Machine-Checked Real-Time System Verification

by

Matthew Michael Wilding, B.S., M.S.C.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

May 1996

Machine-Checked Real-Time System Verification

APPROVED BY

DISSERTATION COMMITTEE:

To Martha.

Acknowledgments

I thank my friends and colleagues at Computational Logic, past and present, where much of this work was accomplished under contract MDA904-93-C-4070 with the Defense Advanced Research Projects Agency. Ken Albin, Bill Bevier, Don Good, Warren Hunt, Mike Smith, David Russinoff, Bill Young, and many other CLIers helped me greatly.

I thank my dissertation committee: my co-supervisors Bob Boyer and Al Mok, J.C. Browne, Matt Kaufmann, and J Moore. I also thank members of my original committee who were replaced when my research focus shifted dramatically: the late Woody Bledsoe, Alan Cline, and Robert van de Geijn.

Bob Boyer's influence pervades this dissertation, and I am deeply grateful for his wisdom and generosity.

Most especially, I thank my entire family for their love and support.

Machine-Checked Real-Time System Verification

Publication No. __________

Matthew Michael Wilding, PhD.
The University of Texas at Austin, 1996

Supervisors: Robert S. Boyer and Aloysius K. Mok

The complexity and importance of real-time systems makes their formalization and verification crucial. We ensure proper real-time system behavior in several ways through mathematical proof. We check proofs using Nqthm, a mechanical proof system also known as the Boyer-Moore theorem prover.

We present a scheduling policy and a theorem that assures that the policy is optimal. A common approach to real-time verification involving tasks and scheduling is formalized that allows the theorem statement and proof.

We verify a high-level language application that, using a proved compiler and microprocessor design, assures the optimality and timely behavior of an application. The verified program uses a clever algorithm to meet its specification and a method for structuring the proof allows large verified programs to be developed. An interpreter-based language semantics allows reliable execution behavior reasoning, including reasoning about timing.

We formalize in a logic a real-time model of the FM9001 microprocessor and a timed automata specification method for real-time system properties. We specify and prove the correct operation of a small real-time system, a quiz-show signaling system, using our microprocessor model to model computation and timed automata to describe system properties.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

A real-time system is a system in which correct operation depends upon meeting timing constraints. A real-time program is a program that works in a real-time system. Real-time programs operate power plants, fly aircraft, regulate pacemakers, control chemical processing, and do many other things. The system within which a real-time program operates typically has elements that are not under program control whose behavior is somewhat unpredictable. Real-time systems tend to be safety-critical and thus the reliability of real-time programs is of particular concern.

The adoption of traditional software engineering practices such as rigorous testing helps computer programmers develop applications with fewer errors, but a high degree of confidence cannot be achieved using these methods. One approach to developing dependable software is to formalize a problem specification and prove that a particular program meets its requirements. Proving real-time programs correct is a particular challenge because of timing constraints and because real-time systems are not completely under program control. Advances in software and hardware verification have led to the development of very detailed models of computer systems that allow reasoning about computer applications based upon realistic computational models. Extra reliability can be attained by checking the proofs with an automatic theorem prover, especially since proofs about computer systems tend to be very complicated and difficult to check completely without mechanical assistance.

This dissertation describes a project in which properties of real-time systems are specified and proved. There are three main parts.

1.2 Project Overview

A common framework in which to consider development of real-time programs is to divide an application into tasks and to view the execution of the application in terms of responding to task requests. A real-time scheduling policy is an algorithm for selecting which task of an application to execute. By viewing system execution in this abstract fashion and making assumptions about the timing behavior of the individual tasks and the occurrence of task requests, one can show that a scheduling policy achieves specified constraints on task execution timing.

The first part of this dissertation is a machine-checked proof of the optimality of a real-time scheduling policy that chooses a task with an unfulfilled task request that has earliest deadline assuming that task requests for each task occur periodically.

Researchers have defined the semantics of a high-level language using an interpreter that allows direct calculation of the effect of instructions and proved that the language semantics are preserved by a compiler that generates machine code. The target microprocessor has also been described with an interpreter function proved implemented by the processor's hardware design. The formal high-level language description allows more than the development of a dependable compiler; it can support formal reasoning about the behavior of applications written in the language.

The second part of this dissertation presents an application that uses the definition of a language interpreter in its specification. The specification includes functional correctness properties as well as bounds on the number of instructions executed and is proved using a mechanical theorem prover. The target microprocessor has been fabricated and powers a single board computer, so we are able to execute our application on an actual processor with great confidence in its operation because of our verification and the efforts of those who verified the compiler and processor.

Researchers and software developers have used timed automata to describe aspects of real-time systems including desired system operation and computation. Although an intuitive and therefore attractive approach to real-time system specification, this approach is rather abstract and does not take advantage of the very detailed models of computing systems available.

The third part of this dissertation presents several formalisms useful for verifying real-time system with applications executing within them. We present a detailed processor model that allows useful reasoning about its functional, timing, and I/O behavior and an abstract, timed automata method for specifying real-time system properties. We present two small examples verified with our method using a mechanical theorem prover. Our approach to real-time system verification combines the best of both worlds: intuitive, abstract specifications of needed system properties and computation modeled with a very detailed model of a real processor running the bit-vectors of our application. We use mathematical proof to connect these different levels of detail and abstraction. These proofs are very elaborate even for simple systems, and real-time applications tend to be safety-critical, so we use a mechanical theorem prover to ensure that the proof is correct. Our specifications are appropriately abstract and our computation model is realistic in the sense that the verified bit-vectors are exactly what is loaded onto the processor.

1.3 Dissertation Overview

Chapter 2 describes the mechanical theorem prover Nqthm and our use of it, and the notation we use in this dissertation.

The first part of the dissertation is contained in Chapter 3, which presents a formalization and mechanical proof of a real-time scheduling policy theorem.

The second part of the dissertation is contained in Chapter 4, which presents a formalization and mechanical proof of an application whose specification requires functional correctness, bounds on numbers of executed instructions, and proper execution in a verified environment.

The third part of the dissertation is contained in Chapters 5-8.

Chapter 5 presents a processor model that can be used as the basis for real-time program verification, and describes the physical construction of a single board computer that uses it.

Chapter 6 presents a method for real-time system specification that uses timed automata.

Chapter 7 presents an example verified real-time system, a quiz-show signaling system, including its specification using timed automata and the processor model and an outline of its machine-checked proof.

Chapter 8 describes some quiz-show system verification implications.

Chapter 9 presents work related to this dissertation that is not discussed in earlier chapters.

Several appendixes contain listings of hardware designs, programs, and theorem prover input associated with this project. There is an index of Nqthm theorem and definition names and a bibliography.

Chapter 2

A Mechanical Theorem Prover

2.1 Overview of Nqthm

Nqthm is the name of both a logic and an associated theorem proving system that is sometimes called the Boyer-Moore theorem prover [10]. A large number of mathematical theorems from many disparate domains have been proved using Nqthm. The Nqthm logic is a quantifier-free, first-order logic resembling Pure Lisp. The user inputs definitions and conjectures to the Nqthm theorem prover which, when successful, outputs a script indicating why each conjecture is a theorem. Proved conjectures are applied in later proofs. In a shallow sense the theorem prover is fully automatic since once a conjecture is input to the theorem prover the proof is uninfluenced by the user. Usually, however, important theorems require the proof and subsequent use of many subsidiary lemmas, so a carefully designed sequence of conjectures is typically needed to lead Nqthm to a non-trivial proof.

A crucial property of Nqthm is soundness. We have confidence that a conjecture that Nqthm accepts is a mathematical truth. As with other proof systems, Nqthm runs on unverified hardware and is not proved correct in a formal way, so a bug could conceivably cause us to conclude that a false conjecture is a theorem. Even so, extensive use of Nqthm over many years suggests that Nqthm-checked proofs are extremely dependable. The use of Nqthm often uncovers mistakes in hand-constructed proofs.

Nqthm provides us the confidence associated with formal proof while shielding us somewhat from the drudgery of writing the complete proof ourselves. Nqthm has been successfully applied in computer systems verification [6]. The requirement imposed by Nqthm to get every detail of a proof exactly correct is very important in computer systems verification as any mistake can be catastrophic. Another reason Nqthm is well-suited for modeling of and proof about computer systems is the executable nature of the Lisp-like Nqthm logic. After one introduces a definition into the Nqthm system, one can "run" it as one might execute any other program. When building models of computer systems upon which to base theorems about computer system correctness, it is very useful to be able to execute the definitions and thereby use the formal model as a system simulator.

Many Nqthm introductions have been written by this author and others. Although we use attribution, parts of this chapter, particularly Sections 2.2 and 2.3, have been influenced by a variety of sources [10, 21, 53].

2.2 A Brief Introduction to the Nqthm Logic

The Nqthm logic is explained precisely in its documentation [10]. We present a brief description because our work makes extensive use of Nqthm, and we wish this dissertation to be as self-contained as possible. The Nqthm logic is a simple quantifier-free, first-order logic resembling the Lisp programming language. Terms in the logic are written using a prefix syntax. The logic is an extension of propositional calculus with variables, function symbols, and the equality relation. Axioms are added defining the following [21]:

- the logical constants TRUE and FALSE, abbreviated **t** and **f**;

- the function 'if' where `(IF x y z)` is z if x is **f** and y otherwise;
- the Boolean connectives 'and', 'or', 'not', and 'implies';
- the 'equal' function where `(EQUAL x y)` is **t** when x is y and **f** otherwise;
- and inductively constructed objects including natural numbers, ordered pairs, and literal atoms.

Some builtin functions are axiomatized in the standard release of Nqthm, including the following functions.

- `(cons x y)` constructs a pair whose elements are `x` and `y`.
- `(listp x)` returns **t** if x is a pair and **f** otherwise.
- `(car x)` returns the first element of x if x is a pair and `0` otherwise.
- `(cdr x)` returns the second element of x if x is a pair and 0 otherwise.
- `(assoc name l)` returns the first e of list l where `(car e)` equals *name* and **f** if no such element exists.

We also use various builtin arithmetical functions on natural numbers, such as 'plus', 'difference', and 'remainder'.

The logic provides a principle of recursive definition under which new function symbols may be introduced in a sound manner. The following, for example, is a definition of a function 'length' that calculates list length.

```
(defn length (x) (if (listp x) (add1 (length (cdr x))) 0))
```

Nqthm provides an execution environment, R-loop, that allows the evaluation of Nqthm-defined functions on constants. For example, if we have defined the function 'length' as suggested above, we can evaluate it on a constant using R-loop.

```
*(length '(a b c))
 3
```

The Nqthm logic is convenient for theorem proving and execution. However, those unfamiliar with it often find the prefix notation opaque. In this dissertation we use a more conventional infix notation when possible. A Lisp program generates infix notation automatically from Nqthm input [11], and the following list excerpted from the program documentation describes the relationship between Nqthm terms and this notation.

1. Variables are printed in italics, unless specified otherwise in Table 2.1.

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 $\mathrm{fn}\,(x,\ y,\ z)$. 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 $\textsc{c}$, 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. Quoted constants are printed in the ordinary syntax of the Nqthm logic, in a 'typewriter font.' For example, `'(a b c)` is still printed just that way. `#b001` is printed as $\texttt{001}_2$, `#o765` is printed as $\texttt{765}_8$, and `#xa9` is printed as $\texttt{a9}_{16}$, representing binary, octal and hexadecimal, respectively.

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. `(not x)` is printed as $\neg\ x$.

8. The remaining symbols that are printed specially are described in Table 2.1, which is taken from the infix program documentation [11].

For example, the definition of 'length' presented previously in the Nqthm logic when presented in infix notation is:

DEFINITION:
length (x)
= **if** listp (x) **then** $1 +$ length (cdr (x))
else 0 **endif**

In the text we refer to Nqthm definitions and theorems by their names surrounded by single quotes. So, the name of a theorem proved using Nqthm is displayed 'example-theorem', and the name of a definition is similarly displayed 'foo'. Variables and constants in the text are presented in the style of the notation described above, so x is a variable and TRUE is a constant.

2.3 Theorem Proving Using Nqthm

We present a brief summary of Nqthm's theorem proving capability. Nqthm prover capabilities are explained precisely in the Nqthm documentation [10] and in a tutorial [32]. A powerful prover has been written for reasoning

Nqthm Syntax	Conventional Syntax
`t`	$\mathbf{t}$
`f`	$\mathbf{f}$
`nil`	$\mathbf{nil}$
`(geq x y)`	$x \geq y$
`(greaterp x y)`	$x > y$
`(leq x y)`	$x \leq y$
`(lessp x y)`	$x < y$
`(or x y)`	$x \vee y$
`(and x y)`	$x \wedge y$
`(times x y)`	$x * y$
`(plus x y)`	$x + y$
`(union x y)`	$x \cup y$
`(remainder x y)`	$x \ \mathbf{mod} \ y$
`(quotient x y)`	$x \div y$
`(difference x y)`	$x - y$
`(iff x y)`	$x \leftrightarrow y$
`(implies x y)`	$x \rightarrow y$
`(member x y)`	$x \in y$
`(equal x y)`	$x = y$
`(not (geq x y))`	$x < y$
`(not (greaterp x y))`	$x \leq y$
`(not (leq x y))`	$x > y$
`(not (lessp x y))`	$x \geq y$
`(not (member x y))`	$x \notin y$
`(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))`	$x \notin \mathbf{N}$

Table 2.1: Symbols used in Nqthm infix notation

about Nqthm expressions. The Nqthm theorem prover takes as input conjectures formalized as terms in the logic and attempts to prove them by repeated transformation and simplification. The theorem prover employs eight basic operations [21]:

- decision procedures for propositional calculus, equality, and linear arithmetic;
- rewriting based on axioms, definitions, and previously-proved lemmas;
- automatic application of user-supplied simplifiers previously proved;
- elimination of calls to certain functions in favor of others that are "better" from a proof perspective;
- heuristic use of equality hypotheses;
- generalization by the replacement of terms by variables;
- elimination of apparently irrelevant hypotheses; and
- mathematical induction.

The theorem prover employs many heuristics to coordinate these basic techniques. The system displays a script of the proof attempt that allows a user to follow the progress of the proof and abort proof attempts gone awry. From the script it is often apparent to the skilled user how to improve the prover's knowledge base so that a subsequent proof attempt will succeed. In a shallow sense, the prover is fully automatic since the system accepts no advice or directives from the user once a proof attempt has started. The only way the user can alter the behavior of the system during a proof attempt is to abort the attempt. However, in a deeper sense, the theorem prover is interactive since the system's behavior is influenced by the database of lemmas formulated by the user and proved by the system. Each conjecture, once proved, may be stored in the prover's database to guide the theorem prover's actions in subsequent proof attempts.

A database is thus more than a logical theory since it is a set of rules for proving theorems in the given theory [21]. The user leads the theorem prover to difficult proofs by programming the rule base. Given a goal theorem, the user generally discovers a proof himself, identifies the key steps in the proof, and then formulates them as lemmas, paying particular attention to their interpretation as rules. The user guides the theorem prover to proofs by the strategic selection of the sequence of theorems to prove and the proper formulation of those theorems. Successful users of the system know how to prove theorems in the logic and understand how the theorem prover interprets them as rules.

2.4 Building Large Proofs with Nqthm

We use the Nqthm logic to specify properties of real-time systems and guide the prover to a proof that an implementation satisfies this specification. Because we have confidence in the soundness of Nqthm this process enhances our confidence in our proofs about real-time systems.

The development of a large proof using Nqthm is a major undertaking. The need to get every detail right in the proof is often very difficult, even for conjectures that are "obviously" true, and it is a common experience among Nqthm users to discover that a conjecture being "proved" with Nqthm is not a theorem. Large proofs require careful strategic thinking from the Nqthm user of two main types: ruthless factoring of the theorem into constituent, easier-to-prove lemmas, and clever design of proved inference rules that allow Nqthm to accomplish much of the reasoning automatically.

The factoring of theorems into lemmas is, of course, also needed with the creation of hand proofs. Whatever its advantages, proof using Nqthm is not easier than informal proofs since for non-trivial theorems the Nqthm user must keep in mind a good informal proof with which to guide his strategic choices while leading Nqthm to a mechanical proof.

Clever design of inference rules is peculiar to the construction of mechanical proofs. The standard Nqthm arithmetic libraries are an example of this kind of design [7, 31, 51]. These libraries contain theorems proved about arithmetic that constitute a strategy for how to divine needed truths about involving arithmetic operations. The libraries have proved useful for accomplishing proofs from a variety of domains that involve arithmetic [20, 51, 52, 58] and we use them in the proofs described in this dissertation. A relatively simple mechanical proof of Matijasevich's lemma – a theorem about Fibonnaci numbers – demonstrates the efficacy of the arithmetic libraries [53]. The sequence of Nqthm conjectures ending in Matijasevich's lemma that is accepted by Nqthm is similar to an informal hand proof in a textbook [24]. Nqthm is able to "fill in the formality" because of the strategy incorporated in the libraries. The same kind of strategy that the arithmetic libraries contain for arithmetic is needed for the problem domain of any non-trivial theorem to be proved using Nqthm.

2.5 The Nqthm Proof-checker Enhancement

A proof-checker enhancement of Nqthm called Pc-Nqthm is available for construction of proofs of theorems expressed in the Nqthm logic [30]. This enhancement allows the user some control over the direction of a proof attempt in addition to the design of the preceding lemmas.

