
set-cvznx-z, 260
set-n, 156
set-set-cvznx1, 260
set-set-cvznx2, 260
set-v, 156
set-x, 156
set-z, 156
setn, 25
setn, 144
slen, 105
slen, 346, 348
slen-01, 347
slen-add1, 347
slen-lbound, 347
slen-put, 347
slen-put0, 347
slen-rec, 347
slen-ubound, 347
sp, 158
splus, 52, 56, 59, 64, 70, 71, 76, 81, 84
splus, 291, 334
sq, 56, 57
sq, 363
sq-add1-non-zero, 368
sr-cnt, 180
stepi, 11, 31, 32
stepi, 220
stepi-h, 333
stepi-p, 333
stepn, 11, 17, 31, 45, 48, 49, 52, 56, 60, 65, 72, 77, 80–84, 87, 89, 102, 105
stepn, 220, 329
stepn-gcd-loadp, 355
stepn-gcd3-loadp, 432
stepn-h, 333
stepn-lemma, 291
stepn-mcode-addrp, 334
stepn-mem-ilst, 327
stepn-mem-lst, 326
stepn-pc-read-mem, 334
stepn-pc-read-memp, 333
stepn-ram-addrp, 334
stepn-read-mem, 45
stepn-read-mem, 334
stepn-read-memp, 333
stepn-readm-mem, 326
stepn-rewriter, 291
stepn-rewriter0, 291
stepn-rom-addrp, 45
stepn-rom-addrp, 333
stepn-strcmp-loadp, 537
stepn-strcoll-loadp, 542
stepn-strlen-loadp, 562
stepn-strncmp-loadp, 579
stepn-strstr-loadp, 623
stepn-strtok-loadp, 643
stepn-write-memp, 333
strcat, 93, 339, 521
strcat-code, 522
strcat-correctness, 529
strcat-get-1, 680
strcat-get-2, 680
strcat-induct0, 522
strcat-induct1, 523
strcat-s-s0, 525
strcat-s-s0-mem, 525
strcat-s-s0-rfile, 525
strcat-s-s1-1, 524
strcat-s-s1-1-mem, 525
strcat-s-s1-1-rfile, 524
strcat-s-s1-2, 528
strcat-s-s1-2-mem, 529
strcat-s-s1-2-rfile, 528
strcat-s0-s0, 526

strcat-s0-s0-rfile, 526
strcat-s0-s1, 526
strcat-s0-s1-base, 525
strcat-s0-s1-rfile, 526
strcat-s0-s1-rfile-base, 525
strcat-s0p, 523
strcat-s0p-info, 526
strcat-s1-info, 528
strcat-s1-s1, 527
strcat-s1-s1-mem, 527
strcat-s1-s1-rfile, 527
strcat-s1-sn, 528
strcat-s1-sn-base, 527
strcat-s1-sn-mem, 528
strcat-s1-sn-mem-base, 527
strcat-s1-sn-rfile, 528
strcat-s1-sn-rfile-base, 527
strcat-s1p, 524
strcat-statep, 523
strcat-t, 522, 529
strcat-t0, 522
strcat-t1, 522
strcat-t2, 522
strchr, 95, 340, 529
strchr*, 343, 535
strchr*-strchr, 535
strchr-bounds, 348
strchr-code, 530
strchr-correctness, 534
strchr-get, 682
strchr-induct, 531
strchr-la, 555
strchr-non-zerop, 535
strchr-non-zerop-la, 535
strchr-s-s0, 532
strchr-s-s0-else, 532
strchr-s-s0-mem, 532
strchr-s-s0-rfile, 532
strchr-s0-s0, 533
strchr-s0-s0-rfile, 533
strchr-s0-sn, 534
strchr-s0-sn-base1, 532
strchr-s0-sn-base2, 533
strchr-s0-sn-rfile, 534
strchr-s0-sn-rfile-base1, 532
strchr-s0-sn-rfile-base2, 533
strchr-s0p, 531
strchr-s0p-info, 533, 534
strchr-statep, 531
strchr-strchr1, 692, 693
strchr-t, 531, 534
strchr-thm1, 680
strchr-thm2, 680
strchr-thm3, 680
strchr-thm4, 680
strchr1, 341
strchr1*, 343
strchr1*-strchr1, 639
strchr1-bounds, 341
strchr1-ch0, 690
strchr1-false-0, 341
strchr1-la, 616
strchr1-nth, 690
strchr1-nth-first, 690
strchr1-t, 531
strchr1-thm1, 688
strchr1-thm2, 688
strchr1-thm3, 688
strcmp, 94, 339, 535
strcmp-antisym, 678
strcmp-code, 536
strcmp-correctness, 540
strcmp-id, 678
strcmp-induct, 537
strcmp-j, 678
strcmp-j-1, 679
strcmp-j-2, 679