Only one of the hundreds of theorems we prove in this dissertation relies on the Pc-Nqthm enhancements. However, this is a very misleading measure of the worth of Pc-Nqthm in our proof efforts. We have found Pc-Nqthm to be a valuable tool during the development of these proofs and use it extensively. Often, when a proof attempt fails, the finer control afforded by Pc-Nqthm allows easier diagnosis of the problem. After the problem is identified and fixed the proof checker enhancements are usually not needed to process the theorem. Undisciplined use of Pc-Nqthm can interfere with proof development because ultimate success with Nqthm or Pc-Nqthm depends on effective rule construction and reliance on "micro-managing" the proof using the Pc-Nqthm enhancements can distract the user. However, our experience is that disciplined use of Pc-Nqthm enhancements aids proof development.

Chapter 3

An Optimal Real-time Scheduling Policy

3.1 Introduction

Real-time applications often have several required functions with different timing constraints. In a seminal paper for building real-time systems, Liu and Layland introduce abstractions that facilitate real-time application development [34]. Using a simple computation model, they exhibit different real-time scheduling policies that are used to choose which of an application's various tasks to assign a processor and argue that these policies have certain useful properties. One scheduling policy is earliest-deadline-first (EDF), which assigns the processor to a task that has earliest deadline among the tasks that are currently running. An EDF scheduler is optimal for tasks requested periodically in the sense that if any schedule can meet the timing constraints on the requests then an EDF schedule will. The original proof of this theorem contained flaws [34], but a subsequent published proof developed independently of the machine-checked proof presented in this chapter appears correct [61]. EDF schedulers are currently rarely used, apparently because developers prefer static priority scheduling policies where tasks do not change priority so that they can ensure that the most crucial tasks always meet their deadlines. (There is a popular misconception that dynamic priority schedulers require far more overhead which may also partly explain the relative unpopularity of EDF schedulers.) An application proved never to miss a deadline may be easier to develop using an EDF scheduler.

Formal mathematics, where proofs are constructed in a well-defined proof system and no step in the proof is skipped, is harder to use than standard, informal mathematics, but it allows mechanical checking. A proof checked mechanically by a trusted proof-checking program is very dependable. Nqthm is such a mechanical theorem prover [10] and is described in Chapter 2. This chapter describes an Nqthm-checked proof of the EDF optimality theorem. The proof uses an approach suggested by Al Mok [35]. Section 3.2 presents the Nqthm definitions related to EDF schedules and periodic tasks used in the theorems. Section 3.3 presents the optimality theorem and an example of its application. The proof of EDF optimality is described in Section 3.4. A complete list of the definitions and theorems that lead Nqthm to a proof of the optimality theorem is presented in Appendix C.

3.2 Some Real-Time Scheduling Definitions

In this section we present some definitions in the Nqthm logic related to the scheduling of a processor to the tasks of a real-time application. The abstractions defined in this section are similar to those in Liu and Layland [34]. A real-time application is composed of tasks each with a unique name. Occasionally there is a request for a task, which has four elements: the requested task's name, the time of the task request, the task request deadline, and the task's duration. Time is modeled with discrete units, tasks are interruptible, and we ignore overhead for switching between tasks.

A schedule is a list that represents a processor schedule assignment over time. A task name tk at location n in a schedule signifies that the processor is assigned to task tk at time n, and **nil** signifies that no task is assigned. A schedule fulfills a task request if during the period no earlier than the task request time and before the task request deadline it contains the task name a number of times equal to the requested task duration. A schedule *overflows* for a set of task requests if some task request is not fulfilled in the schedule. A good schedule never overflows for a set of task requests.

The term firstn(n, l) is the first n elements of list l, nthcdr(n, l) returns all but the first n elements of list l, and occurrences(v, l) returns the number of occurrences of value v in list l. Each of the functions 'name', 'request-time', 'deadline', and 'duration' when applied to a task request returns the appropriate value associated with that request.

The property of being a good schedule with respect to a request list is formalized in the Nqthm logic.

DEFINITION:
good-schedule(s, r)
$=$ **if** listp(r)
then (occurrences (name (car(r)),
firstn (deadline (car(r)) $-$ request-time (car(r)),
nthcdr (request-time (car(r)), s)))
$=$ duration (car(r)))
$\wedge$ good-schedule$(s$, cdr$(r))$
else t endif

Since we are using Nqthm's definition facility our definitions are executable. In the following examples 'good-schedule' is applied to two example schedules and lists of task requests. Task requests are represented by a list containing in order the task name, request time, deadline, and duration.

```
*(GOOD-SCHEDULE '(A NIL C A B A B C NIL NIL)
                '((A 0 7 3) (B 2 6 2) (C 1 5 1) (C 7 8 1)))
 F
*(GOOD-SCHEDULE '(A NIL C A B B A C NIL NIL)
                '((A 0 7 3) (B 2 6 2) (C 1 5 1) (C 7 8 1)))
 T
```

The term active-task-requests $(time,\ r)$ is the list of task requests in r for which $time$ is at least the task request time and is less than the task request deadline.

DEFINITION:
active-task-requests $(time,\ r)$
$=$ **if** listp (r)
then if $(time <$ deadline (car $(r))) \wedge (time \geq$ request-time (car $(r)))$
then cons (car (r), active-task-requests $(time$, cdr $(r)))$
else active-task-requests $(time$, cdr $(r))$ **endif**
else nil endif

```
*(ACTIVE-TASK-REQUESTS 3 '((A 0 7 3) (B 2 6 2) (C 1 5 1) (C 7 8 1)))
 '((A 0 7 3) (B 2 6 2) (C 1 5 1))
```

The term unfulfilled $(time,\ s,\ r)$ is a list of requests in r for which the task duration is not equal to the number of occurrences of the task name in s before $time$ but no earlier than the task request time.

DEFINITION:
unfulfilled $(time,\ s,\ r)$
$=$ **if** listp (r)
then if occurrences (name (car (r)),
firstn $(time -$ request-time (car (r)),
nthcdr (request-time (car (r)), s)))
$=$ duration (car (r)) **then** unfulfilled $(time,\ s$, cdr $(r))$
else cons (car (r), unfulfilled $(time,\ s$, cdr $(r)))$ **endif**
else nil endif

```
*(UNFULFILLED 3 '(A NIL C A B B A C NIL NIL)
                '((A 0 7 3) (B 2 6 2) (C 1 5 1)))
 '((A 0 7 3) (B 2 6 2))
```

The term least-deadline (r) is a request in r with least deadline.

DEFINITION:
least-deadline (r)
$=$ **if** listp (r)
then if listp (cdr (r))
then if deadline (car (r)) $<$ deadline (car (cdr (r)))
then least-deadline (cons (car (r), cdr (cdr (r))))
else least-deadline (cdr (r)) **endif**
else car (r) **endif**
else nil endif

```
*(LEAST-DEADLINE '((A 0 7 3) (B 2 6 2)))
 '(B 2 6 2)
```

The term edf(n, r) represents a schedule of length n for requests r such that an unfulfilled active task request with least deadline is chosen at each location in the schedule if one exists or **nil** if there is no such request.

DEFINITION:
edf(*lengthschedule*, r)
= **if** *lengthschedule* $\simeq$ 0 **then nil**
  else let s **be** edf(*lengthschedule* $-$ 1, r)
    in
    let *unfulfilled* **be** unfulfilled(*lengthschedule* $-$ 1, s,
                  active-task-requests(*lengthschedule* $-$ 1, r))
    in
    if listp(*unfulfilled*)
    then append(s, list(name(least-deadline(*unfulfilled*))))
    else append(s, list(**nil**)) **endif endlet endlet endif**

```
*(EDF 10 '((A 0 7 3) (B 2 6 2) (C 1 5 1) (C 7 8 1)))
 '(A C B B A A NIL C NIL NIL)
```

EDF scheduling appears to be a sensible way to schedule task requests. Note that the EDF scheduler applied to the task requests of the previous example generates a good schedule.

```
*(GOOD-SCHEDULE '(A C B B A A NIL C NIL NIL)
                '((A 0 7 3) (B 2 6 2) (C 1 5 1) (C 7 8 1)))
 T
```

Our model for tasks is that they are *periodic*, meaning that task requests for a particular task occur periodically. We assume the duration of the task requests for a particular task is constant. Periodic tasks have unique non-nil names and non-zero periods and durations. A valid list of periodic tasks is identified by 'periodic-tasksp'.

```
*(PERIODIC-TASKSP '((A 3 2) (B 0 3)))
 F
*(PERIODIC-TASKSP '((A 3 2) (B 9 3)))
 T
```

The term periodic-tasks-requests (pts, $n1$, $n2$) generates task requests from periodic tasks list pts. A request is generated for each periodic task at every time less than $n2$ that is the sum of $n1$ and a multiple of the task's period.

DEFINITION:
periodic-task-requests (*pt*, *starting-time*, *ending-time*)
= **if** periodic-taskp (*pt*)
  then if *starting-time* < *ending-time*
    then cons (list (tk-name (*pt*), *starting-time*,
        starting-time + tk-period (*pt*), tk-duration (*pt*)),
      periodic-task-requests (*pt*, *starting-time* + tk-period (*pt*),
        ending-time))
    else nil endif
  else nil endif

DEFINITION:
periodic-tasks-requests (*pts*, *starting-time*, *ending-time*)
= **if** periodic-tasksp (*pts*)
  then if listp (*pts*)
    then append (periodic-task-requests (car (*pts*), *starting-time*, *ending-time*),
      periodic-tasks-requests (cdr (*pts*), *starting-time*, *ending-time*))
    else nil endif
  else nil endif

```
*(PERIODIC-TASKS-REQUESTS '((A 3 2) (B 9 3)) 0 18)
 '((A 0 3 2) (A 3 6 2) (A 6 9 2) (A 9 12 2)
   (A 12 15 2) (A 15 18 2) (B 0 9 3) (B 9 18 3))
```

Nqthm has been used to reason about rationals [51], but it is simpler when using Nqthm to stay in the realm of naturals. We introduce a function that we use to express the desired theorem using only natural number arithmetic. The term big-period (pts) is the product of the task periods in pts. The term cpu-utilization (pts, n) is the sum of ($n * duration$) $\div$ $period$ for each task in pts.

DEFINITION:
big-period (pts)
$=$ **if** listp (pts) **then** tk-period (car (pts)) $*$ big-period (cdr (pts))
else 1 **endif**

DEFINITION:
cpu-utilization (pts, $bigp$)
$=$ **if** listp (pts)
then (($bigp$ $*$ tk-duration (car (pts))) $\div$ tk-period (car (pts)))
$+$ cpu-utilization (cdr (pts), $bigp$)
else 0 **endif**

```
*(BIG-PERIOD '((A 3 2) (B 9 3)))
 27
*(CPU-UTILIZATION '((A 3 2) (B 9 3)) 27)
 27
```

3.3 EDF Optimality for Periodic Tasks

We present the optimality theorem about EDF schedules on periodic tasks using the definitions of the previous section. If pts is a list of periodic tasks such that $\sum_{tasks} duration/period \leq 1$ and n is a multiple of big-period (pts), then an EDF schedule satisfies the requests of pts through n time units.

((big-period (pts) $\geq$ cpu-utilization (pts, big-period (pts)))
$\wedge$ periodic-tasksp (pts)
$\wedge$ ((n **mod** big-period (pts)) $=$ 0))
$\rightarrow$ good-schedule (edf (n, periodic-tasks-requests (pts, 0, n)),
periodic-tasks-requests (pts, 0, n))

We call this an optimality theorem since the theorem shows that any set of periodic tasks for which there exists a good schedule can be scheduled using an EDF scheduler, and prove this theorem using Nqthm as described in Section 3.4. We illustrate the application of this remarkable theorem on a small example below. Assume task A has period 16 and duration 4, task B has period 5 and duration 2, and task C has period 3 and duration 1. For the first 240 time slots we show that the hypotheses of the theorem hold and, as guaranteed by the theorem, the conclusion holds. So, letting n be 240 and pts be the tasks described above, we evaluate each of the hypotheses and demonstrate that each is satisfied.

```
*(BIG-PERIOD PTS)
 240
*(CPU-UTILIZATION PTS 240)
 236
*(NOT (LESSP (BIG-PERIOD PTS) (CPU-UTILIZATION PTS (BIG-PERIOD PTS))))
 T
*(PERIODIC-TASKSP PTS)
 T
*(EQUAL (REMAINDER N (BIG-PERIOD PTS)) 0)
 T
```

Since the theorem is proved (as described in Section 3.4) and the hypotheses are satisfied, the EDF schedule is a good schedule for the generated task requests of this example. This is demonstrated by executing the conclusion of the theorem on the example.

```
*(GOOD-SCHEDULE
    (EDF N (PERIODIC-TASKS-REQUESTS PTS 0 N))
    (PERIODIC-TASKS-REQUESTS PTS 0 N))
 T
*
```

The first few assignments of the EDF schedule are calculated below, and the beginning part of the schedule is displayed pictorially in Figure 3.1.

```
*(EDF N (PERIODIC-TASKS-REQUESTS PTS 0 N))
 '(C B B C A B C B A C B B C A A C B B C A B C B A C B B C A
     A C B B C A B C B A C B B C A A C B B C A B C B A C B B
     C A A C B B C A B C B A C B B C A A C B B C NIL B C B A
     ....
```

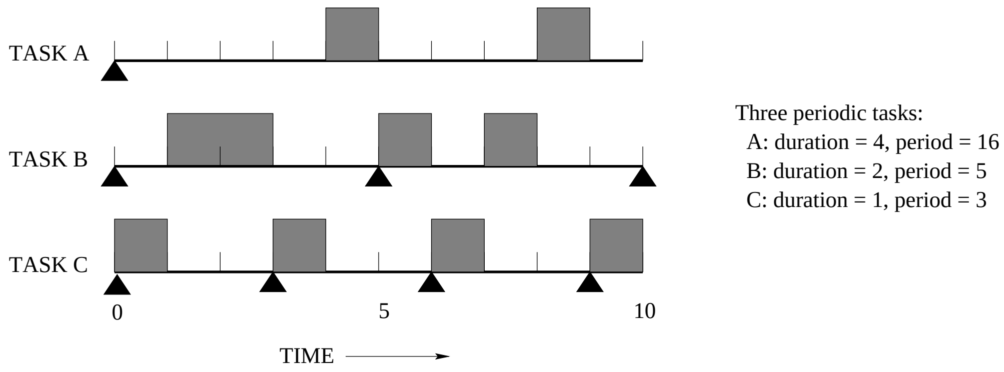

Figure 3.1: An example EDF schedule

3.4 EDF Optimality Proof

In this section we describe the proof of EDF optimality. We first present an informal proof of the optimality theorem presented in Section 3.3. The approach used to prove this theorem was suggested by Al Mok [35]. We present it to help motivate the Nqthm proof that uses the same basic approach and to contrast the informal proof with the formal proof. We then outline the formal proof of the optimality theorem by presenting some of its key lemmas. We discuss why it is harder to construct proofs that are checked with Nqthm and give some statistics about the proof development.

3.4.1 An Informal Proof of EDF Optimality

Theorem: Let R be a periodic request history corresponding to a periodic set P of m tasks. Let the list $(tk_i,\ d_i,\ p_i)$ denote an element of P. $\sum_{i=1}^{m}(d_i/p_i) \leq 1$ implies that an EDF schedule of R has no overflow.

Proof: Let c be the product of the periods of P. Let P' be identical to P except let each duration d'_i and period p'_i in P be c times the corresponding values d_i and p_i in P. Let R' be a task request history of P'.

Let S' be a schedule for R' by repeating length(R) times the following:

$$(tk_1)^{d'_1/p_1}\ (tk_2)^{d'_2/p_2}\ \ldots (tk_m)^{d'_m/p_m}\ \mathbf{nil}^{c-\sum d'_i/p_i}$$

Since $\sum_{i=1}^{m}(d_i/p_i) \leq 1$, length$(R)$*$c$ is the length of S' and S' does not overflow for R'.

We define swap(t, S, R) for time t, schedule S, and request history R as follows. Let *tk* be the least task with earliest absolute deadline unfulfilled at t. Return a schedule identical to S except that the task name at location t and the task name at the location of the first occurrence of *tk* no earlier than t are swapped. If schedule S does not overflow for request history R, then neither does swap(i, S, R).

Let $S'_0 = S'$ and $S'_{n+1} =$ swap(n, S'_n, R'). Let $S'' = S_{length(S')'}$. Note that S'' does not overflow for R' and that the element n in S'' is equal to the $n - (n \textbf{ mod } c)$ element of S''. Let S''' be composed of the elements of S'' in locations that are multiples of c. S''' is the earliest deadline first schedule of R and does not overflow.

Q.E.D.

3.4.2 Outline of Nqthm Proof

We present in this section the machine-checked proof of EDF optimality, including the main theorem and several of the key lemmas. The theorem and each of the supporting lemmas has been proved using Nqthm. The Nqthm proof is presented fully in Appendix C, so a reader who believes that Nqthm is sound — that is, that Nqthm accepts only theorems — and who is interested only in the reliability of the final theorem might well skip this section. We hope that the following outline of the Nqthm proof makes understanding it easier, but it does not add to the high degree of confidence we have that the final proved conjecture is in fact a theorem.

For each theorem presented we informally characterize some of the definitions and relate the lemma to the informal proof of the previous section.

THEOREM: good-simple-schedule

$$
\begin{array}{ll}
\multicolumn{2}{l}{((\mathit{bigp} \geq \text{length}\,(\text{substring-schedule}\,(\mathit{pts},\ \mathit{bigp})))} \\
\land & \text{periodic-tasksp}\,(\mathit{pts}) \\
\land & \text{expanded-tasksp}\,(\mathit{pts},\ \mathit{bigp}) \\
\land & (0 < \mathit{bigp}) \\
\land & ((n \ \mathbf{mod}\ \mathit{bigp}) \simeq 0) \\
\land & ((n \ \mathbf{mod}\ \text{big-period}\,(\mathit{pts})) \simeq 0)) \\
\rightarrow & \text{good-schedule}\,(\text{make-simple-schedule}\,(\mathit{pts},\ \mathit{bigp},\ n), \\
& \qquad\qquad\quad \text{periodic-tasks-requests}\,(\mathit{pts},\ 0,\ n))
\end{array}
$$

The predicate expanded-tasksp (pts, $bigp$) holds when each periodic task in pts has a duration and period that are multiples of $bigp$ and $bigp$ * $duration$ is a multiple of period. The term substring-schedule (pts) is a list of task names from pts such that the number of occurrences of each task is equal to its original duration. The term make-simple-schedule (pts, $bigp$, n) generates a schedule for pts as S' in the informal proof of length n with repeated segments of length $bigp$.

Theorem 'good-simple-schedule' corresponds to the statement in the informal optimality proof that S' does not overflow for R'.

THEOREM: make-schedule-edf-preserves-good-schedule

$$
\begin{array}{ll}
\multicolumn{2}{l}{(\text{non-overlapping-requests}\,(r)} \\
\land & \text{cars-non-nil-litatoms}\,(r) \\
\land & \text{all-litatoms}\,(s) \\
\land & \text{all-non-nil-corresponding}\,(s,\ r,\ 0) \\
\land & \text{all-nils-or-cars}\,(s,\ \mathit{pts}) \\
\land & \text{good-schedule}\,(s,\ r) \\
\land & (r = \text{periodic-tasks-requests}\,(\mathit{pts},\ 0,\ \text{length}\,(s))) \\
\land & \text{periodic-tasksp}\,(\mathit{pts})) \\
\rightarrow & \text{good-schedule}\,(\text{make-schedule-edf}\,(s,\ r,\ n),\ r)
\end{array}
$$

The term non-overlapping-requests (r) holds when no two distinct requests in r have identical task names and overlapping start time/deadline time windows. The term all-non-nil-corresponding $(s,\ r,\ n)$ holds when all non-NIL elements of schedule s that are at location no less than n have corresponding request in r. The term make-schedule-edf $(s,\ r,\ n)$ is a schedule in which starting at location n to the end of schedule s the assignment at each location is swapped with an assignment no earlier that has earliest deadline.

The combination of 'make-schedule-edf-preserves-good-schedule' and 'good-simple-schedule' correspond to the statement in the informal optimality proof that S'' does not overflow for R'.

THEOREM: make-schedule-edf-is-edf
(periodic-tasksp (*pts*)
 ∧ cars-non-nil-litatoms (periodic-tasks-requests (*pts*, 0, length (*s*)))
 ∧ all-litatoms (*s*)
 ∧ good-schedule (*s*, periodic-tasks-requests (*pts*, 0, length (*s*)))
 ∧ (plist (*s*) = *s*)
 ∧ all-non-nil-corresponding (*s*, periodic-tasks-requests (*pts*, 0, length (*s*)), 0)
 ∧ all-nils-or-cars (*s*, *pts*)
 ∧ periodic-tasksp (*pts*)
 ∧ (*n* = length (*s*)))
→ (make-schedule-edf (*s*, periodic-tasks-requests (*pts*, 0, *n*), 0)
  = edf (length (*s*), periodic-tasks-requests (*pts*, 0, length (*s*))))

Theorem 'make-schedule-edf-is-edf' corresponds to the idea implicit in the informal proof that from its construction S'' is an EDF schedule.

THEOREM: edf-simple-expand-tasks-requests
((1 < *bigp*) ∧ non-overlapping-requests (*r*) ∧ valid-requests (*r*))
→ (edf (*n*, expand-tasks-requests (*r*, *bigp*))
  = append (expand-list (*bigp*, edf (*n* ÷ *bigp*, *r*)),
   let *undone* **be** unfulfilled (*n* ÷ *bigp*,
     edf (*n* ÷ *bigp*, *r*),
     active-task-requests (*n* ÷ *bigp*, *r*))
   in
   repeat (*n* **mod** *bigp*,
    if listp (*undone*) **then** car (least-deadline (*undone*))
    else nil endif) **endlet**))

The term expand-tasks-requests $(r,\ bigp)$ denotes a list of task requests identical to r except that each request's start time, deadline, and duration are multiplied by $bigp$. The term expand-list $(n,\ l)$ returns a list of length $n *$ length (l) where each element in location p has the value of element $p \div n$ in l.

Note the special case of 'edf-simple-expand-tasks-requests' where n is a multiple of $bigp$. This case corresponds to the notion in the informal proof that element n in S'' is equal to the $n - (n \textbf{ mod } c)$ element of S''.

Finally, we present the main theorem whose proof establishes the optimality of the EDF scheduler.

THEOREM: good-edf-periodic
((big-period $(pts) \geq$ cpu-utilization $(pts,$ big-period $(pts)))$
$\wedge$ periodic-tasksp (pts)
$\wedge$ $((n \textbf{ mod }$ big-period $(pts)) = 0))$
$\rightarrow$ good-schedule (edf $(n,$ periodic-tasks-requests $(pts,\ 0,\ n)),$
periodic-tasks-requests $(pts,\ 0,\ n))$

The EDF optimality theorem is a consequence of the earlier lemmas and many other easier-to-prove lemmas.

3.4.3 Why the Nqthm Proof Requires Much Effort

The Nqthm proof of EDF optimality consists of the approximately 500 Nqthm "events" listed in Appendix C. Most of these are lemmas that are stated and proved in order to guide Nqthm to the proof of later theorems, and this listing does not include the standard library of arithmetic theorems upon which this proof depends [53]. The proof took approximately 3 months for the author to construct. Nqthm requires approximately 2.5 hours to process the events on an unloaded Sun Sparcstation IPC and generates an 8 Megabyte file containing an English-language justification. A comparison between the informal proof in Section 3.4.1 and the Nqthm proof makes it obvious that doing the proof with Nqthm requires a much greater effort than doing an informal hand proof. Is the extra effort really needed because formal proof is difficult, or is the difference simply due to difficulty using Nqthm?

We believe we faced great difficulties proving this theorem using Nqthm because of the precision formal mathematics requires. Nqthm merely enforces the requirement that we be precise and not skip any details of the proof. On several occasions during the development of the proof the final theorem appeared nearly proved but was not because the remaining "simple" lemmas when done formally were difficult to prove. The complexity of doing formal mathematics is apparent in the complexity of the lemmas in Section 3.4.2. Formal mathematics requires we be precise, and Nqthm enforces this requirement.

Many of the concepts used in the proof of this theorem would not be introduced in an informal proof but are needed in order to do a formal proof. As an example we examine 'firstn', which returns the first n elements of a list. This concept is defined in the Nqthm proof of the EDF optimality theorem because it is needed to define precisely some of the notions in the proof, but a mathematician not doing formal proofs would not trouble himself to define it explicitly. In the Nqthm logic we define this concept as follows.

DEFINITION:
firstn (n, *list*)
= **if** $n \simeq 0$ **then nil**
else cons (car (*list*), firstn ($n - 1$, cdr (*list*))) **endif**

In order for Nqthm to reason about the theorems we want to prove involving schedules and task requests, many theorems must be proved that "program" the theorem prover in a way that allows it to reason effectively

1. $\text{length}(\text{firstn}(n,\ \mathit{list})) = \text{fix}(n)$

2. $\text{firstn}(n,\ \text{append}(\mathit{l1},\ \mathit{l2}))$
$= \ \textbf{if}\ \text{length}(\mathit{l1}) < n\ \textbf{then}\ \text{append}(\mathit{l1},\ \text{firstn}(n - \text{length}(\mathit{l1}),\ \mathit{l2}))$
$\quad \textbf{else}\ \text{firstn}(n,\ \mathit{l1})\ \textbf{endif}$

3. $\text{firstn}(\text{length}(x),\ x) = \text{plist}(x)$

4. $\text{plist}(\text{firstn}(n,\ l)) = \text{firstn}(n,\ l)$

5. $\text{nthcdr}(n,\ \text{firstn}(n + x,\ s)) = \text{firstn}(x,\ \text{nthcdr}(n,\ s))$

6. $\text{firstn}(n,\ \text{replace-nth}(i,\ v,\ l))$
$= \ \textbf{if}\ i < n\ \textbf{then}\ \text{replace-nth}(i,\ v,\ \text{firstn}(n,\ l))$
$\quad \textbf{else}\ \text{firstn}(n,\ l)\ \textbf{endif}$

7. $\text{nth}(n,\ \text{firstn}(\mathit{n2},\ s))$
$= \ \textbf{if}\ n < \mathit{n2}\ \textbf{then}\ \text{nth}(n,\ s)$
$\quad \textbf{else}\ \mathsf{0}\ \textbf{endif}$

8. $\text{firstn}(a,\ \text{firstn}(b,\ x))$
$= \ \textbf{if}\ b < a\ \textbf{then}\ \text{append}(\text{firstn}(b,\ x),\ \text{repeat}(a - b,\ \mathsf{0}))$
$\quad \textbf{else}\ \text{firstn}(a,\ x)\ \textbf{endif}$

9. $\text{firstn}(\mathsf{1},\ s) = \text{list}(\text{car}(s))$

10. $(i < j) \rightarrow (\text{nth}(i,\ s) \in \text{firstn}(j,\ s))$

11. $(\text{car}(x) \in \text{firstn}(n,\ x)) = (n \not\simeq \mathsf{0})$

12. $((i \geq y) \wedge (i < (x + y))) \rightarrow (\text{nth}(i,\ s) \in \text{firstn}(x,\ \text{nthcdr}(y,\ s)))$

13. $(x \notin l) \rightarrow ((x \in \text{firstn}(n,\ l)) = ((x = \mathsf{0}) \wedge (\text{length}(l) < n)))$

14. $\text{firstn}(\mathit{n1},\ \text{repeat}(\mathit{n2},\ v))$
$= \ \textbf{if}\ \mathit{n2} < \mathit{n1}\ \textbf{then}\ \text{append}(\text{repeat}(\mathit{n2},\ v),\ \text{repeat}(\mathit{n1} - \mathit{n2},\ \mathsf{0}))$
$\quad \textbf{else}\ \text{repeat}(\mathit{n1},\ v)\ \textbf{endif}$

15. $\text{nthcdr}(n,\ \text{firstn}(n,\ l)) = \textbf{nil}$

16. $\text{listp}(\text{firstn}(n,\ l)) = (n \not\simeq \mathsf{0})$

17. $(\textbf{nil} = \text{firstn}(n,\ l)) = (n \simeq \mathsf{0})$

18. $n \geq \text{occurrences}(v,\ \text{firstn}(n,\ l))$

19. $\text{firstn}(a + b,\ l) = \text{append}(\text{firstn}(b,\ l),\ \text{firstn}(a,\ \text{nthcdr}(b,\ l)))$

20. $(\text{length}(l) = \text{fix}(n)) \rightarrow (\text{firstn}(n,\ l) = \text{plist}(l))$

Figure 3.2: Some theorems proved about 'firstn'

about them. Figure 3.2 presents some of the theorems involving 'firstn' that were proved in order to prove EDF optimality. It is not an exhaustive list, but it includes theorems that are needed in a formal proof involving 'firstn' but which probably would be skipped in an informal proof. Each of these lemmas is proved using Nqthm and is applied subsequently in theorems involving 'firstn'. We believe that these facts would be needed in a formal proof involving 'firstn' whether or not one is using Nqthm. If one wishes to prove formally something that involves functions 'firstn' and 'append', for example, there is no getting around the requirement to prove lemma (2) in Figure 3.2. An Nqthm user is forced to prove something akin to this theorem as a part of the proof of EDF optimality only because any formal proof of EDF optimality requires it. Many of the facts listed above about 'firstn' would not be stated, let alone proved, in an informal proof.

Lemmas like these about 'firstn' can be reused in future Nqthm proofs that involve 'firstn' because the collection of theorems constitutes a strategy for proving theorems. In our proof of this theorem, for example, we take advantage of the libraries of previously-proved arithmetic facts described briefly in Section 2.4 that have been developed for Nqthm-checked proofs of other theorems [53]. In contrast, the many theorems we have proved involving earliest-deadline-first scheduling are probably not applicable in other domains.

It is our belief that for the most part the difficulties we faced in the construction of an Nqthm proof of this theorem result from the complexity and precision of machine-checked mathematics rather than a deficiency in Nqthm.

3.4.4 Advantages of Machine-Checked Proof

As can be seen in the earlier sections, the machine-checked proof is more complex than the informal proof. It nevertheless has several advantages.

We are more sure of the correctness of the proof because it is machine-checked. Informal proofs often make intuitive leaps that are difficult to justify. As mentioned previously, the original proof of the EDF optimality theorem was flawed because such a leap in the informal proof was not valid. The discipline of proving things formally doesn't allow any steps to be skipped that might lead to a mistake. A theorem with a machine-checked proof can be relied upon to be a theorem, at least to the extent that the machine is trusted. Although formal proofs are more detailed than informal proofs, machine-checked formal proofs need not be understood by someone whose only interest is that an alleged theorem is in fact true. With regard to the EDF optimality theorem, for example, if one trusts Nqthm then one need not understand the details of the proof to believe it is a theorem.

Proofs of this kind lead to more-precise statements of theorems. Often the work of proving a theorem formally leads to explicit listing of hypotheses that would be omitted and precise definitions of underlying concepts. Many of the lemmas proved during the proof outlined in Section 3.4.2, for example, have hypotheses that are not explicit in the informal proof but which are needed for the lemma to be true. Also, the formal proof forced us to define all the subsidiary functions, which we would not have otherwise done. This kind of precision is important in computer systems verification.

Informal proofs can be tedious when done formally. Even so, formal proofs that are machine-checked provide a very high level of assurance about the validity of a conjecture and provide a reliable basis for other formal work.

Using an automated proof system such as Nqthm, the development of formal proofs of non-trivial theorems with high confidence is feasible.

Chapter 4

A Proved Application with Simple Real-time Properties

4.1 Introduction

As suggested in Chapter 1, computer programs and the systems software and hardware that support them can be very complex. One approach to developing dependable software is to formalize a problem specification in a logic and prove that a particular program written in a computer language meets the requirements of the problem specification. Of course, formal reasoning about program execution requires a formal semantics of the language in which the program is expressed. Ideally, the language semantics will come with a verified compiler targeted to hardware whose design has been proved to work to specification. Extra reliability can be attained by checking the proofs with a trustworthy automatic theorem prover.

In this chapter we describe the verified "short stack" that includes a verified compiler for the programming language Piton [38]. We discuss the proofs of that compiler and the design of the FM9001 microprocessor [14] in Section 4.2. In Section 4.4 we develop a specification for a program that plays nim, a centuries-old game introduced in Section 4.3. Section 4.5 presents an algorithm and Piton implementation of this program. The development of an Nqthm-checked proof that the program meets the requirements outlined in Section 4.4 is discussed in Section 4.6. This work is also reported in [56].

4.2 Background

Proofs about computer systems are often mind-numbingly complex but not particularly deep, which is to say perfectly suited to automatic generation. The requirement imposed by Nqthm to get every detail of a proof exactly correct is very important in computer systems verification as even the most trivial-seeming mistake can lead to catastrophe. Piton is a computer language for which there is a compiler and associated machine-checked compiler correctness theorem [38]. A formal semantics for Piton, a formal description of the FM9001 microprocessor, and the Piton compiler are introduced as Nqthm functions. The compiler correctness theorem relates the data values expected after running a Piton program using Piton's formal semantics to the data values computed by the FM9001 running a compiled Piton program.

The semantics of Piton is described using an interpreter function. A Piton state consists of elements that constitute a programmer's model of how Piton executes: a list of Piton programs to run, a user-addressable stack, a current instruction pointer, a subroutine calling stack, a data area, the word size, stack size limits, and a program status flag. The interpreter function 'p' takes as arguments a Piton state and the number of Piton instructions to run, and returns the Piton state resulting from the computation. The semantics of FM9001 is also described using an interpreter function. An FM9001 state consists of elements that constitute a programmer's model for the FM9001 design: values for the register file, the condition flags, and the memory. The FM9001 interpreter function takes as arguments an FM9001 state and the number of FM9001 instructions to run and returns the FM9001 state resulting from the computation.