strcmp-s-s0, 538
strcmp-s-s0-else, 538
strcmp-s-s0-mem, 538
strcmp-s-s0-rfile, 538
strcmp-s0-s0, 539
strcmp-s0-s0-rfile, 540
strcmp-s0-sn, 540
strcmp-s0-sn-base1, 539
strcmp-s0-sn-base2, 539
strcmp-s0-sn-rfile, 540
strcmp-s0-sn-rfile-base1, 539
strcmp-s0-sn-rfile-base2, 539
strcmp-s0p, 538
strcmp-s0p-info, 540
strcmp-statep, 537
strcmp-strcpy, 678
strcmp-t, 537, 541
strcmp-thm1, 678
strcmp-thm2, 679
strcmp-trans-1, 678
strcmp1-t, 537
strcoll, 94, 339, 541
strcoll-code, 542
strcoll-correctness, 545
strcoll-s-s0, 543
strcoll-s-s0-else, 543
strcoll-s-s0-mem, 544
strcoll-s-s0-rfile, 543
strcoll-s0-s1, 544
strcoll-s0-s1-else, 544
strcoll-s0-s1-mem, 544

strcoll-s0-s1-rfile, 544
strcoll-s0p, 543
strcoll-s0p-strcmp-statep, 544
strcoll-s1-sn, 544
strcoll-s1-sn-rfile, 545
strcoll-s1p, 543
strcoll-statep, 542
strcoll-t, 542, 545
strcpy, 92, 339, 545
strcpy-0, 685
strcpy-code, 546
strcpy-correctness, 550
strcpy-cpy, 677
strcpy-get-1, 677
strcpy-induct, 547
strcpy-s-s0, 548
strcpy-s-s0-else, 548
strcpy-s-s0-mem, 548
strcpy-s-s0-rfile, 548
strcpy-s0-s0, 549
strcpy-s0-s0-mem, 549
strcpy-s0-s0-rfile, 549
strcpy-s0-sn, 549
strcpy-s0-sn-base, 548
strcpy-s0-sn-mem, 550
strcpy-s0-sn-mem-base, 549
strcpy-s0-sn-rfile, 549
strcpy-s0-sn-rfile-base, 548
strcpy-s0p, 547
strcpy-s0p-info, 549, 550
strcpy-statep, 547
strcpy-strchr, 681
strcpy-stringp, 678
strcpy-strlen, 677
strcpy-t, 547, 550
strcpy1, 339
strcpy1-get-1, 679
strcpy1-get-2, 679
strcpy1-get-3, 680
strcpy1-t, 546
strcpy2, 339
strcpy2-get-1, 684
strcpy2-get-2, 684
strcspn, 96, 340, 550
strcspn*, 344
strcspn-bounds, 348
strcspn-code, 552
strcspn-correctness, 560
strcspn-induct0, 552
strcspn-induct1, 553
strcspn-s-s0, 554
strcspn-s-s0-else, 554
strcspn-s-s0-mem, 555
strcspn-s-s0-rfile, 555
strcspn-s0-s0, 559
strcspn-s0-s0-rfile, 559
strcspn-s0-s1, 555
strcspn-s0-s1-1, 555
strcspn-s0-s1-else, 555
strcspn-s0-s1-rfile, 555
strcspn-s0-sn, 560
strcspn-s0-sn-base, 558
strcspn-s0-sn-rfile, 560
strcspn-s0-sn-rfile-base, 559
strcspn-s0p, 553
strcspn-s0p-info, 559, 560
strcspn-s1-1p, 554
strcspn-s1-1p-info, 557, 558
strcspn-s1-s0, 558
strcspn-s1-s0-base, 556
strcspn-s1-s0-rfile, 558
strcspn-s1-s0-rfile-base, 556
strcspn-s1-s1, 557
strcspn-s1-s1-1, 556
strcspn-s1-s1-rfile, 557
strcspn-s1-sn, 557
strcspn-s1-sn-base, 556
strcspn-s1-sn-rfile, 557
strcspn-s1-sn-rfile-base, 556
strcspn-s1p, 554
strcspn-s1p-s1-1p, 558
strcspn-statep, 553
strcspn-t, 552, 560
strcspn-t0, 552
strcspn-t1, 552
strcspn-t2, 552
strcspn-thm1, 688
strcspn-thm2, 688
strcspn-thm3, 688
stringp, 347
stringp-la, 348
strlen, 106
strlen, 98, 338, 561
strlen*, 343
strlen-01, 677
strlen-0p, 677
strlen-addr, 105
strlen-cdr, 689
strlen-code, 562
strlen-correctness, 565
strlen-ii, 677
strlen-ii1, 677
strlen-induct, 562
strlen-lb, 677
strlen-loadp, 105
strlen-mcdr, 689
strlen-non0p, 677
strlen-s-s0, 563

strlen-s-s0-else, 564
strlen-s-s0-mem, 564
strlen-s-s0-rfile, 564
strlen-s-sn, 563
strlen-s-sn-mem, 563
strlen-s-sn-rfile, 563