The Piton correctness theorem is suggested by Figure 4.1. The func-

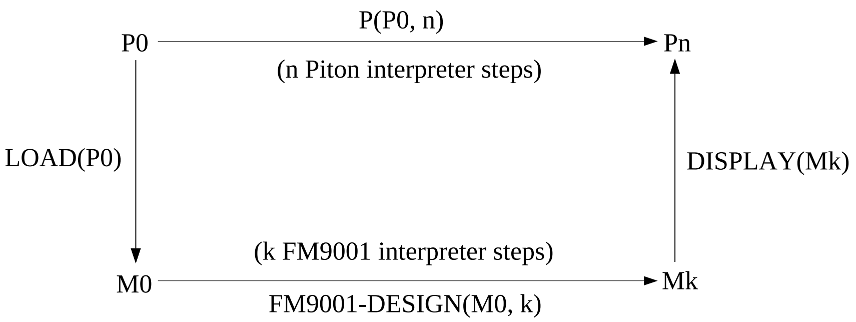

Figure 4.1: The Piton correctness proof.

tion 'load' is the Piton compiler, and the function 'display' extracts a Piton data segment from an FM9001 image. The Piton compiler is a cross-compiler since the compiler is a function in the Nqthm logic that produces code for another processor, the FM9001. Roughly speaking, the Piton compiler correctness theorem guarantees that the data segment calculated by a Piton program running on the Piton interpreter and the data segment calculated by running a compiled Piton program on the FM9001 are equivalent. The interpreter function serves as a precise specification for the expected behavior of a system component. This general approach to system verification has been used on other projects [6, 18]. Interpreter functions can be complex: Piton has 71 instructions and some high-level features, and the definition of 'p' in the Nqthm logic requires about 50 pages.

The FM9001 [27] is a fairly conventional microprocessor with an instruction set somewhat like that on a PDP-11. Unlike most processors, however, the FM9001 has been specified, designed, and proved correct in the sense that the design is shown to meet the specification. The same specification was also used as the target machine in the Piton compiler correctness proof, so the proofs can be "stacked" and we need only assume that the hardware is working properly for a Piton program compiled onto the FM9001 to work according the formal semantics of Piton. The FM9001 has been fabricated and used to run

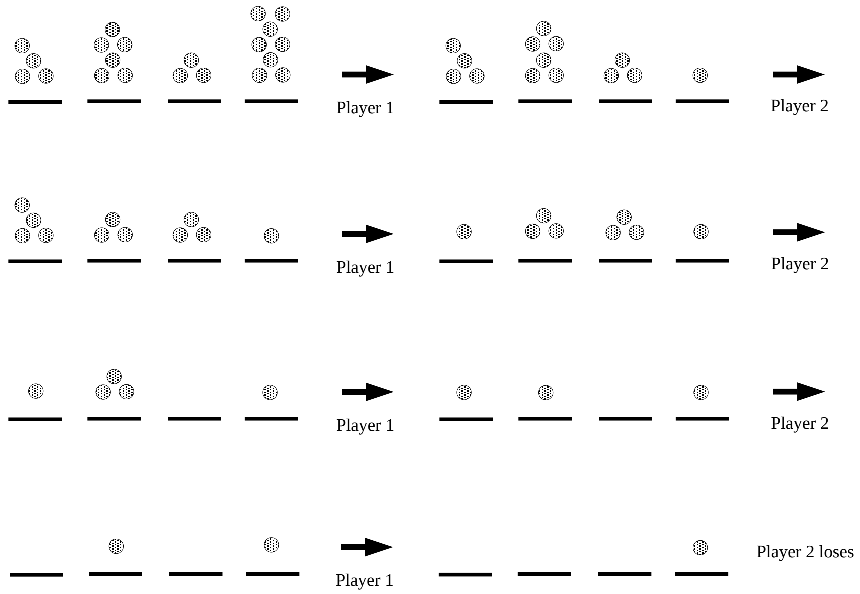

Figure 4.2: An example nim game

programs, including the compiled Piton program described in this chapter.

In short, formalized in Nqthm are a language semantics for Piton, a verified compiler to the FM9001, a verified design for the FM9001, and a fabricated chip that implements the FM9001 design. The rest of this paper describes the development of a verified application that runs in this environment.

4.3 Nim

Nim is one of the oldest and one of the most engaging mathematical games [19]. The game is played with piles of stones and two players who alternate turns. On his turn a player removes at least one stone from exactly one pile. The player who removes the final stone loses. Figure 4.2 shows an example nim game.

A nim state consists of a list of numbers that represents the number of stones in each of the piles. A *strategy* is a function that maps non-empty nim states to nim states in a manner consistent with the notion of a legal nim move. A *winning strategy* is a strategy that guarantees for a particular nim state that the player who is about to take a turn will win. An *optimal move* for a particular non-empty nim state is the first step in some winning strategy if one exists, or any legal move otherwise.

Sometimes there is no winning strategy. For example, if a player has to take a turn and there are two piles left each with two stones, there will be a winning strategy for his opponent on the next move regardless of the move he makes, and thus there is no winning strategy for the player and all legal moves are optimal. Sometimes there is a move a player can make that will cause his opponent in his next turn to have no winning strategy. For example, if the player is left with two piles with 4 and 2 stones respectively, he can remove 2 stones from the 4 pile and leave his opponent in the 2 piles of 2 situation. Thus, there is a winning strategy when there exists a move that results in a state from which there is no winning strategy.

Since for any non-empty nim state there either is one stone left, or there is a winning strategy for the next player, or there will be a winning strategy for the opponent on his next move, we can search for an optimal move from any nim state. Figure 4.3 depicts the search tree of the search for an optimal move from a state. The nodes in the tree in bold type are *losing states* from which there is no winning strategy.

Blind search is not a practical approach for developing a Nim-playing program. But it is useful for describing optimal Nim play, and we formalize it in the Nqthm for use in our specification. We implement the search idea with function 'wsp', which takes two arguments representing a nim state or list of nim states and a flag signifying the type of the first argument. The definition

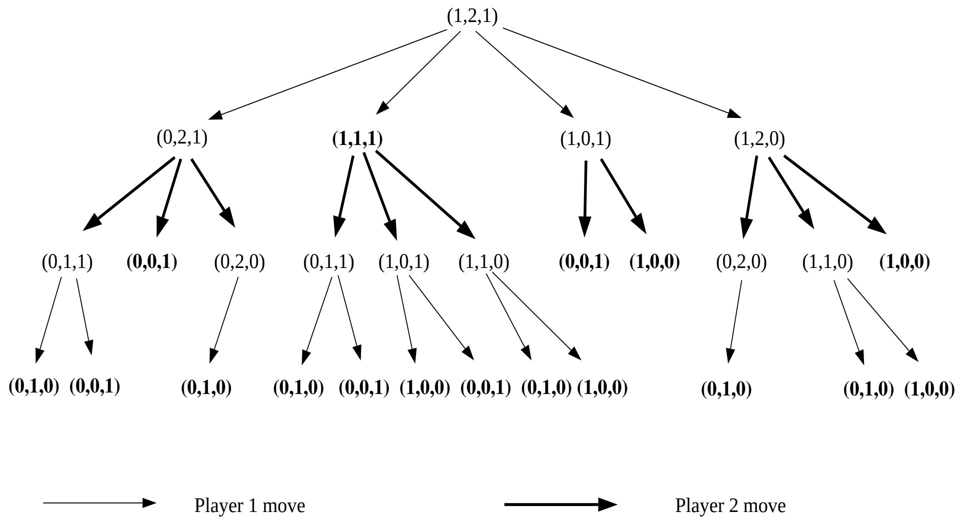

Figure 4.3: State-space search for an optimal move

of 'wsp', and all the functions and lemmas associated with this proof, are listed in Appendix D. Some subsidiary functions are defined in the Piton proof report [38]. In particular, wsp ($state$, **t**) returns FALSE if state is a losing state, and an optimal move otherwise.

DEFINITION:
wsp ($state$, $flag$)
= **if** $flag$
then if all-zero-bitvp ($state$) $\vee$ ($\neg$ nat-listp-simple ($state$)) **then** $state$
else wsp (all-valid-moves ($state$), **f**) **endif**
elseif listp ($state$)
then if $\neg$ wsp (car ($state$), **t**) **then** car ($state$)
else wsp (cdr ($state$), **f**) **endif**
else f endif

Nqthm, described in Chapter 2, provides a facility for executing functions on constants and a trace facility. Figure 4.4 is a trace of the calculation of an optimal move from the same state whose search tree is presented in Figure 4.3. The result of the calculation is '(1 1 1), which is indeed an optimal move.

```
*(wsp '(1 2 1) t)
  1> (LIST '(1 2 1) T)
    2> (LIST '((0 2 1) (1 1 1) (1 0 1) (1 2 0)) F)
      3> (LIST '(0 2 1) T)
        4> (LIST '((0 1 1) (0 0 1) (0 2 0)) F)
          5> (LIST '(0 1 1) T)
            6> (LIST '((0 0 1) (0 1 0)) F)
              7> (LIST '(0 0 1) T)
                8> (LIST '((0 0 0)) F)
                  9> (LIST '(0 0 0) T)
                  <9 '(0 0 0)
                  9> (LIST NIL F)
                  <9 f
                <8 f
              <7 f
            <6 '(0 0 1)
          <5 '(0 0 1)
          5> (LIST '((0 0 1) (0 2 0)) F)
            6> (LIST '(0 0 1) T)
              7> (LIST '((0 0 0)) F)
                8> (LIST '(0 0 0) T)
                <8 '(0 0 0)
                8> (LIST NIL F)
                <8 f
              <7 f
            <6 f
          <5 '(0 0 1)
        <4 '(0 0 1)
      <3 '(0 0 1)
      3> (LIST '((1 1 1) (1 0 1) (1 2 0)) F)
        4> (LIST '(1 1 1) T)
          5> (LIST '((0 1 1) (1 0 1) (1 1 0)) F)
            6> (LIST '(0 1 1) T)
              7> (LIST '((0 0 1) (0 1 0)) F)
                8> (LIST '(0 0 1) T)
                  9> (LIST '((0 0 0)) F)
                    10> (LIST '(0 0 0) T)
                    <10 '(0 0 0)
                    10> (LIST NIL F)
                    <10 f
                  <9 f
                <8 f
              <7 '(0 0 1)
            <6 '(0 0 1)
            6> (LIST '((1 0 1) (1 1 0)) F)
              7> (LIST '(1 0 1) T)
                8> (LIST '((0 0 1) (1 0 0)) F)
                  9> (LIST '(0 0 1) T)
                    10> (LIST '((0 0 0)) F)
                      11> (LIST '(0 0 0) T)
                      <11 '(0 0 0)
                      11> (LIST NIL F)
                      <11 f
                    <10 f
                  <9 f
                <8 '(0 0 1)
              <7 '(0 0 1)
              7> (LIST '((1 1 0)) F)
                8> (LIST '(1 1 0) T)
                  9> (LIST '((0 1 0) (1 0 0)) F)
                    10> (LIST '(0 1 0) T)
                      11> (LIST '((0 0 0)) F)
                        12> (LIST '(0 0 0) T)
                        <12 '(0 0 0)
                        12> (LIST NIL F)
                        <12 f
                      <11 f
                    <10 f
                  <9 '(0 1 0)
                <8 '(0 1 0)
                8> (LIST NIL F)
                <8 f
              <7 f
            <6 f
          <5 f
        <4 f
      <3 '(1 1 1)
    <2 '(1 1 1)
  <1 '(1 1 1)
'(1 1 1)
*
```

Figure 4.4: Nqthm trace of 'wsp' execution

4.4 A Nim-Playing Program Specification

We specify a Piton program that efficiently calculates an optimal nim move. The specification is in the form of several Nqthm predicates listed in this section that introduce five undefined Nqthm functions related to the implementation. In this methodology, the programmer provides these functions and is obliged to prove that they conform to the specification constraints. The five functions are:

- 'cm-prog' : the program that implements a good nim move generator;
- 'computer-move-clock' : number of instructions the program will execute;
- 'nim-piton-ctrl-stk-requirement' : upper bound on control stack use;
- 'nim-piton-temp-stk-requirement' : upper bound on temporary stack use;
- 'computer-move' : modified nim state resulting from an optimal move.

The following subsections present the constraints the specification imposes on the programmer. We describe each constraint informally and then provide the Nqthm term that makes it precise. The conjunction of these terms is the program specification.

4.4.1 Algorithm Legality

The function 'computer-move' must return legal nim moves. This requirement is expressed in the Nqthm logic with the term below. (Note that 'computer-move' is one of the undefined functions being constrained in the specification, and that 'good-non-empty-nim-statep' and 'valid-movep' are defined in Appendix D.)

$$\text{good-non-empty-nim-statep}\,(state) \rightarrow \text{valid-movep}\,(state,\ \text{computer-move}\,(state))$$

4.4.2 Algorithm Optimality

We require that ‘computer-move’ returns an optimal move.

$$
\begin{array}{l}
(\text{good-non-empty-nim-statep}\,(\mathit{state}) \wedge \text{wsp}\,(\mathit{state},\, \mathtt{t})) \\
\rightarrow \quad (\neg\ \text{wsp}\,(\text{computer-move}\,(\mathit{state}),\, \mathtt{t}))
\end{array}
$$

4.4.3 Algorithm Implementation

We require that the execution of the Piton program produces the same result as execution of the Nqthm function ‘computer-move’. We represent the nim state by an array of naturals and a length that are passed to the Piton program as parameters. We require that when the Piton subroutine `computer-move` in a list of programs returned by the function ‘cm-prog’ is executed using the Piton interpreter on a reasonable Piton state for ‘computer-move-clock’ ticks, the resulting state has an incremented program counter, the program status word set to ’run, and the naturals array representing the nim state is replaced by a new array with the same value as that calculated by ‘computer-move’.

$$
\begin{array}{l}
((\mathit{p0} = \text{p-state}\,(\mathit{pc},\ \mathit{ctrl\text{-}stk}, \\
\qquad\qquad \text{cons}\,(\mathit{wa},\ \text{cons}\,(\mathit{np},\ \text{cons}\,(\mathit{s},\ \mathit{temp\text{-}stk}))), \\
\qquad\qquad \text{append}\,(\text{cm-prog}\,(\mathit{word\text{-}size}),\ \mathit{prog\text{-}segment}), \\
\qquad\qquad \mathit{data\text{-}segment},\ \mathit{max\text{-}ctrl\text{-}stk\text{-}size}, \\
\qquad\qquad \mathit{max\text{-}temp\text{-}stk\text{-}size},\ \mathit{word\text{-}size},\ \mathtt{'run})) \\
\ \wedge \quad (\text{p-current-instruction}\,(\mathit{p0}) = \mathtt{'(call\ computer\text{-}move)}) \\
\ \wedge \quad \text{computer-move-implemented-input-conditionp}\,(\mathit{p0})) \\
\rightarrow \quad \textbf{let}\ \mathit{result}\ \textbf{be}\ \text{p}\,(\mathit{p0}, \\
\qquad\qquad\qquad \text{computer-move-clock}\,(\text{untag-array}\,(\text{array}\,(\text{car}\,(\text{untag}\,(\mathit{s})), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{data\text{-}segment})), \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{word\text{-}size})) \\
\qquad \textbf{in} \\
\qquad (\text{p-pc}\,(\mathit{result}) = \text{add1-addr}\,(\mathit{pc})) \\
\qquad \wedge \quad (\text{p-psw}\,(\mathit{result}) = \mathtt{'run}) \\
\qquad \wedge \quad (\text{untag-array}\,(\text{array}\,(\text{car}\,(\text{untag}\,(\mathit{s})),\ \text{p-data-segment}\,(\mathit{result}))) \\
\qquad\qquad = \quad \text{computer-move}\,(\text{untag-array}\,(\text{array}\,(\text{car}\,(\text{untag}\,(\mathit{s})), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{data\text{-}segment})))) \ \textbf{endlet}
\end{array}
$$

The function ‘computer-move-implemented-input-conditionp’ above identifies “reasonable” Piton states from which we expect our program to work properly. The conditions that must be met are:

- The control stack is non-empty.
- The word-size is at least 8 bits.
- At least NIM-PITON-CTRL-STK-REQUIREMENTS bytes are available on the control stack.
- At least NIM-PITON-TEMP-STK-REQUIREMENTS bytes are available on the temporary stack
- A naturals array address is first on the temporary stack. (This array is used to represent the nim state.)
- The length of the naturals array is second on the temporary stack.
- An array address is third on the temp stack. (This array is used as a work area for the program.)
- The array of naturals whose address is first on the stack contains at least one non-zero element.

4.4.4 Predictable Response Time

We require that the program calculate a move within a window of time. The program must execute between 10,000 and 20,000 Piton instructions. We assume a word size of at most 32 bits, and that the nim state has no more than 6 piles.

$$\begin{aligned}
&(\text{nat-listp}\,(state,\ ws) \\
&\ \wedge\quad (0 < ws) \\
&\ \wedge\quad (32 \geq ws) \\
&\ \wedge\quad (1 < \text{length}\,(state)) \\
&\ \wedge\quad (6 \geq \text{length}\,(state))) \\
&\rightarrow\quad ((10000 < \text{computer-move-clock}\,(state,\ ws)) \\
&\qquad \wedge\quad (\text{computer-move-clock}\,(state,\ ws) < 20000))
\end{aligned}$$

Note that this part of the specification eliminates some possible implementations. One is the blind search implementation suggested by the definition of `WSP`, since the first level of the search tree has as many as $6 * 2^{32}$ nodes, and there are as many as $6 * 2^{32}$ levels to the tree.

4.4.5 Realistic Memory Use

We require the implementation use little stack space.

$$(\text{NIM-PITON-CTRL-STK-REQUIREMENT} + \text{NIM-PITON-TEMP-STK-REQUIREMENT}) < 1000$$

This part of the specification eliminates, for example, a table-driven implementation since there are 2^{177} distinct possible nim states.

4.4.6 FM9001 Loadability

We require that the program work on an FM9001 and that it meet the requirements of the compiler correctness proof [38]. This requires among other things that the compiled Piton programs fit into the FM9001 address space and that the Piton programs be well-formed. The interested reader is referred to the Piton compiler correctness proof for details of this part of the specification, which has enabled us to apply the compiler correctness theorem [38].

This constraint eliminates some undesirable implementations that would otherwise satisfy the specification. An immensely long program that uses alternation to solve the nim problem by cases will not fit into the FM9001 address space when compiled, and will therefore not meet the requirements of this part of the specification.

4.5 The Nim Implementation

In this section we develop an algorithm for efficient calculation of optimal moves, and present a Piton program that implements this algorithm. In Section 4.6 we discuss the machine-checked proof that this implementation meets the specification developed in Section 4.4.

Since a formal specification has been developed for this program as well as a mechanical proof that the program meets the specification, a reader interested only in the behavior of the nim software would best skip this section. Note that as a part of the correctness proof, for example, each of the hundreds of program instructions has been proved to execute on arguments of the expected Piton type, each loop is proved to terminate, the stacks are proved never to be exhausted, etc. In contrast to conventional program development efforts where the program source code is the only place where a dependable description of the behavior of the system can be found, in this effort the specification in Section 4.4 is dependable because the correctness proof outlined in Section 4.6 guarantees that it is a correct description of the program's behavior.

We present the program because we hope to demonstrate how verified software for the short stack is developed and to make more concrete the description of the development effort.

4.5.1 An Efficient Nim Algorithm

Recall from Section 4.4 the definition of 'wsp' and that the term wsp ($state$, **t**) is FALSE if no winning strategy exists for $state$ and non-FALSE otherwise. Let bigp ($state$) = number of piles with at least 2 stones. Let a bit-vector representation be the low-order-bit-first base 2 representation for some number of bits. Let xor-bvs ($state$, ws) = bitwise exclusive-or of the bit-vector representations for ws bits of the number of stones in each pile. Let green-statep ($state$, ws) = (0 < bigp ($state$)) ↔ (xor-bvs ($state$, ws) ≠ 0-VECTOR).

THEOREM: wsp-green-state
nat-listp (s, $wordsize$) → (wsp (s, **t**) ↔ green-statep (s, $wordsize$))

This remarkable property was rediscovered for this project, but has in fact been known at least since its publication in 1901 [9]. The hypothesis requires that s be a list of naturals each representable with ws bits. The most obvious proof uses an induction on the search tree. The mechanical proof was achieved as a part of this effort since it is a needed lemma in the proof of program correctness that we discuss in the next section. Figure 4.5 is the search tree of Figure 4.3 except that stars are added to nodes where green-statep ($state$, 32) is non-false. Note that, as guaranteed by the theorem above, the non-bold nodes from which there is a winning strategy are exactly the nodes that are marked with stars.

We exploit this theorem in the following algorithm that computes optimal moves efficiently.

If bigp ($state$) < 2 and there are an even number of non-empty piles, remove all the stones from a largest pile.

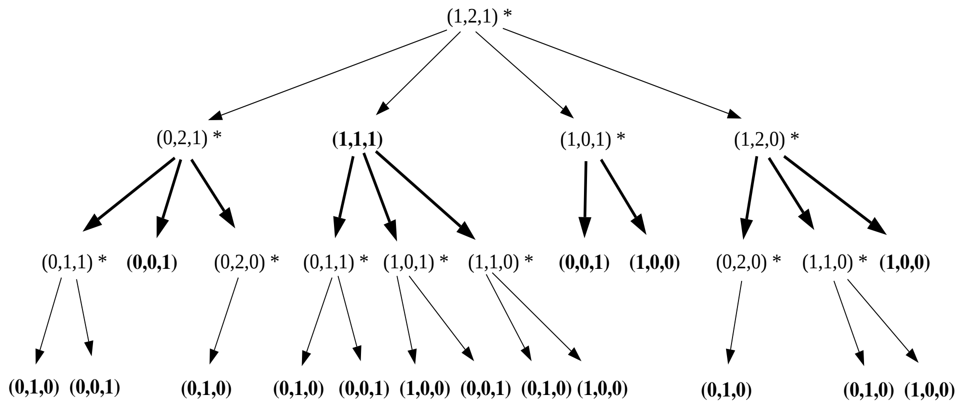

Figure 4.5: Search for an optimal move with marked "green" states

> If bigp $(state) < 2$ and there are an odd number of non-empty piles, remove all but one stone from a largest pile.
>
> Otherwise, find a pile whose binary representation has a 1-bit in the position of the highest 1-bit in xor-bvs $(state)$, and remove enough stones so that the new pile's binary representation is the 'xor' of the binary representations of the other piles.

From the previous theorem we prove that this algorithm efficiently generates optimal moves when a winning strategy exists. A mechanical proof of this was constructed as Nqthm prove-lemma event 'computer-move-works'.

4.5.2 The Piton Nim Program

A Piton program that implements this algorithm has been coded and it appears as the Nqthm definition 'cm-prog'. A listing with subsidiary definitions expanded and with cosmetic changes intended to enhance readability appears in Appendix B, and Figure 4.6 lists two subroutines.

```
SUBROUTINE NAT-TO-BV
   (VALUE) (CURRENT-BIT TEMP)
        CALL PUSH-1-VECTOR
        POP-LOCAL CURRENT-BIT
        CALL PUSH-1-VECTOR
        RSH-BITV
 LOOP:  PUSH-LOCAL VALUE
        TEST-NAT-AND-JUMP ZERO DONE
        PUSH-LOCAL VALUE
        DIV2-NAT
        POP-LOCAL TEMP
        POP-LOCAL VALUE
        PUSH-LOCAL TEMP
        TEST-NAT-AND-JUMP ZERO LAB
        PUSH-LOCAL CURRENT-BIT
        XOR-BITV
 LAB:   PUSH-LOCAL CURRENT-BIT
        LSH-BITV
        POP-LOCAL CURRENT-BIT
        JUMP LOOP
 DONE:  RET
```

```
SUBROUTINE NAT-TO-BV-LIST
   (NAT-LIST BV-LIST LENGTH) (I 0)
 LOOP:  PUSH-LOCAL NAT-LIST
        FETCH
        CALL NAT-TO-BV
        PUSH-LOCAL BV-LIST
        DEPOSIT
        PUSH-LOCAL I
        ADD1-NAT
        SET-LOCAL I
        PUSH-LOCAL LENGTH
        EQ
        TEST-BOOL-AND-JUMP T DONE
        PUSH-LOCAL NAT-LIST
        PUSH-CONSTANT (NAT 1)
        ADD-ADDR
        POP-LOCAL NAT-LIST
        PUSH-LOCAL BV-LIST
        PUSH-CONSTANT (NAT 1)
        ADD-ADDR
        POP-LOCAL BV-LIST
        JUMP LOOP
 DONE:  RET
```

Figure 4.6: Two example Piton subroutines

4.6 The Nqthm Correctness Proof

Nqthm has been used to construct proofs that correspond to the six specification constraints given in Section 4.4, so our Piton implementation has been proved correct. Approximately 750 lemmas were proved in order to guide Nqthm to the proof of these theorems. Most of these lemmas fall into one of the following four categories.

- Some lemmas relate to an Nqthm function the behavior of Piton loops and subroutines when interpreted by the Piton interpreter. Typically, a special Nqthm function that calculates a result in precisely the same manner as the Piton program in question is defined. A clock function that computes the number of Piton instructions the loop or subroutine executes is defined. A correctness lemma for a loop or subroutine states that the Piton interpreter running the loop or subroutine for the number of instructions computed by the clock function yields the same result as the Nqthm function.

  We call proofs of this kind of lemma *code correctness proofs*.

- Some lemmas demonstrate the equivalence of Nqthm functions that calculate values in a manner similar to Piton programs with Nqthm functions defined more naturally that are therefore easier with which to reason.

  We call proofs of this kind of lemma *specification proofs*.

- Some lemmas are used to prove the optimality of the algorithm formalized in the Nqthm function 'computer-move'. That is, that the algorithm outlined in Section 4.5 works.

  We call proofs of this kind of lemma *algorithm proofs*.

```
(prove-lemma correctness-of-nat-to-bv (rewrite)
        (implies
         (and
          (equal p0 (p-state pc ctrl-stk (cons v temp-stk) prog-segment
                     data-segment max-ctrl-stk-size max-temp-stk-size
                     word-size 'run))
          (equal (p-current-instruction p0) '(call nat-to-bv))
          (nat-to-bv-input-conditionp p0))
         (equal
          (p p0 (nat-to-bv-clock num))
          (p-state
            (add1-addr pc) ctrl-stk
            (cons (list 'bitv (nat-to-bv (cadr v) word-size)) temp-stk)
            prog-segment data-segment max-ctrl-stk-size
            max-temp-stk-size word-size 'run))))
```

Figure 4.7: An Nqthm prove-lemma event

- Some lemmas establish bounds on the clock functions.

  We call proofs of this kind of lemma *timing proofs*.

Although these proofs require only Nqthm, they were developed with extensive use of Pc-Nqthm [30], the interactive enhancement to Nqthm. The proof that the Piton program meets the specification uses the default arithmetic library [53] and the Piton interpreter definitions [38].

The correctness of the Piton subroutine `nat-to-bv` listed in Figure 4.6 is described by the prove-lemma command in Figure 4.7. The proof of this lemma illustrates what it takes to prove Piton correctness theorems. First, we accomplish a code correctness proof of the loop in `nat-to-bv`. We introduce a recursive function 'foo1' in the Nqthm logic that computes the values calculated in the loop, and use Nqthm to prove inductively that the interpreter applied to a Piton state that is about to execute that loop calculates the same values as 'foo1'. Several trivial subsidiary lemmas must be proved first so as to guide Nqthm to this proof. We then introduce a function 'foo2' that computes values in the same manner as the Piton subroutine `nat-to-bv`, and use Nqthm to generate a code correctness proof of this. We next lead Nqthm to generate a specification proof of the equivalence of 'foo2' and 'nat-to-bv', a conventional Nqthm definition of a natural number to bit vector conversion function. This requires dozens of subsidiary lemmas about arithmetic. Finally, we use Nqthm to prove the lemma shown in Figure 4.7, which it does using the previously-proved lemmas.

The correctness lemma is even more complex than it may appear at first glance. The function 'nat-to-bv-input-conditionp' that appears in a hypothesis contains additional preconditions that this subprogram requires in order to run correctly. There must be enough stack space to do the calculation, the value at the top of the stack must be a natural number representable on the machine, and the program segment must have the needed programs loaded. The function 'nat-to-bv-clock' computes the number of instructions the Piton subroutine `nat-to-bv` executes when called.

Like most conjectures that Nqthm accepts as a theorem, 'correctness-of-nat-to-bv' has two important properties. First, since it is accepted by the Nqthm prover we believe that it represents mathematical truth. Second, it is an instruction to the prover about how to prove future theorems. By constructing the exact form of the theorem mindful of the theorem's interpretation in later proof efforts, we add to the prover's ability to reason soundly about Piton programs that call `nat-to-bv`.

The lemma 'correctness-of-nat-to-bv' is useful because it equates the behavior of a Piton subprogram as defined by the Piton interpreter to an Nqthm term that does not involve the interpreter. By applying this lemma Nqthm can reason about this program without regard to the semantics of

Piton. This makes proofs achievable since, as a practical matter, proofs involving Piton programs running on the very complicated Piton interpreter are much more complex than proofs about Nqthm functions that compute similar results. The lemma is stored in Nqthm as a replacement rule, and has been constructed so that *later proofs apply this lemma automatically.* The subroutine `nat-to-bv-list` contains several kinds of Piton instructions, and the proof of the correctness of `nat-to-bv-list` depends on the semantics of these instructions as defined in the Piton interpreter. For example, `PUSH-LOCAL` is defined in the Piton interpreter and the Nqthm theorem prover uses the definitions that describe the effect of executing a `PUSH-LOCAL` instruction automatically when constructing a proof. Similarly, the instruction `CALL NAT-TO-BV` in the subprogram is reasoned about by Nqthm automatically using the proved theorem 'correctness-of-nat-to-bv'.

Once a carefully-constructed correctness theorem such as this one about a subroutine has been added to the database of proved lemmas, a call to that subroutine in a Piton program is reasoned about automatically just as any built-in Piton instruction.

The development of the correctness proof required about 3 months, not including time to develop the events of the Piton compiler or arithmetic library or to accomplish an earlier proof related to nim [54]. Approximately 40% of the development time was spent on code correctness proofs, 30% on specification proofs, 20% on algorithm proofs, and 10% on timing proofs. Nqthm checks the proof in approximately 10 hours on a Sun Sparcstation IPC.

4.7 Nim Program Overview

Mechanical verification of programs in this manner is time-consuming and difficult. Nevertheless, and quite remarkably, the experience of building the nim program suggests that development time scales linearly with program length. Once a Piton subroutine has been proved correct, a call to this subroutine can be reasoned about as easily as any basic Piton statement.

A modest but non-trivial application has been constructed that makes use of a verified compiler and microprocessor. Its functional correctness has been verified using Nqthm and mechanically-checked proofs of bounds on the number of executed instructions have been constructed. An FM9001 was fabricated and runs a compiled version of the nim program. The fabricated FM9001 microprocessor, the Piton compiler, and the nim program were never tested in a conventional manner during development. Even so, each worked the first time and we would have been surprised if any had not.

Chapter 5

A Real-time Model of the FM9001

Chapter 4 presents the FM9001 processor briefly because it is the target of the verified Piton compiler. In this chapter we describe the FM9001 in more detail and present a new model of its behavior that we use to reason about real-time systems that contain an operating FM9001.

5.1 The Verified and Fabricated FM9001

The FM9001 is a general-purpose 32-bit microprocessor with 15 arithmetic and logical operations, five addressing modes, extensive arithmetical flag support, 16 general-purpose registers, and other features that make it practical to use [14]. The FM9001 is verified in the sense that a netlist-level description of the FM9001 has been proved to implement a programmer-level model of the processor. The verification was accomplished using Nqthm, so the specifications are expressed in the Nqthm logic. The netlist-level description has been used to fabricate physically the processor, and the resulting integrated circuit has run programs [55]. The programmer-level model has served as a basis for the development of highly-reliable applications such as the nim program described in Chapter 4.

Figure 5.1 presents pictorially the programmer-level model of the verified FM9001. The state of the processor is composed of the values of the

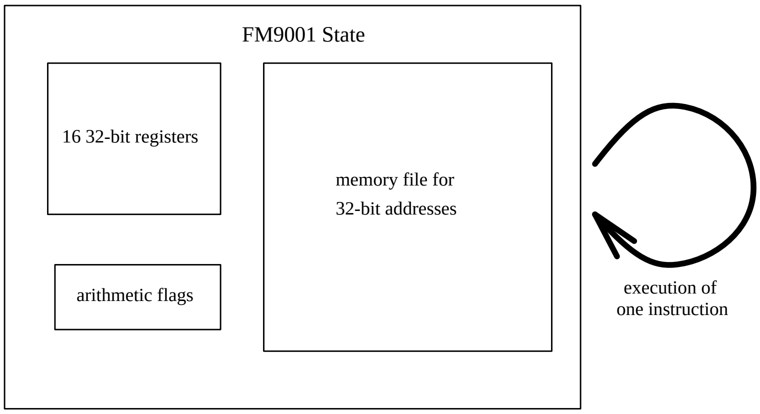

Figure 5.1: Programmer model of the FM9001

general-purpose registers, the arithmetical flags, and the memory. The execution of an instruction is accomplished each "tick" of a computation. The Nqthm function 'fm9001-interpreter' from the FM9001 correctness proof [14] models the execution of n instructions starting from an FM9001 state *state* using *pc-reg* as the designated program counter.

DEFINITION:
fm9001-interpreter (*state*, *pc-reg*, n)
= **if** $n \simeq 0$ **then** *state*
else fm9001-interpreter (fm9001-step (*state*, *pc-reg*), *pc-reg*, $n - 1$) **endif**

The function 'fm9001-step' is rather complex since it captures precisely the semantics of each FM9001 instruction. The reader interested in the details of 'fm9001-step' is referred to the FM9001 verification project report [14]. We use this function as our definition of the effect of one instruction in a new model of FM9001 operation. This new model, the *real-time FM9001 model*, allows useful reasoning about real-time properties of FM9001 operation. We develop and use a model of the FM9001 not identical to that proved implemented correctly in hardware, so we do not retain the extremely high confidence in correct hardware behavior achieved by the FM9001 hardware proofs. Nevertheless, we use as much of the FM9001 programmer specification as possible to retain as much of that confidence as possible.

Why not use the FM9001 model already used for the verification of the fabricated FM9001? There are weaknesses in 'fm9001-interpreter', the FM9001 programmer's model, from the standpoint of using it to model the behavior of real-time programs.

1. The timing of the instructions is not modeled. Each instruction is modeling to consuming one "tick", although in fact two instructions may require different amounts of actual time to execute.
2. There is no provision for I/O in the model.
3. A single, final state is returned. Real-time programmers are generally interested in invariants on the system rather than a final answer.

We address in the next several sections each of these issues in our development of the real-time FM9001 model that allows verification of real-time applications.

5.2 Timing

We wish to calculate how much time an instruction consumes during its execution in order to include that information in our FM9001 model. We express time in terms of the hardware clock that is being used to drive the execution of the FM9001. We call a tick of this underlying clock a *microcycle*. Each FM9001 instruction's execution is defined in terms of a sequence of internal FM9001 states, one for each microcycle during the execution of the instruction [14].