strlen-s0-s0, 564
strlen-s0-s0-rfile, 565
strlen-s0-sn, 565
strlen-s0-sn-base, 564
strlen-s0-sn-rfile, 565
strlen-s0-sn-rfile-base, 564
strlen-s0p, 563
strlen-s0p-info, 565
strlen-statep, 562
strlen-statep-info, 565
strlen-strchr1, 690
strlen-strchr1-nth, 690
strlen-t, 562, 566
strlen-ub, 676
strlen1, 679
strlen1-ii, 684
strlen1-strlen, 679
strlen1-t, 562
strncat, 93, 339, 566
strncat-code, 567
strncat-correctness, 576
strncat-get-1, 684
strncat-get-2, 685
strncat-induct0, 568
strncat-induct1, 568
strncat-s-s0, 570
strncat-s-s0-mem, 571
strncat-s-s0-rfile, 571
strncat-s-s1, 571
strncat-s-s1-1, 573
strncat-s-s1-else, 571
strncat-s-s1-else-1, 573
strncat-s-s1-mem, 571
strncat-s-s1-mem-1, 574
strncat-s-s1-rfile, 571
strncat-s-s1-rfile-1, 573
strncat-s-sn, 570
strncat-s-sn-mem, 570
strncat-s-sn-rfile, 570
strncat-s0-s0, 572
strncat-s0-s0-rfile, 572
strncat-s0-s1, 572
strncat-s0-s1-base, 572
strncat-s0-s1-rfile, 573
strncat-s0-s1-rfile-base, 572
strncat-s0-sn-rfile-base1, 574
strncat-s0p, 569
strncat-s0p-info, 572, 573
strncat-s1-s1, 575
strncat-s1-s1-mem, 576
strncat-s1-s1-rfile, 575
strncat-s1-sn, 576
strncat-s1-sn-base1, 574
strncat-s1-sn-base2, 574
strncat-s1-sn-mem, 576
strncat-s1-sn-mem-base1, 574
strncat-s1-sn-mem-base2, 575
strncat-s1-sn-rfile, 576
strncat-s1-sn-rfile-base2, 575
strncat-s1p, 569
strncat-statep, 568
strncat-t, 568, 577
strncat-t0, 568
strncat-t1, 568
strncat-t2, 568
strncmp, 106
strncmp, 94, 340, 577
strncmp-code, 578
strncmp-correctness, 584
strncmp-id, 686
strncmp-induct, 579
strncmp-j, 687
strncmp-j-1, 687
strncmp-j-2, 687
strncmp-loadp, 105
strncmp-s-s0, 581
strncmp-s-s0-else, 581
strncmp-s-s0-mem, 581
strncmp-s-s0-rfile, 581
strncmp-s-sn, 580
strncmp-s-sn-mem, 581
strncmp-s-sn-rfile, 580
strncmp-s0-s0, 583
strncmp-s0-s0-rfile, 583
strncmp-s0-sn, 583
strncmp-s0-sn-base1, 581
strncmp-s0-sn-base2, 582
strncmp-s0-sn-base3, 582
strncmp-s0-sn-rfile, 583
strncmp-s0-sn-rfile-base1, 582
strncmp-s0-sn-rfile-base2, 582
strncmp-s0-sn-rfile-base3, 582
strncmp-s0p, 580
strncmp-statep, 579
strncmp-strncmp2, 689
strncmp-t, 579, 584
strncmp-thm1, 686
strncmp-thm2, 687
strncmp1, 340
strncmp1-id, 686
strncmp1-strncmp2, 689
strncmp1-t, 579

strncmp1-thm1, 686
strncmp2, 106
strncmp2, 689
strncmp2-0, 689
strncmp2-mcdr-1, 689
strncmp2-mcdr-2, 689
strncmp2-non-numberp, 689
strncpy, 93, 340, 584
strncpy-0s, 686
strncpy-code, 586
strncpy-correctness, 594
strncpy-cpy, 686
strncpy-induct1, 586
strncpy-induct2, 586
strncpy-s-s0, 588
strncpy-s-s0-else, 588
strncpy-s-s0-mem, 589
strncpy-s-s0-rfile, 589
strncpy-s-sn, 588
strncpy-s-sn-mem, 588
strncpy-s-sn-rfile, 588
strncpy-s0-s0, 592
strncpy-s0-s0-mem, 593
strncpy-s0-s0-rfile, 593
strncpy-s0-s1, 589
strncpy-s0-s1-mem, 589
strncpy-s0-s1-rfile, 589
strncpy-s0-sn, 593
strncpy-s0-sn-base1, 591
strncpy-s0-sn-base2, 592
strncpy-s0-sn-mem, 593
strncpy-s0-sn-mem-base1, 592
strncpy-s0-sn-mem-base2, 592
strncpy-s0-sn-rfile, 593
strncpy-s0-sn-rfile-base1, 591