The instruction execution time calculation is in some sense outside the scope of the FM9001 design since the response time of the processor's memory is a component of the execution time of instructions. The board we have built to run FM9001 programs, described in Section 5.8, provides constant-time access to all memory locations. We assume in our real-time FM9001 model that memory access time is constant, and we parameterize a function that calculates instruction timing.

The proof of FM9001 correctness relates the execution of the programmer model to the execution of the underlying machine. Functions that calculate instruction timing in terms of microcycles have previously been developed for Nqthm, since they were needed in that proof. For a programmer-level FM9001 state *state* and a memory response time k, we calculate using the function 'microcycles-constant' the number of microcycles. Definitions of the real-time FM9001 model are listed in Appendix E. Several of the definitions presented in this section are defined in terms of subsidiary functions defined in the proof of the FM9001 design. The function 'microcycles-constant', for example, has the function 'map-down' in its definition. We refer the user to the description of the FM9001 hardware proof for definitions of these functions [14].

The Nqthm constant MIRROR-INITIAL-STATE is a representation of an FM9001 state that we present in Section 7.6. The current instruction of MIRROR-INITIAL-STATE — that is, the instruction in memory pointed to by the FM9001 program counter — is an assembled version of a move instruction. As an example, we use the Nqthm execution facility to calculate the number of microcycles of that instruction assuming a memory response time of 3 microcycles.

```
*(microcycles-constant (mirror-initial-state) 3)
 16
```

5.3 I/O

As noted previously the FM9001 programmer model has no provision for input or output. We change the model to allow memory-mapped I/O operations. That is, we associate I/O devices with locations in the FM9001 memory and use FM9001 instructions that access memory to accomplish I/O.

No changes to the FM9001 model need to be made to allow us to reason about output devices since there is nothing that keeps us from interpreting the meaning of a value at some memory location however we wish. For example, a bell might be associated with a memory location 23 so that a 0 value turns it off and a non-0 value turns it on. The situation with regard to input is more complicated since the current FM9001 programmer model incorporates the assumption that the values in memory are changed only by the action of an executing program. We add inputs into our model in the most general possible manner, by allowing any memory location to be changed at any microcycle during execution.

The function 'update-state-with-inputs' takes as its two arguments a list of location/value pairs `inputs` and an FM9001 programmer state and returns a new FM9001 state.

DEFINITION:
update-state-with-inputs (*inputs*, *state*)
= **if** listp (*inputs*)
  then update-state-with-inputs (cdr (*inputs*),
                list (car (*state*),
                  write-mem (nat-to-v (caar (*inputs*), 32),
                    cadr (*state*),
                    nat-to-v (cadar (*inputs*), 32))))
  else *state* **endif**

5.4 Instruction Initialization Times

Since we wish our model of I/O to be as general as possible, we allow inputs and outputs to change at any microcycle. How can we incorporate this into our FM9001 model simply but realistically? Our approach is to consider the execution of each instruction as having two phases. Roughly speaking, the first phase represents the fetching of the instruction from memory and the second phase represents the effect of the execution of the instruction. Three kinds of instructions are modeled: input instructions that update registers and flags based on memory value, output instructions that update memory with register values or instruction constants, and register-to-register instructions that do not interact with memory.

Obviously, the internal state of the actual FM9001 processor need be consistent with our model after the execution of each instruction. When modeling the interaction of the processor with memory, particularly I/O devices, we require even more precision from our model since it must reflect *to the microcycle* the effect of our processor on memory and the effect of inputs on the processor. In order to accomplish this the real-time model is more complex than the programmer's model captured in 'fm9001-interpreter'.

The function 'init-time' calculates the number of microcycles in phase 1 for the current instruction in state *state* assuming memory response time k. It calculates the "right" number of microcycles so that that phases are of the proper length depending on what kind of instruction it is. Note for example that the instruction `(move t f r1 (pc+))` has a phase 1 of length 11, assuming a memory response time of 3.

```
*(init-time (mirror-initial-state) 3)
 11
```

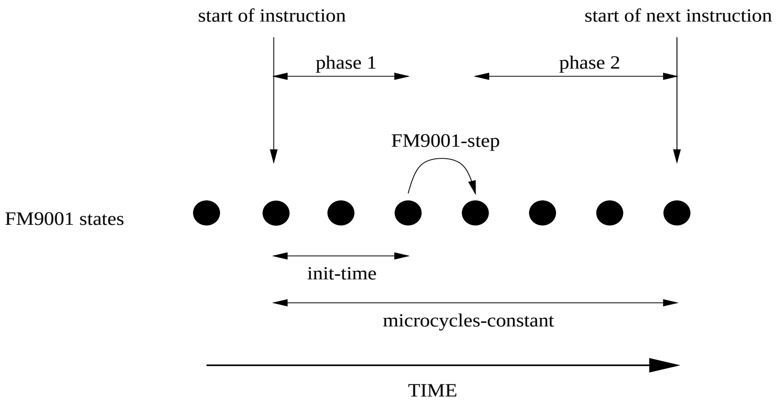

Figure 5.2: The real-time instruction execution model

The instruction is an input instruction that moves to register 1 the contents of the memory location immediately following the location of the instruction. Note that the value from memory that is copied is the value that occurs at that location during the 11th microcycle of the execution of the instruction.

5.5 Potential Two-Phase Model Inaccuracies

There are two ways that the simple two-phase instruction model we use to model the FM9001 is potentially inaccurate.

In our model the first FM9001 state of the second phase of an instruction's execution is determined by the last FM9001 state of the first phase and the inputs of that microcycle. This allows us to model an instruction's read from or write to memory accurate to the microcycle. But it may fail to read the proper value of the memory location from which the instruction is fetched. An instruction is fetched during the execution of an FM9001 instruction, but in our model it is only when an instruction does a read or write that its effect is visible. Given how we are modeling I/O, it is possible that the value of the instruction has changed during the first phase by action of an input device. So, a potential problem with our model is that we may use a value for an instruction that occurs a few clock ticks before or after the value that it should have in order to model accurately the operation of the FM9001.

There are a number of ways we could complicate our FM9001 model in order to avoid this problem. But as a practical matter we are not very concerned with reading the value of an instruction that is a few microcycles off, particularly as it seems unlikely we shall execute programs that reside in memory that is used for memory-mapped input. (Note, for example, that our model accurately models the execution of self-modifying programs.) Although we note this potential inaccuracy in the model we accept it in the interest of greater simplicity of our model.

We shall, however, include in our FM9001 model a check that will guarantee that we avoid a second potential problem with the two-phase instruction model. Some FM9001 instructions are *read-write* instructions that both read from and write to memory. Since any such instruction can be written as several functionally equivalent albeit slower instructions that are not read-write, and since building an FM9001 programmer's model that accurately models the behavior of read-write instructions would complicate our model, we have chosen to omit them. Our simple two-phase model is not accurate for read-write instructions, so we write a predicate that identifies FM9001 states in which the current instruction is one of these instructions.

The function 'read-write-statep' formalizes this predicate in Nqthm. Note that the current instruction in the example state described above is not a read-write instruction.

```
*(read-write-statep (mirror-initial-state))
 F
```

5.6 Tying It All Together

The *I/O configuration* of the FM9001 is how the input and output devices are connected to the FM9001. That is, which memory locations are associated with which I/O devices. We assume in our processor model that the I/O configuration of the FM9001 is unchanged during execution. The function 'fm9001-rt' calculates an FM9001 state given an initial state *state*, a list of input lists *inputs-list*, and a memory access time *mat*. If a read-write instruction is encountered a 0 is returned.

```
DEFINITION:
fm9001-rt (state, inputs-list, mat)
=   if inputs-list ≃ nil then state
    elseif read-write-statep (state) then 0
    else let init-time be init-time (state, mat),
             total-time be microcycles-constant (state, mat)
         in
         if init-time ≥ length (inputs-list)
         then update-state-with-inputs (last (inputs-list), state)
         else let new be fm9001-step (update-state-with-inputs (nth (init-time − 1,
                                                                      inputs-list),
                                                                 state),
                                      nat-to-v (15, REG-SIZE))
              in
              if total-time ≥ length (inputs-list)
              then update-state-with-inputs (last (inputs-list),
                                             new)
              else fm9001-rt (new,
                              nthcdr (total-time,
                                      inputs-list),
                              mat) endif endlet endif endlet endif
```

We assume in our model that the set of memory addresses updated at each microcycle is the same. This allows us to write the relatively simple definition of 'fm9001-rt' above. We take care in our use of this function to provide each input location with a value for each microcycle, so that our use of this assumption to write a simple version of 'fm9001-rt' does not interfere with our use of 'fm9001-rt' to mimic accurately the actual operation of the FM9001.

We define a function that extracts values from an FM9001 state and produces an association list. The variable *loclist* is either a list of location/name pairs where location is a list containing an FM9001 register number or else a number representing a memory location and name is the name to be used in the resulting association list.

```
DEFINITION:
extract-locs (loclist, state)
=   if listp (loclist)
    then cons (list (cadar (loclist),
                     if listp (caar (loclist))
                     then v-to-nat (read-mem (nat-to-v (caaar (loclist), REG-SIZE),
                                              regs (car (state))))
                     else v-to-nat (read-mem (nat-to-v (caar (loclist), 32),
                                             cadr (state))) endif),
               extract-locs (cdr (loclist), state))
    else nil endif
```

We wish to have our real-time FM9001 model produce a history of values, not just a single, final result. The function 'fm9001-rt-history' generates values until either the inputs are exhausted or a read-write instruction is encountered.

```
DEFINITION:
fm9001-rt-history-helper (state, mat, inputs-list, loclist, n)
=   if n < length (inputs-list)
    then let next-state be fm9001-rt (state, firstn (1 + n, inputs-list), mat)
         in
         if next-state = 0 then nil
         else cons (extract-locs (loclist, next-state),
                    fm9001-rt-history-helper (state,
                                              mat,
                                              inputs-list,
                                              loclist,
                                              1 + n)) endif endlet
    else nil endif
```

```
DEFINITION:
fm9001-rt-history (state, mat, inputs-list, loclist)
=   fm9001-rt-history-helper (state, mat, inputs-list, loclist, 0)
```

5.7 FM9001 Representation and Execution Example

The representation of the FM9001 state we use is identical to that used in the FM9001 verification effort [14], even though as discussed the model of execution we use is more detailed. We shall not describe in detail how we represent the components of the FM9001 state or how to write and assemble an FM9001 state that contains a real-time application. We omit this seemingly important information not just because it is already described fully elsewhere [14], but because how we build an FM9001 state is not needed to understand the correctness results. As is suggested by the real-time correctness theorems we prove in Chapter 7, we assume that the system about which we are proving a desired behavior accurately reflects the operation of an FM9001 running a particular application. He who is interested in the successful operation of the system as guaranteed by the correctness lemma need not be familiar with the details of the application or other components of the FM9001 state.

Nevertheless, for the sake of concreteness and to illustrate the definitions in this chapter, we demonstrate our FM9001 model on an example. Figure 5.3 presents an Nqthm definition of a constant that represents an FM9001 state. It is defined using an assembler and other functions that allow convenient building of FM9001 states. (They appear, of course, in Appendix E.) Again, we do not discuss these functions in detail because how we build a state is not terribly important in this project. The example state has a program at location $\texttt{40000000}_{16}$ that increments repeatedly the value at location $\texttt{50000000}_{16}$, a bit-vector representing the value 5 at location $\texttt{50000000}_{16}$, and a pointer to the loaded program in the program counter also known as register 15.

We use the R-loop facility to evaluate our FM9001 model starting with this example state, a memory response time of 3, for 80 microcycles with

```
(defn fm9001-example-state ()
  (list
   (list
      (write-mem-ignore-stubs
         (nat-to-v 15 4)
         (list-to-mem (repeat 15 (nat-to-v 0 32)) 4 (repeat 32 f))
         (nat-to-v #x40000000 32))
         (list t f f f))
   (write-array-memory
      (list (nat-to-v 5 32))
      #x50000000
      (write-array-memory
         (asm '((move t f r1 (pc+))
            #x50000000
            (move t f r0 (r1))
            (inc t f (r1) r0)
            (move t f pc (pc+))
            #x40000000))
      #x40000000 (empty-memory 32)))))
```

Figure 5.3: Nqthm definition of a constant representing an FM9001 state

no memory locations associated with memory-mapped input devices, reporting the value at location $\mathtt{50000000}_{16}$ at each microcycle and associating it with the name *out*.

```
*(fm9001-rt-history
     (fm9001-example-state) 3 (repeat 80 nil) '((#x50000000 out)))
   '(((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5))
     ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5))
     ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5))
     ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5))
     ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5))
     ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5))
     ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 6)) ((OUT 6)) ((OUT 6)) ((OUT 6))
     ((OUT 6)) ((OUT 6)) ((OUT 6)) ((OUT 6)) ((OUT 6)) ((OUT 6)) ((OUT 6))
     ((OUT 6)) ((OUT 6)) ((OUT 6)) ((OUT 6)) ((OUT 6)) ((OUT 6)) ((OUT 6))
     ((OUT 6)) ((OUT 6)) ((OUT 6)) ((OUT 6)) ((OUT 6)) ((OUT 6)) ((OUT 6))
     ((OUT 6)) ((OUT 6)) ((OUT 6)) ((OUT 6)) ((OUT 6)) ((OUT 6)) ((OUT 6))
     ((OUT 6)) ((OUT 6)) ((OUT 6)))
```

Note that after some time the value of the location is incremented by the FM9001 running our example state.

5.8 The FM9001 Single Board Computer

The FM9001 single board computer contains an FM9001 and several other devices needed to run FM9001 programs [2, 55]. We have modified the board in order to run applications that perform I/O. Rather than present a complete description of the FM9001 board, we present here a description of the changes we made to the board for this project. The original board design is therefore needed to understand in detail what the modified board contains [2]. The original single board computer was designed and built by Ken Albin, who also helped the author make these modifications.

The changes to the board include the following.

- The "UART wait" memory access mode that was used for memory addresses $\texttt{C0000000}_{16}$ to $\texttt{FFFFFFFF}_{16}$ has been eliminated.
- A memory access mode for memory-mapped I/O devices is implemented for memory locations $\texttt{C0000000}_{16}$ to $\texttt{FFFFFFFF}_{16}$.
- Two push-buttons installed on the board are wired to provide values to locations $\texttt{C0000000}_{16}$ and $\texttt{C0000002}_{16}$.
- Two LEDs installed on the board are wired to display the least significant bit of the value most recently written by the FM9001 to locations $\texttt{C0000001}_{16}$ and $\texttt{C0000003}_{16}$.
- The timing of memory access is changed. Previously, the memory acknowledgment signal was generated using a clock other than the FM9001 clock. Although it is good engineering practice to make memory access independent of the FM9001 clock, this complicated the timing of

instructions. Memory access to the ROM, RAM, or an I/O device is now acknowledged after 3 microcycles.

- Some unnecessarily complex logical expressions in a PLD design were simplified.

These modifications resulted in the redesign of FM_addr, the PLD responsible for memory addressing, and the creation of a new PLD, FM_button, that drives the buttons and lights. Also, 30 wiring changes were made to the board. The two new PLD designs and a list of the wiring changes are presented as Appendix A.

The only loss of functionality of the board as a result of these changes is the elimination of UART wait memory access mode. No known FM9001 program used this feature of the old board. The new board runs all known FM9001 benchmarks, including the nim program described in Chapter 4.

5.9 'fm9001-rt-history' Versus 'fm9001-interpreter'

The real-time model of the FM9001 is different from the programmer's model of the FM9001 that was used to specify the behavior of the FM9001 for its verification [14] and serves as the target of the Piton compiler described in Chapter 4. The hardware verification proof gives us confidence that the fabricated FM9001 will operate as predicted by the programmer's model, but we use the FM9001 real-time model described in this chapter as our model of the FM9001 for real-time system verification. The subject of this part of this dissertation is the proof of abstract properties about real-time systems containing applications using detailed processor models, so possible model inaccuracies would not compromise our ultimate technical objective. We do not claim that the real-time model is proved equivalent to the fabricated FM9001, because parts of the real-time model formalized in the Nqthm function 'fm9001-rt-history' have not been shown to be an accurate description of the FM9001. We see no reason that a hardware verification proof using the real-time FM9001 model could not be accomplished, but we have not done it. The construction of such an Nqthm-checked proof would require perhaps several months of effort. Still, since the models are similar we would like to formally characterize their relationship, so we present a theorem that relates the functional behavior of these models in the absence of inputs. In order to make this relationship precise, we define two functions.

The Nqthm function 'read-write-encounteredp' is a predicate that holds when a read-write instruction is encountered during the execution of the FM9001 assuming no changes to the FM9001 state due to the action of input devices, starting in FM9001 state s, using program-counter pc, and executing n FM9001 instructions.

DEFINITION:
read-write-encounteredp (s, n, pc)
= **if** read-write-statep (s) **then t**
  elseif $n \simeq 0$ **then f**
  else read-write-encounteredp (fm9001-step (s, pc), $n - 1$, pc) **endif**

This theorem has been proved using Nqthm.

The Nqthm function 'number-fm9001-instructions' returns the number of FM9001 instructions executed in n microcycles starting in state s with memory access time mat using program counter pc.