strncpy-s0-sn-rfile-base2, 592
strncpy-s0p, 587
strncpy-s1-s1, 590
strncpy-s1-s1-mem, 590
strncpy-s1-s1-rfile, 590
strncpy-s1-sn, 590
strncpy-s1-sn-base, 589
strncpy-s1-sn-mem, 591
strncpy-s1-sn-rfile, 591
strncpy-s1-sn-rfile-base, 590
strncpy-s1p, 587
strncpy-statep, 586
strncpy-strlen, 685
strncpy-t, 586, 594
strncpy-t0, 586
strncpy-t1, 586
strncpy-t2, 586
strncpy1, 340
strncpy1-0s, 686
strncpy1-cpy, 686
strncpy1-get, 685
strncpy1-strlen, 685
strpbrk, 96, 341, 594
strpbrk*, 344, 605
strpbrk*-strpbrk, 604
strpbrk-bounds, 348
strpbrk-code, 595
strpbrk-correctness, 604
strpbrk-induct0, 596
strpbrk-induct1, 596
strpbrk-non-zerop, 605
strpbrk-non-zerop-la, 604
strpbrk-s-s0, 598
strpbrk-s-s0-else, 598
strpbrk-s-s0-mem, 598
strpbrk-s-s0-rfile, 598
strpbrk-s0-s0, 603
strpbrk-s0-s0-rfile, 603
strpbrk-s0-s1, 599
strpbrk-s0-s1-else, 599
strpbrk-s0-s1-rfile, 599
strpbrk-s0-sn, 603
strpbrk-s0-sn-base1, 599
strpbrk-s0-sn-base2, 602
strpbrk-s0-sn-rfile, 603
strpbrk-s0-sn-rfile-base1, 599
strpbrk-s0-sn-rfile-base2, 602
strpbrk-s0p, 597
strpbrk-s0p-info, 603
strpbrk-s1-s0, 600
strpbrk-s1-s0-base, 599
strpbrk-s1-s0-rfile, 601
strpbrk-s1-s0-rfile-base, 600
strpbrk-s1-s1, 600
strpbrk-s1-s1-rfile, 600
strpbrk-s1-sn, 601
strpbrk-s1-sn-base, 601
strpbrk-s1-sn-rfile, 602
strpbrk-s1-sn-rfile-base, 601
strpbrk-s1p, 597
strpbrk-s1p-info, 600
strpbrk-statep, 597
strpbrk-t, 596, 604
strpbrk-t0, 596
strpbrk-t1, 596
strpbrk-t2, 596
strpbrk-thm1, 687
strpbrk-thm2, 687
strpbrk-thm3, 687
strrchr, 96, 341, 605
strrchr*, 344, 611
strrchr*-strrchr, 611
strrchr-bounds, 348

strrchr-code, 606
strrchr-correctness, 610
strrchr-induct, 606
strrchr-la1, 682
strrchr-la2, 682
strrchr-la3, 682
strrchr-non-zerop, 611
strrchr-non-zerop-la, 611
strrchr-s-s0, 607
strrchr-s-s0-else, 607
strrchr-s-s0-mem, 608
strrchr-s-s0-rfile, 608
strrchr-s0-s0-1, 609
strrchr-s0-s0-2, 609
strrchr-s0-s0-rfile-1, 609
strrchr-s0-s0-rfile-2, 610
strrchr-s0-sn, 610
strrchr-s0-sn-base1, 608
strrchr-s0-sn-base2, 608
strrchr-s0-sn-rfile, 610
strrchr-s0-sn-rfile-base1, 608
strrchr-s0-sn-rfile-base2, 609
strrchr-s0p, 607
strrchr-s0p-info, 610
strrchr-s0p-la, 610
strrchr-statep, 607
strrchr-strchr-la, 682
strrchr-t, 606, 611
strrchr-t1, 606
strrchr-thm1, 682
strrchr-thm2, 683
strrchr-thm3, 683
strrchr-thm4, 683
strspn, 96, 341, 612
strspn*, 344
strspn-bounds, 348
strspn-code, 613
strspn-correctness, 621
strspn-induct0, 613
strspn-induct1, 614
strspn-s-s0, 615
strspn-s-s0-else, 615
strspn-s-s0-mem, 616
strspn-s-s0-rfile, 616
strspn-s0-s0, 620
strspn-s0-s0-rfile, 620
strspn-s0-s1, 616
strspn-s0-s1-1, 616
strspn-s0-s1-else, 616
strspn-s0-s1-rfile, 616
strspn-s0-sn, 620
strspn-s0-sn-base, 619
strspn-s0-sn-rfile, 621
strspn-s0-sn-rfile-base, 619
strspn-s0p, 614
strspn-s0p-info, 620, 621
strspn-s1-1p, 615
strspn-s1-1p-info, 618
strspn-s1-s0, 619
strspn-s1-s0-base, 617
strspn-s1-s0-rfile, 619
strspn-s1-s0-rfile-base, 617
strspn-s1-s1, 618
strspn-s1-s1-1, 617
strspn-s1-s1-rfile, 618
strspn-s1-sn, 618
strspn-s1-sn-base, 616
strspn-s1-sn-rfile, 618
strspn-s1-sn-rfile-base, 617
strspn-s1p, 615
strspn-s1p-s1-1p, 619
strspn-statep, 614
strspn-strchr1, 691
strspn-t, 613, 621
strspn-t0, 613
strspn-t1, 613
strspn-t2, 613
strspn-thm1, 688
strspn-thm2, 688
strspn-thm3, 688
strspn-true, 692
strspn-ubound, 348
strstr, 104, 106
strstr, 90, 97, 103, 342, 621
strstr*, 106
strstr*, 345, 640
strstr*-strstr, 639
strstr-addr, 104, 105
strstr-bounds, 348
strstr-code, 105
strstr-code, 623
strstr-correctness, 105
strstr-correctness, 638
strstr-induct1, 624
strstr-induct2, 625
strstr-load, 104
strstr-loadp, 104, 105
strstr-non-zerop, 640
strstr-non-zerop-la, 639
strstr-s-s0, 628
strstr-s-s0-else, 628
strstr-s-s0-mem, 629
strstr-s-s0-rfile, 629
strstr-s-s2, 635
strstr-s-s2-else, 635
strstr-s-s2-mem, 635
strstr-s-s2-rfile, 635
strstr-s-sn, 628

strstr-s-sn-mem, 628
strstr-s-sn-rfile, 628
strstr-s0-s1, 629
strstr-s0-s1-else, 629
strstr-s0-s1-mem, 629
strstr-s0-s1-rfile, 629
strstr-s0p, 625
strstr-s0p-strlen-statep, 629
strstr-s1-s2, 630
strstr-s1-s2-else, 630
strstr-s1-s2-rfile, 630
strstr-s1p, 626
strstr-s2-s2, 631
strstr-s2-s2-1, 635
strstr-s2-s2-mem-1, 636
strstr-s2-s2-rfile, 631
strstr-s2-s2-rfile-1, 636
strstr-s2-s3, 633
strstr-s2-s3-base, 630
strstr-s2-s3-base-mem, 631
strstr-s2-s3-base-rfile, 631
strstr-s2-s3-else, 633
strstr-s2-s3-mem, 634
strstr-s2-s3-rfile, 634
strstr-s2-sn, 634
strstr-s2-sn-2, 638
strstr-s2-sn-base, 630
strstr-s2-sn-base-1, 637
strstr-s2-sn-base-mem-1, 637
strstr-s2-sn-base-rfile, 630
strstr-s2-sn-base-rfile-1, 637
strstr-s2-sn-mem-2, 638
strstr-s2-sn-rfile, 634
strstr-s2-sn-rfile-2, 638
strstr-s2p, 626
strstr-s2p-info, 633, 638
strstr-s3-s4, 632
strstr-s3-s4-else, 632
strstr-s3-s4-mem, 632
strstr-s3-s4-rfile, 632
strstr-s3p, 627
strstr-s3p-strncmp-statep, 631
strstr-s4-s2, 633
strstr-s4-s2-rfile, 633
strstr-s4-sn, 632
strstr-s4-sn-rfile, 633
strstr-s4p, 627
strstr-statep, 104, 105
strstr-statep, 625
strstr-statep-info, 638, 639
strstr-t, 104, 105
strstr-t, 624, 639
strstr-t0, 624
strstr-t1, 624
strstr-t2, 624
strstr-t3, 624
strstr-thm1, 690
strstr-thm2, 106
strstr-thm2, 690
strstr-thm3, 106
strstr-thm3, 691
strstr1, 106
strstr1, 342
strstr1*, 344
strstr1-thm1, 106
strstr1-thm1, 690
strstr1-thm2, 690
strstr1-thm3, 691
strtok, 97, 108, 640
strtok-code, 643
strtok-correctness, 663
strtok-induct0, 644
strtok-induct1, 645
strtok-induct2, 645
strtok-induct3, 645
strtok-last, 345
strtok-last0, 345
strtok-last1, 345
strtok-lst, 345
strtok-lst-1, 664
strtok-lst-2, 664
strtok-lst0, 345
strtok-lst1, 345
strtok-lst2, 345
strtok-mem, 665
strtok-rfile, 664
strtok-s-s0-1, 649
strtok-s-s0-2, 650
strtok-s-s0-else-1, 649
strtok-s-s0-else-2, 650
strtok-s-s0-mem-1, 649
strtok-s-s0-mem-2, 650
strtok-s-s0-rfile-1, 649
strtok-s-s0-rfile-2, 650
strtok-s-sn, 648
strtok-s-sn-mem, 649
strtok-s-sn-rfile, 649
strtok-s0-s0, 654
strtok-s0-s0-rfile, 654
strtok-s0-s1, 650
strtok-s0-s1-else, 650
strtok-s0-s1-rfile, 651