DEFINITION:
number-fm9001-instructions (n, s, mat, pc)
= **let** *init* **be** init-time (s, mat),
    total **be** microcycles-constant (s, mat)
  in

$$
\begin{array}{l}
\textbf{if } \mathit{total} \geq n \\
\textbf{then if } \mathit{init} < n \textbf{ then } \mathtt{1} \\
\qquad \textbf{else } \mathtt{0} \textbf{ endif} \\
\textbf{else } 1 + \text{number-fm9001-instructions}\,(n - \mathit{total}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{fm9001-step}\,(s,\ pc), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{mat}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad pc) \textbf{ endif endlet}
\end{array}
$$

We formalize the relationship between the models. As is evident from the development of the real-time model in this chapter, the programmer's model is a special case of the real-time model where no memory values are changed due to input, no read-write instructions are encountered during execution, and the programmer's model is executed for as many instructions as the real-time model calculates for a some execution time.

Theorem: fm9001-rt-history-versus-fm9001-interpreter

$$
\begin{array}{l}
\textbf{let } \mathit{numinstr} \textbf{ be } \text{number-fm9001-instructions}\,(n,\ s,\ \mathit{mat},\ pc) \\
\textbf{in} \\
((\neg \text{ read-write-encounteredp}\,(s,\ \mathit{numinstr},\ pc)) \\
\ \land \quad (pc = \text{nat-to-v}\,(\mathtt{15},\ \text{REG-SIZE})) \\
\ \land \quad \text{listp}\,(s) \\
\ \land \quad (\mathtt{0} < \mathit{mat}) \\
\ \land \quad (\mathtt{0} < n)) \\
\rightarrow \quad (\text{last}\,(\text{fm9001-rt-history}\,(s,\ \mathit{mat},\ \text{repeat}\,(n,\ \textbf{nil}),\ \mathit{loclist})) \\
\qquad = \quad \text{extract-locs}\,(\mathit{loclist},\ \text{fm9001-interpreter}\,(s,\ pc,\ \mathit{numinstr}))) \textbf{ endlet}
\end{array}
$$

We use the FM9001 real-time model to describe the operation of a processor in real-time system correctness theorems in Chapter 7. In the next chapter we introduce a method for describing other real-time system properties.

Chapter 6

A Formalism for Real-Time System Properties

6.1 Introduction

As suggested in previous chapters, the correct operation of computer systems is difficult to assure because of their immense complexity. Building a reliable computer-based controller in a real-time environment is particularly difficult because events to which the computer must respond occur unpredictably. Many computer systems that operate in a real-time environment are safety-critical, so their correctness is crucial. This chapter describes a method for describing real-time system properties.

One can represent the state of a real-time system at an instant in time as an association of values with variables, and the state of a system over time by a *history*, a sequence of associations where consecutive associations represent the system at instants separated by some constant amount of time [57]. If for example the relevant aspects of a system were the state of a buzzer and the value of particular location in memory, we might choose to represent the system with the following history.

```
(((buzzer on) (loc1 3))  ((buzzer off) (loc1 3))  ((buzzer off) (loc1 4))
```

The development of formal processor models, in particular for use in hardware verification, provides in principle the capability to reason about real-time systems using very realistic models of computation [13, 14, 33]. A model of the FM9001 processor is developed in Chapter 5. But how do we describe precisely the properties of real-time systems about which we wish to reason?

Our approach is to view a real-time system as a history. We describe a property of the system by detailing the possible states of the system and how the system progresses from one state to another. This abstract view of real-time systems has been used in a variety of contexts. Real-time system that are described using various kinds of timed automata have been analyzed [3]. Modechart and Statechart are two state-based design approaches that support the design and implementation of real-time systems using a state-based model of their operation [26, 29]. Representing real-time system operation using a state-based formalism appears to be an appropriate and useful way to describe the properties we wish to verify.

We present a kind of timed automata called *behaviors* that we use to describe various real-time system properties and explain how they are related to histories. In later chapters we describe Nqthm-checked correctness theorems for simple real-time systems that demonstrate that a small application succeeds in maintaining system properties as described by behaviors.

6.2 A Syntactic Description of Behaviors

A *variable* is a symbol used to represent a value. A *counter* has form `(c n)`, where `n` is a natural number. An *expression* is either a natural number, a constant, a counter, a variable, a term of the form old(v) where v is a variable, or an operator applied to the appropriate number of arguments where each argument is an expression. An *assignment* is a list of name/value pairs. We evaluate an expression on two assignments *a1* and *a0* by replacing each variable in a term of the form old(v) that occurs in *a0* by the corresponding value in *a0*. Other expression variables and counters are replaced by their associated value in *a1* if it exists and 0 otherwise. We apply the definition of the operators to derive a value for the expression. The definitions of these operators are the same as the correspondingly-named functions in Nqthm and include 'equal', 'lessp', 'not', 'plus', 'remainder', 'difference', 'if', and 'and'.

For example, 'my-eval' evaluated on the term $b = (\text{old}\,(a) + 5)$ and assignments `((a 3) (b 7))` and `((a 2))` returns the boolean constant TRUE.

A *transition* has three components: a name called the *to-state* of the transition, a set of counters called the *resets* of the transition, and an expression called the *predicate* of the transition. A *transition list* is a list of transitions.

A *state* has three components: a name called the *name* of the state, an expression called the *predicate* of the state, and a transition list called the *trans-list* of the state. A *state list* is a list of states with unique state names.

A *behavior* has two components: a state list called the *state-list* of the behavior, and a name that occurs as a state name in the state-list called the *initial state name* of the behavior.

We often find it convenient to represent behaviors pictorially. Each state is represented by a circle containing its name and predicate. Each transition is represented by an arrow originating in the state containing the trans-list of which the transition is a member and ending at the state denoted by the transition's to-state. Each transition is labeled with its predicate and, if non-empty, its list of the reset counters preceded by "reset:". The behavior's initial state is pointed to by an arrow not originating from any state. We abbreviate counter `(c n)` as `cn` in behavior diagrams. Figure 6.1 presents an example of the pictorial representation of a behavior.

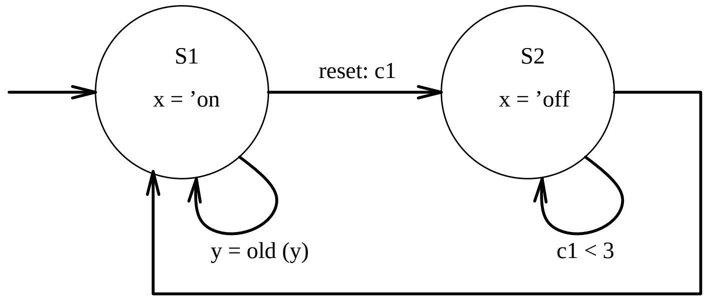

Figure 6.1: An example behavior

6.3 A Semantics for Behaviors

We present a semantics for behaviors by defining a predicate that decides whether a behavior is *satisfied* by a history.

A *counter extension of assignment* a by list c is an assignment that associates a value with each element of c and for non-elements of c associates (or does not associate) exactly as a. *A counter extension of history* h by list c is a history in which each assignment is a counter extension by c of the corresponding assignment in h.

A non-empty history h with assignments $h_0, h_1, ..., h_m$ is *reset consistent* with a list of transitions l when

1. Each counter associated with a value in h_0 is associated with 0.

2. Each counter c associated with a value in h_{i+1} where $c \in \text{resets}(l_i)$ is associated with 0.

3. Each counter c bound in h_{i+1} where $c \notin \text{resets}(l_i)$ is associated with one plus the value associated with c in h_i.

There is exactly one counter extension of h by c that is reset consistent with transition list l.

The *counters of a behavior* are the counters that occur in any predicate of a state or a predicate or resets of a transition in the state-list of a behavior.

A non-empty history h *satisfies* a behavior b if and only if there exists a list of transitions l that meets the following conditions. Let $a_0, a_1, ..., a_m$ denote the elements of the counter extension of h by the counters of b that is reset consistent with l, and let S be the list of state names such that $S_0 =$ the initial state name of b and $S_{i+1} =$ to-state(l_i).

1. $\forall_{i \leq m}$ S_i is the name of a state in the state-list of b.

2. $\forall_{i<m}$ $l_i \in$ trans-list(state in b with name S_i).

3. $\forall_{i \leq m}$ predicate(state in b with name S_i) holds evaluated on a_i, **nil**.

4. $\forall_{i<m}$ predicate(l_i) holds evaluated on a_{i+1}, a_i.

The behavior in Figure 6.1, for example, is satisfied by the history

```
((x on) (y 2))
((x off) (y 1))
((x off) (y 2))
((x off) (y 2)))
```

since there exists a list of transitions that meets the conditions of the definition of behavior satisfaction. In particular,

```
((x on) (y 2) ((c 1) 0))
((x off) (y 1) ((c 1) 0))
((x off) (y 2) ((c 1) 1))
((x off) (y 2) ((c 1) 2))
```

is the counter extension of the history by the counters of the behavior that is reset consistent with the list of the transition from S1 to S2, the transition from S2 to S2, and the transition from S2 to S2.

In contrast, the history

```
((x on) (y 2))
((x off) (y 1))
((x off) (y 2))
((x off) (y 2))
((x off) (y 2))
```

does not satisfy the behavior in Figure 6.1 since the conditions in the definition of satisfiability do not hold for any list of transitions. To demonstrate this assume the contrary, that a list of transitions and associated consistent counter extension exist that meet the conditions in the definition of satisfiability. The list of transitions starts with the transition from S1 to S2, since S1 is the initial state name and the value of y changes between the first two assignments. The next three transitions in the list are the transition from S2 to S2, since x has value 'off in the next four assignments. We can therefore calculate the value of (c 1) in the fifth assignment of the counter extension that is reset consistent with the transition list: it has value 3. But this is a contradiction, since the transition from S2 to S2 that occurs as the fourth transition in the list implies that the value of (c 1) in the fifth assignment of the counter extension reset consistent with it has value less than 3. So, our original assumption is false and we conclude that no such transition list exists. Therefore this history does not satisfy the behavior.

6.4 An Nqthm Formalization of Behaviors

We present a series of definitions expressed in the logic of the Nqthm theorem-proving program that formalizes the notion of satisfaction. We present some of the relevant definitions to convey the nature of our work, and refer the reader to descriptions of Nqthm [10]. We use the standard, Lisp-like Nqthm logic when showing execution examples and the Nqthm infix notation [11] when presenting definitions and theorems. Each of these definitions appears in Appendix E.

The function 'my-eval' evaluates an expression with respect to two assignments in the manner described in Section 6.2. Counters are represented by lists whose first element is `c`. Expressions of the form old(x) are interpreted as a reference to x's value in the second assignment. Other variables are replaced by the values associated with them in the first assignment. Several kinds of function symbols are interpreted as in the Nqthm logic [10].

We present the result of evaluating 'my-eval' on examples below, taking advantage of the executability of Nqthm functions. In the examples we type a formula with constant arguments to the "*" prompt and Nqthm responds with the result of evaluating the formula. (Note that the Nqthm logic is case-insensitive.)

```
*(my-eval '(plus a (old b)) '((a 3) (b 10)) '((a 11) (b 33)))
 36
*(my-eval '(equal (c 2) (plus 4 a)) '(((c 2) 3) (a 0)) ())
 F
```

We represent behaviors, states, and transitions using lists. The first three elements of a list representing a state are interpreted as its name, predicate, and trans-list. The first three elements of a list representing a transition are interpreted as the to-state, resets, and predicate of the transition. The first two elements of a list representing a behavior are interpreted as the initial state name and the state-list of the behavior.

As an example of the way we will use lists to represent behaviors we present the behavior that is presented pictorially in Figure 6.1.

```
((s1 (equal x 'on) ((s1 () (equal y (old y))) (s2 ((c 1)) ())))
 (s2 (equal x 'off) ((s1 () ()) (s2 () (lessp (c 1) 3)))))
```

We represent the values of counters using an association list that binds counter names to natural numbers. Using a transition's resets we can calculate the values of the counters after the transition from the values. Counters that are not reset are incremented and reset counters have value 0. We make this calculation with 'update-counter-alist' which takes a counter value association list and a list of reset counters as arguments.

```
*(update-counter-alist '(((c 1) 3) ((c 10) 11)) '((c 1)))
 '(((C 1) 0) ((C 10) 12))
```

The function 'make-list-alist' constructs an association list that binds the elements of its single parameter to the value 0. For example,

```
*(make-list-alist '(a b c))
 '((A 0) (B 0) (C 0))
```

The function 'counters-of-state-list' takes as an argument a list of behavior states and returns a list containing the counters that occur in a predicate or reset list of any of its states. We evaluate this function on an example that represents the state-list of the behavior represented in Figure 6.1.

```
*(counters-of-state-list
   '((s1 (equal x 'on) ((s1 () (equal y (old y))) (s2 ((c 1)) ())))
     (s2 (equal x 'off) ((s1 () ()) (s2 () (lessp (c 1) 3))))))
 '((C 1))
```

As previously noted, a history h satisfies a behavior b when there exists a list of transitions such that certain properties hold. This poses a challenge in our effort to write a function that formalizes satisfaction in the executable Nqthm logic. When possible we prefer to define functions in Nqthm that calculate results rather than define concepts by axiomatizing them, thereby simplifying proofs and allowing us to execute the functions. Since by conditions

```
DEFINITION:
satisfiesp-helper (name, pred, trans, h, counters, b-states)
=   if listp (h)
    then if listp (cdr (h))
         then if listp (trans)
              then assoc (name, b-states)
                   ∧  my-eval (pred, append (counters, car (h)), nil)
                   ∧  let new-state  be  assoc (trans-to (car (trans)), b-states),
                          new-counters  be  update-counter-alist (counters,
                                                                  trans-resets (car (trans)))
                      in
                      (my-eval (trans-pred (car (trans)),
                                append (new-counters, cadr (h)),
                                append (counters, car (h)))
                       ∧  satisfiesp-helper (state-name (new-state),
                                             state-pred (new-state),
                                             state-trans-list (new-state),
                                             cdr (h), new-counters, b-states))
                      ∨  satisfiesp-helper (name, pred, cdr (trans),
                                            h, counters, b-states) endlet
              else f endif
         else assoc (name, b-states)
              ∧  my-eval (pred, append (counters, car (h)), nil) endif
    else t endif
```

Figure 6.2: Definition of 'satisfiesp-helper'

(1) and (2) of the definition of satisfaction this list of transitions constitutes a "path" through the behavior, we formalize satisfaction in Nqthm by implementing a blind search of all possible paths. If any path meets the criteria of the definition then the history satisfies the behavior, and if none do then the history does not satisfy the behavior.

In Figure 6.2 we present 'satisfiesp-helper' that takes as arguments

```
DEFINITION:
satisfiesp (h, b)
=   let init-state  be  assoc (behavior-initial (b), behavior-state-list (b))
    in
    satisfiesp-helper (state-name (init-state), state-pred (init-state),
                       state-trans-list (init-state), h,
                       make-list-alist (counters-of-state-list (behavior-state-list (b))),
                       behavior-state-list (b)) endlet
```

Figure 6.3: Definition of 'satisfiesp'

a current state name, a predicate, a list of transitions, a history h, a counter association list and a list containing the states of a behavior b. The function 'satisfiesp-helper' assumes that the first element of h is mapped to a state of b with name *name*, predicate *pred*, and trans-list *trans* and returns a boolean result signifying whether there exists a transition list in which the values of the counters have the values of *counters* that demonstrate that h satisfies b. If h has fewer than 2 elements the definition of 'satisfiesp-helper' is simple. Otherwise, we check each possible transition to see if it could be the first element of a list of transitions that demonstrates the satisfaction of h by b. If none do we return FALSE, but if at least one does we return TRUE.

We use 'satisfiesp-helper' to define the function 'satisfiesp' in Figure 6.3 which formalizes the notion of satisfaction. In the previous section we showed using informal arguments that some example histories satisfied or did not satisfy the example behavior in Figure 6.1. We use our executable Nqthm formalization to calculate those results using the definitions presented in Figures 6.2 and 6.3.

```
*(satisfiesp
   '((((x on) (y 2) ((c 1) 0))
     ((x off) (y 1) ((c 1) 0))
     ((x off) (y 2) ((c 1) 1))
     ((x off) (y 2) ((c 1) 2)))
   '(s1 ((s1 (equal x 'on) ((s1 nil (equal y (old y))) (s2 ((c 1)) nil)))
         (s2 (equal x 'off) ((s1 nil nil) (s2 nil (lessp (c 1) 3)))))))
 T
*(satisfiesp
   '(((x on) (y 2))
     ((x off) (y 1))
     ((x off) (y 2))
     ((x off) (y 2))
     ((x off) (y 2)))
   '(s1 ((s1 (equal x 'on) ((s1 nil (equal y (old y))) (s2 ((c 1)) nil)))
         (s2 (equal x 'off) ((s1 nil nil) (s2 nil (lessp (c 1) 3)))))))
 F
```

In the next chapter we demonstrate the use of behaviors. We describe interesting properties of real-time systems, including properties having to do with the execution of programs on the FM9001, and prove theorems about these properties using Nqthm.

Chapter 7

A Verified Real-time System

We present a simple real-time system verified to have certain desirable properties. The system we have chosen as an example is a quiz-show in which each of two contestants has a button used to signal his desire to answer a question and a light indicating whether he has been selected. The buttons "bounce", so the button signals are not simply determined by whether the button was most recently pressed or released. We reason about the operation of an application using the real-time FM9001 model presented in Chapter 5 and describe the properties we desire from the system containing the FM9001 using behaviors as presented in Chapter 6.