strtok-s0-s2, 654
strtok-s0-s2-base, 653
strtok-s0-s2-else, 654
strtok-s0-s2-rfile, 655
strtok-s0-s2-rfile-base, 653
strtok-s0p, 646

strtok-s0p-info, 654
strtok-s1-s0, 652
strtok-s1-s0-base, 651
strtok-s1-s0-rfile, 652
strtok-s1-s0-rfile-base, 651
strtok-s1-s1, 652
strtok-s1-s1-rfile, 652
strtok-s1-s2, 653
strtok-s1-s2-base, 651
strtok-s1-s2-rfile, 653
strtok-s1-s2-rfile-base, 651
strtok-s1p, 646
strtok-s1p-info, 652, 653
strtok-s2-s3, 655
strtok-s2-s3-else, 656
strtok-s2-s3-rfile, 656
strtok-s2-sn, 662
strtok-s2-sn-1, 655
strtok-s2-sn-mem, 663
strtok-s2-sn-mem-1, 655
strtok-s2-sn-rfile, 663
strtok-s2-sn-rfile-1, 655
strtok-s2p, 647
strtok-s3-s3, 661
strtok-s3-s3-rfile, 661
strtok-s3-s4, 656
strtok-s3-s4-else, 656
strtok-s3-s4-rfile, 656
strtok-s3-sn, 662
strtok-s3-sn-base, 660
strtok-s3-sn-mem, 662
strtok-s3-sn-mem-base, 661
strtok-s3-sn-rfile, 662
strtok-s3-sn-rfile-base, 661
strtok-s3p, 647
strtok-s3p-info, 662
strtok-s4-s3, 660
strtok-s4-s3-base, 658
strtok-s4-s3-rfile, 660
strtok-s4-s3-rfile-base, 658
strtok-s4-s4, 658
strtok-s4-s4-rfile, 659
strtok-s4-sn, 659
strtok-s4-sn-1, 656
strtok-s4-sn-2, 657
strtok-s4-sn-mem, 659
strtok-s4-sn-mem-1, 657
strtok-s4-sn-mem-2, 658
strtok-s4-sn-rfile, 659
strtok-s4-sn-rfile-1, 657
strtok-s4-sn-rfile-2, 657
strtok-s4p, 648
strtok-s4p-info, 660
strtok-statep, 645
strtok-t, 644, 665
strtok-t0, 643
strtok-t1, 644
strtok-t2, 644
strtok-t3, 644
strtok-t4, 644
strtok-t5, 644
strtok-t6, 644
strtok-thm1, 691
strtok-thm2, 691
strtok-thm3, 692
strtok-thm4, 692
strtok-thm5, 692
strtok-thm6, 692
strtok-tok, 345
strxfrm, 95, 107, 346, 665
strxfrm-code, 667
strxfrm-correctness, 675
strxfrm-induct1, 668
strxfrm-induct2, 668
strxfrm-n, 667
strxfrm-s-s0, 670
strxfrm-s-s0-else, 670
strxfrm-s-s0-mem, 670
strxfrm-s-s0-rfile, 670
strxfrm-s-sn, 669
strxfrm-s-sn-mem, 670
strxfrm-s-sn-rfile, 670
strxfrm-s0-s0, 674
strxfrm-s0-s0-mem, 675
strxfrm-s0-s0-rfile, 674
strxfrm-s0-s1, 673
strxfrm-s0-s1-else, 673
strxfrm-s0-s1-mem, 673
strxfrm-s0-s1-rfile, 673
strxfrm-s0-sn, 675
strxfrm-s0-sn-base1, 672
strxfrm-s0-sn-base2, 673
strxfrm-s0-sn-mem, 675
strxfrm-s0-sn-mem-base1, 673
strxfrm-s0-sn-mem-base2, 674
strxfrm-s0-sn-rfile, 675
strxfrm-s0-sn-rfile-base1, 672
strxfrm-s0-sn-rfile-base2, 674
strxfrm-s0p, 668
strxfrm-s0p-info, 675
strxfrm-s1-s1, 671
strxfrm-s1-s1-rfile, 671
strxfrm-s1-sn, 672
strxfrm-s1-sn-base, 670
strxfrm-s1-sn-mem, 672
strxfrm-s1-sn-mem-base, 671
strxfrm-s1-sn-rfile, 672
strxfrm-s1-sn-rfile-base, 671

strxfrm-s1p, 669
strxfrm-s1p-info, 671, 672
strxfrm-statep, 668
strxfrm-t, 668, 676
strxfrm-t0, 667
strxfrm-t1, 667
strxfrm-t2, 667
strxfrm1, 346
sub, 26, 33, 34, 51, 53, 55, 56, 59, 60, 63, 65, 69, 72, 76–78, 81–85, 87–89, 102, 103, 105, 106
sub, 145, 247
sub-0, 245
sub-add, 248
sub-adder, 247, 249
sub-addr-modep1, 32, 33
sub-addr-modep1, 171
sub-addr-modep2, 171
sub-bcs, 284
sub-bcs&cc, 44
sub-beq-ext, 280
sub-beq-int-0, 279
sub-beq-int-1, 279
sub-beq-uint, 44
sub-beq-uint, 279
sub-bge, 44
sub-bge, 289
sub-bgt, 45
sub-bgt, 290
sub-bhi-0, 278
sub-bhi-1, 278
sub-bhi-int, 278
sub-ble, 290
sub-bls, 44
sub-bls, 278
sub-blt, 289
sub-bmi, 44
sub-bmi, 287
sub-bvs, 286
sub-bvs&vc, 44
sub-c, 33
sub-c, 170, 278
sub-c-bitp, 270
sub-cancel, 248
sub-cancel0, 248
sub-cvznx, 33
sub-cvznx, 170
sub-effect, 33
sub-effect, 170
sub-equal-0, 247
sub-group, 173
sub-group-h, 331
sub-ins1, 32
sub-ins1, 171
sub-ins2, 171
sub-int, 250
sub-leq-1, 274
sub-leq-2, 274
sub-leq-la, 246
sub-mapping, 171
sub-n, 33, 34, 44, 45
sub-n, 170, 327
sub-n-bitp, 270
sub-nat-la, 243, 245
sub-nat-rangep, 246
sub-neg, 274
sub-sub, 248
sub-sub1, 248
sub-subx-c, 277, 291
sub-subx-v, 285, 291
sub-uint, 245
sub-v, 33, 34, 44, 45
sub-v, 170, 327
sub-v-bitp, 270
sub-x-x, 246
sub-z, 33, 34, 44, 45
sub-z, 170, 290
sub-z-bitp, 270
sub-z-la, 278, 291
sub-z-la1, 290, 291
sub1-int-rangep, 237
sub1-nat-rangep, 258
sub1-of-1, 117
sub1-times2-nat-rangep, 258
suba-addr-modep, 171
suba-ins, 171
subbagp, 116
subbagp-bagint1, 116
subbagp-bagint2, 116
subbagp-cdr1, 116
subbagp-cdr2, 116
subbagp-delete, 116
subbagp-implies-plus-tree-geq, 120
subi-addr-modep, 216
subi-ins, 217
subi-subgroup, 217
sublst-ileq, 402
sublst-ileq-lemma, 403
sublst-ileq-put, 403
sublst-ileq-qpart-aux, 404
sublst-ileq-qsort, 404
sublst-ileq-swap, 404
sublst-ileq-swap-la, 403
sublst-ileq-swap-swap, 403
sublsts-ileq, 402
sublsts-ileq-la1, 403
sublsts-ileq-la2, 409
sublsts-ileq-put, 403

sublsts-ileq-qpart-aux, 404
sublsts-ileq-qsort1, 405
sublsts-ileq-qsort2, 405
sublsts-ileq-swap, 404
sublsts-ileq-swap-la, 403
sublsts-ileq-swap-swap, 404
subq-addr-modep, 209
subq-ins, 209
subset, 115
subtracter, 26
subtracter, 250
subtracter-int, 250
subtracter-nat-la, 244, 245
subtracter-nat-rangep, 246
subtracter-uint, 244
subtractor, 145
subx-addx-v, 285, 291
subx-c, 172, 278
subx-c-bitp, 270
subx-c-la, 277, 291
subx-cvznx, 172
subx-effect, 172
subx-ins1, 172
subx-ins2, 172
subx-n, 172, 327
subx-n-bitp, 270
subx-v, 172, 327
subx-v-bitp, 270
subx-v-la, 285, 291
subx-z, 172, 327
subx-z-bitp, 270
swap, 64, 65
swap, 338, 403
swap-0, 411
swap-commute, 402
swap-cvznx, 194
swap-effect, 194
swap-induct, 412
swap-ins, 194
swap-n, 194, 328
swap-n-bitp, 273
swap-put-commute, 402
swap-rec-la, 412
swap-z, 194, 328
swap-z-bitp, 273
switch statement, 85

t3, 330
ta-lemma-1, 140
ta-lemma-2, 140
tail, 25, 27, 30, 39
tail, 143, 254
tail-0, 228
tail-app, 252
tail-equal-0, 229
tail-head, 252
tail-lemma, 229
tail-leq, 229
tail-lessp, 229
tail-lst, 298
tail-mbit, 284, 285
tail-nat-rangep, 259
tail-of-0, 228
tail-replace, 252
tail-tail, 251
tas-addr-modep, 198
tas-effect, 198
tas-ins, 198
times, 130, 375
times-1, 123
times-add1, 123
times-add1-sub1, 123
times-associativity, 123
times-commutativity, 123
times-commutativity1, 123
times-distributes-difference, 124
times-distributes-difference1, 124
times-distributes-plus, 123, 137
times-distributes-plus-new, 137
times-distributes-remainder, 136
times-equal-0, 123
times-equal-cancel, 124
times-equal-cancel0, 123
times-exp2-lessp, 130
times-exp2-nat-rangep, 258
times-fringe, 125
times-lessp, 139, 240–242
times-lessp-0, 124
times-lessp-1, 124
times-lessp-cancel, 124
times-lessp-cancel-1, 124
times-lessp-cancel0, 124
times-lessp-cancel1, 319
times-lessp-dual, 693
times-lessp-linear, 124
times-plus-lessp, 135
times-plus-lessp-cancel, 305, 307
times-sub1, 231
times-tree, 125
times-tree-times-fringe, 125
times-zero, 123
times2-add1-lessp-cancel, 124
transwap, 410
trapv-ins, 203
tst-addr-modep, 197
tst-ins, 197
tst-subgroup, 198

uint-rangep, 43, 63, 102, 105
uint-rangep, 148
uint-rangep-la, 235
uint-to-nat, 40
uint-to-nat, 148, 329
uint-to-nat-rangep, 236
uint-to-nat-to-uint, 236
unavailable, 149
unlk-ins, 202
unlk-subgroup, 202
update-ccr, 157
update-mem, 157
update-pc, 32
update-pc, 157
update-rfile, 157
uread-an, 335
uread-dn, 334
uread-mem, 102
uread-mem, 334

value-field, 148
vec, 224
Verdix Ada compiler, ii, 2, 47, 54

w, 28
w, 141
word-read, 154
word-readp, 153
write, 151
write-an, 157
write-dn, 157
write-else-mem-ilst, 315
write-else-mem-lst, 315
write-mem, 30, 42, 43
write-mem, 155, 329
write-mem-byte-write, 269
write-mem-ilst, 317
write-mem-ilst-asl, 319
write-mem-ilst-int, 318
write-mem-ilst0, 317
write-mem-lst, 317
write-mem-lst-asl, 319
write-mem-lst-int, 317
write-mem-lst-la, 315
write-mem-lst0, 317
write-mem-maintain-byte-readp, 267
write-mem-maintain-byte-writep, 267
write-mem-maintain-movep-writep, 331
write-mem-maintain-pc-byte-readp, 267
write-mem-maintain-pc-read-memp, 267
write-mem-maintain-ram-addrp, 292
write-mem-maintain-read-memp, 267
write-mem-maintain-rom-addrp, 292
write-mem-maintain-write-memp, 267
write-mem-mcode-addrp, 300
write-memp, 155, 329
write-memp->read-memp, 268
write-memp-la0, 293, 329
write-memp-la1, 293, 329
write-memp-la2, 293, 329
write-memp-la3, 293, 329
write-memp-ram0, 296
write-memp-ram1, 296
write-memp-ram1-asl, 320
write-memp-ram1-int, 296
write-memp-ram1-uint, 349
write-memp-ram2, 296
write-memp-ram3, 296
write-rn, 29, 42
write-rn, 152, 262
write-rn-len, 262
write-signal, 28
write-signal, 142
write-sp, 158
write-write, 264
write-write-induct, 269
write-write-la, 264
write-write-mem, 269
write-write-rn, 262
writem-else-mem-ilst, 321
writem-else-mem-lst, 321
writem-mem, 195
writem-mem-h, 332
writem-mem-maintain-byte-readp, 320
writem-mem-maintain-byte-writep, 320
writem-mem-maintain-pc-byte-readp, 320
writem-mem-maintain-pc-read-memp, 320
writem-mem-maintain-ram-addrp, 321
writem-mem-maintain-read-memp, 320
writem-mem-maintain-rom-addrp, 320
writem-mem-maintain-write-memp, 320
writem-mem-mcode-addrp, 321
writem-rn, 195, 329
writemp, 196
writep, 150
writep->readp, 264
wsz, 32
wsz, 141

z-flag-la, 283
zero-list, 340, 685
zero-list-get, 685
zero-list1, 340
zero-list1-get, 685
zero-list1-la, 685
zerop-makes-equal-true-bridge, 128
zerop-makes-lessp-false-bridge, 126
zerop-makes-times-tree-zero, 125