In this chapter we present our requirements informally and describe the application that is conjectured to satisfy them. We present a correctness conjecture involving our FM9001 model, the application, and system properties expressed as behaviors that capture the notion of correctness. The sequence of definitions, theorems, and other events that lead Nqthm to a proof of the correctness conjecture is presented in Appendix E, and we outline that proof in this Chapter. We describe an even simpler application — a light-switch — that has also been similarly built, specified, and verified. The next chapter discusses some of the implications of this work.

7.1 Informal Description of a Quiz-show System

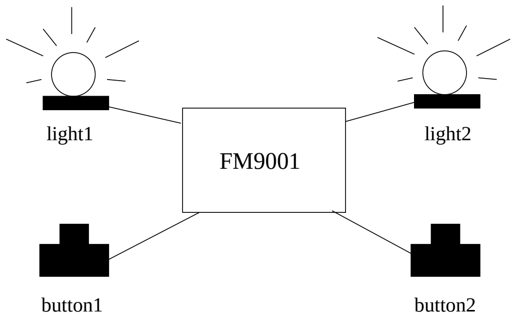

Figure 7.1: Elements of the quiz system

In this section we present an informal specification for a real-time system that works as a quiz-show signaling system. As suggested in Figure 7.1, this simple real-time system consists of two buttons we call button1 and button2, two lights we call light1 and light2, and an FM9001 processor connected to the buttons and lights and wired up with memory and other components necessary for the execution of FM9001 programs. We present the specifications in this section informally to motivate later sections in which we describe precisely a correctness theorem about this system and its Nqthm proof.

A specification for a real-time system is of the form

```
For all operations of the real-time system
  if it is consistent with
     a model of non-processor elements of the system
     and
     a model of the execution of a processor running the executive
   then
     it has a desired property.
```

In the case of the microprocessor-controlled quiz-show signaling system there are two kinds of elements other than the processor for which we need a model. The first of these is a model of the operation of the lights, which

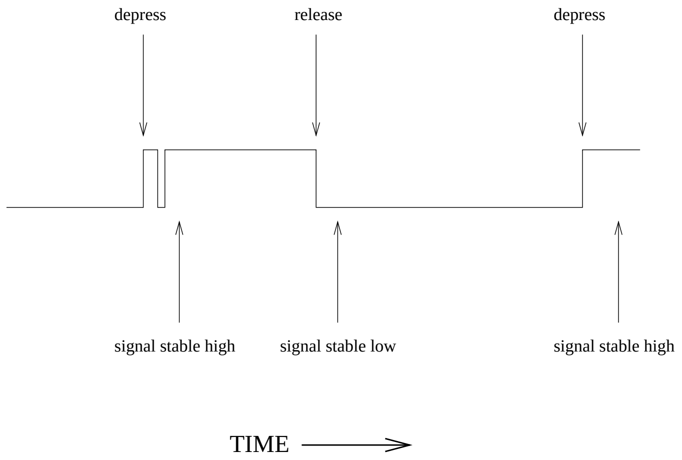

Figure 7.2: An example bouncy signal

is straightforward. When a particular processor memory location has value 0 then a corresponding light is on and when it has value 1 the light is off.

The button model is necessarily more complex. The kind of button we model produces a signal with "bounce", which means that when the contact is depressed or released the signal from the button is briefly unpredictable. Signal debouncing is a problem routinely faced by real-time software developers that is caused by the physical bounce of the contacts in the button [39, 47] . Figure 7.2 displays a signal generated by a bouncy button. After a depress occurs, the signal is not predictable for some time. After a known amount of time has elapsed the signal will be high and will stay high until the button is released. Similarly, after a release occurs, the signal is not predictable for some time. After some known amount of time has elapsed the signal will be low and will stay low until the button is depressed.

Our particular button has the following properties:

1. The button depresses and releases alternate, starting with a depress if any action occurs.

2. No system event — a button depress, button release, or the start of the system — occurs less than 4 milliseconds after another system event.

3. Signal = 0 after a button release or the system start following a period of signal instability of at most 3 milliseconds and until a button depress.

4. Signal = 1 after a button depress following a period of signal instability of at most 3 milliseconds and until a button release.

5. The value read from the button is 0 or 1.

Another important aspect of a specification of a real-time system containing a processor is a model of the processor execution. Chapter 5 presents a real-time model of the FM9001 processor that, for a given initial processor state, memory access time, list of input values during execution, and a list of relevant memory locations, produces a description of the operation. FM9001 input and output is accomplished using memory-mapped I/O.

What property do we wish for the quiz-show system? The lights should reflect the status of the buttons. We allow a very short delay because we have no alternative: the unpredictability of the button signals due to bounce and the delay from our processor necessitate our not requiring an immediate reaction. We shall require 4 milliseconds since it seems fast enough and appears achievable given the buttons and processor we are using.

1. If no button action has occurred within the last 4 milliseconds then if a button is up the corresponding light is off.

2. If no button action has occurred within the last 4 milliseconds then if a button is down a light is on.

3. If no button action has occurred within the past 4 milliseconds then both lights are not on.

Once a contestant has depressed his button and 4 milliseconds have elapsed, if the contestant is selected he should not be unselected until he releases his button.

4. If a light is on and the corresponding button has been down for the last 4 milliseconds, then the light remains on until the button is released

Note that this informal specification may at first appear not as strong as we might expect. For example, this specification allows light1 to be lit for a few milliseconds when button2 is released, even when button1 has not been depressed. We note that occasional pulses to our LEDs of less than 1/100 of a second are undetectable, and that weakening the "obvious" specification as we have done leads to a simpler formal specification. However, the reader should assure himself that these informal specifications really capture a reasonable notion of quiz-show correctness.

In the next section we describe an application that we shall prove implements this specification.

7.2 The FM9001 Quiz-Show Application

We verify that the particular bits that we shall load onto an FM9001 causes the system to have desirable behavior, and in this section we shall present those bits. Understanding how we generate them is not needed in order to understand what has been proved about the system. We nevertheless describe the assembly language program we use to generate machine code as well as other elements of the FM9001 state in order to document the work, to aid in later discussion of proofs of the correctness theorem, and to emphasize the concrete nature of the model of computation we are using.

Figure 7.3 presents an FM9001 assembly language program designed to solve the quiz-show problem. It assumes an I/O configuration that is consistent with the hardware design of the FM9001 single board computer discussed in Section 5.8. Roughly speaking, execution of this program does the following. The first seven instructions initialize the registers used to contain memory addresses and the memory locations associated with the output devices to 0. The program then enters an infinite loop in which it senses the buttons, updates a register in which the number of the currently-picked button is stored based on what button had been picked and the new button values, and updates the lights based on the selected button.

A simple FM9001 assembler was developed during the FM9001 verification effort. It has been modified slightly for this project in order to allow the generation of FM9001 machine code at arbitrary addresses. The semantics of FM9001 assembly language is not important to our project since we verify machine code, not assembly language instructions. The reader interested in the FM9001 assembler is referred to the FM9001 verification project report [14] and the minor modification to allow code relocation in the definition of 'my-asm' in Appendix E. The program in Figure 7.3 assembled for loading at memory address $\mathtt{40000000}_{\mathtt{16}}$ and converted to hexadecimal is listed in Figure 7.4.

7.3 A Formal Description of Correct Behavior

In order to describe precisely what constitutes desirable quiz-show system operation we must formalize several aspects of the system: the behavior

```
; r0: temp reg
; r1, r2: button values
; r3: pick (0= neither picked)
; r4, r5: button addresses
; r6, r7: light addresses

; load constants
(move t f r3 0)
(move t f r4 (pc+))
#xc0000000
(move t f r5 (pc+))
#xc0000002
(move t f r6 (pc+))
#xc0000001
(move t f r7 (pc+))
#xc0000003
(move t f (r6) 1)
(move t f (r7) 1)

loop
; read button values
(move t f r1 (r4))
(move t f r2 (r5))

; if button 2 currently picked handle separately
(xor f z r3 2)
(move eq f pc (pc+))
(value two-picked)

; button1 or no button selected.
(asr f c r1 r1)       ; set carry flag if b1 is high
(move t f r0 1)
(move cs f pc (pc+))
(value update-status)
(asr f c r2 r2)       ; set carry flag if b2 is high
(move t f r0 2)
(move cs f pc (pc+))
(value update-status)
(move t f r0 0)
(move t f pc (pc+))
(value update-status)

two-picked
; button2 selected.
(asr f c r2 r2)       ; set carry flag if b2 high
(move t f r0 2)
(move cs f pc (pc+))
(value update-status)
(asr f c r1 r1)       ; set carry flag if b1 high
(move t f r0 1)
(move cs f pc (pc+))
(value update-status)
(move t f r0 0)

update-status
(move t f r3 r0)

; update lights
(move t f r0 1)
(xor f z r3 1)
(move eq f r0 0)
(move t f (r6) r0)
(move t f r0 1)
(xor f z r3 2)
(move eq f r0 0)
(move t f (r7) r0)

(move t f pc (pc+))
(value loop)
```

Figure 7.3: FM9001 assembly code for quiz-show

list $(\texttt{00E00E00}_{16}$, $\texttt{00E0103F}_{16}$, $\texttt{C0000000}_{16}$, $\texttt{00E0143F}_{16}$, $\texttt{C0000002}_{16}$, $\texttt{00E0183F}_{16}$, $\texttt{C0000001}_{16}$, $\texttt{00E01C3F}_{16}$, $\texttt{C0000003}_{16}$, $\texttt{00E05A01}_{16}$, $\texttt{00E05E01}_{16}$, $\texttt{00E00414}_{16}$, $\texttt{00E00815}_{16}$, $\texttt{0BF10E02}_{16}$, $\texttt{00703C3F}_{16}$, $\texttt{4000001B}_{16}$, $\texttt{09F80401}_{16}$, $\texttt{00E00201}_{16}$, $\texttt{00103C3F}_{16}$, $\texttt{40000024}_{16}$, $\texttt{09F80802}_{16}$, $\texttt{00E00202}_{16}$, $\texttt{00103C3F}_{16}$, $\texttt{40000024}_{16}$, $\texttt{00E00200}_{16}$, $\texttt{00E03C3F}_{16}$, $\texttt{40000024}_{16}$, $\texttt{09F80802}_{16}$, $\texttt{00E00202}_{16}$, $\texttt{00103C3F}_{16}$, $\texttt{40000024}_{16}$, $\texttt{09F80401}_{16}$, $\texttt{00E00201}_{16}$, $\texttt{00103C3F}_{16}$, $\texttt{40000024}_{16}$, $\texttt{00E00200}_{16}$, $\texttt{00E00C00}_{16}$, $\texttt{00E00201}_{16}$, $\texttt{0BF10E01}_{16}$, $\texttt{00700200}_{16}$, $\texttt{00E05800}_{16}$, $\texttt{00E00201}_{16}$, $\texttt{0BF10E02}_{16}$, $\texttt{00700200}_{16}$, $\texttt{00E05C00}_{16}$, $\texttt{00E03C3F}_{16}$, $\texttt{4000000B}_{16})$

Figure 7.4: FM9001 machine code for quiz-show

of the buttons, the FM9001 with its initial state and I/O configuration, and the desired behavior of the system. We model the operation of the system using a history where each assignment represents the state of the system.

We shall informally think of the time elapsed between consecutive assignments in the history is the time between two ticks of the FM9001 clock, which in the case of the FM9001 single board computer is close to 10^{-7} seconds. However, our specification is in fact about clock ticks, not seconds. Since our formal notion of correctness involves the FM9001 clock we have avoided concern about clock inaccuracies.

The assignments of the history we use to model the system use several variables with which we describe the system. We describe these variables in the following sections that describe the parts of our model.

7.3.1 Bouncy Buttons

One element of the quiz-show system we model is button1. There are two values associated with button1 that we shall include in the history that describes the system.

signal1 is the value that can be read from the button.

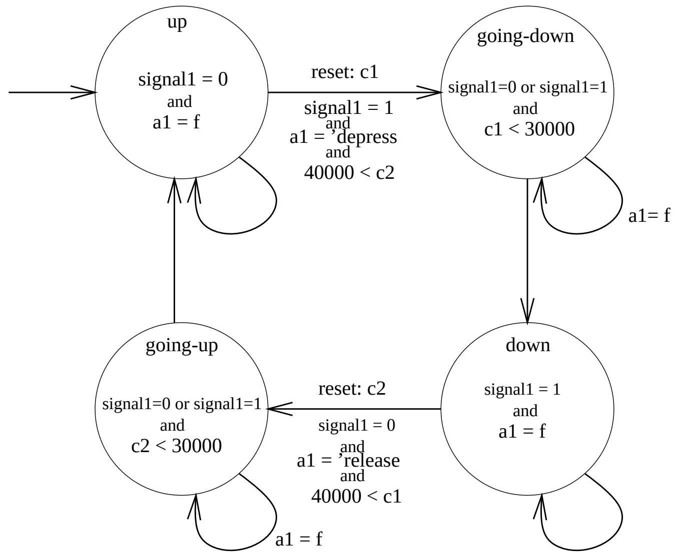

Figure 7.5: Model of button1

a1 has the value **depress** when the user depresses the button, **release** when he releases it, and **f** otherwise. (The name of variable *a1* abbreviates "action of button1".) The microcycle during which the button is depressed is the first microcycle that the button's signal is high after a stretch of low values, and the microcycle during which the button is released is the first microcycle that the button's signal is low after a stretch of high values.

Recall the informal button description in Section 7.1. Figure 7.5 presents a behavior that formalizes reasonable button behavior, and Figure 7.6 presents an Nqthm constant that encodes this behavior. We claim that a history that uses variables *a1* and *signal1* to describe button1 which works as suggested in Section 7.1 satisfies this behavior.

The behavior that describes the operation of button1 is our model of that button, and as such there is no sense in which we can "prove" that it is in fact satisfied by all histories describing reasonable operation of button1.

```
(defn button1-spec ()
  '(up
    ((up
      (and (equal signal1 0) (equal a1 'f))
      ((up () (true))
       (going-down ((c 1))
         (and (equal signal1 1)
              (and (equal a1 'depress)
              (lessp 40000 (c 2)))))))
     (going-down
      (and (or (equal signal1 0) (equal signal1 1))
           (lessp (c 1) 30000))
      ((going-down () (equal a1 'f)) (down () (true))))
     (down
      (and (equal signal1 1)  (equal a1 'f))
      ((down () (true))
       (going-up ((c 2))
         (and (equal signal1 0)
              (and (equal a1 'release) (lessp 40000 (c 1)))))))
     (going-up
      (and (or (equal signal1 0) (equal signal1 1))
           (lessp (c 2) 30000))
      ((going-up () (equal a1 'f)) (up () (true)))))))
```

Figure 7.6: Encoding of the button1 model

Even so, we argue that this model is reasonable. The argument is informal because we are attempting to show that something informal has been correctly described in a behavior. Recall from our development of behaviors in Chapter 6 the definition of 'satisfies', that there exists a mapping from assignments in the history to states in the behavior starting with the initial state such that all the relevant predicates hold on the variable values and calculated counter values. We argue that for any history h with reasonable button operation there exists such a path in the behavior by showing how to construct this mapping and then by using the informal properties of the button to show that the relevant predicates hold.

First we describe how to construct the mapping m from assignments in h to states in the button1 model behavior. Let the elements of h be denoted $h_0, h_1 ... h_n$. If no element of h associates *signal1* with 1 then m maps each element of h to **up**. Otherwise let h_d be the first element of h that associates *signal1* with 1. Let m map each element h_i where $i < d$ to **up** and each element h_i where $d \leq$ i $\leq d$+30000 to **going-down**. If there exists no element h_r that associates *signal1* with 0 where $r > d$+30000 then m maps the remaining elements of h to **down**. Otherwise m maps each element h_i where d+30000 $\leq i$ $< r$ to **down** and each element h_i where $r \leq$ i $\leq r$+30000 to **going-up**. Repeat this construction starting with element $h_{r+30001}$ until m maps all elements of h.

We show from the informal button properties in Section 7.1 that mapping m determines a path through the button model behavior in which each of the relevant predicates holds as required in the definition of 'satisfiesp'. We first note that the elements of the predicates of the states involving the variable *a1* are satisfied by the construction of m and button properties 1 and 3, so we shall focus our informal argument on the other elements of the predicates. The path associated with the initial sequence of assignments mapped to **up** satisfies the state predicate of state **up** by construction of m and satisfies the transition predicates trivially. The transition predicate from the (d-1)th element to the dth element is satisfied because of button property 2. The path associated with the sequence of assignments h_d to $h_{d+30000}$ satisfies the state predicate of **going-down** from button property 5 and the transition predicates are satisfied trivially. The path associated with the sequence of assignments $h_{d+30001}$ to h_r satisfies the state predicate of **down** from button property 4. That the sequence of assignments when button1 is released satisfies the behavior follows using analogous reasoning. Continued application of the informal button properties in this manner guarantees that the path described by m is a path through the behavior that shows that h satisfies it. We conclude that if h represents a history describing reasonable button operation then h satisfies the button model.

We model button2 as a button with the same properties as button1. We use a behavior that is identical to that of button1 except that the variables and counters are renamed. This behavior is presented in Figure 7.7 and its encoding is presented in Figure 7.8.

7.3.2 A History Reflects Configured FM9001 Operation

We formalize in the Nqthm logic that a history is consistent with the operation of an FM9001 that has the I/O configuration of the board we use to execute the quiz-show application. We use the model of the FM9001 presented in Chapter 5. In order to accomplish this we define two new Nqthm functions, 'extension-historyp' and 'extract-from-history', which we present with two subsidiary definitions in Figure 7.9. Assignment *a2* is an extension of assignment

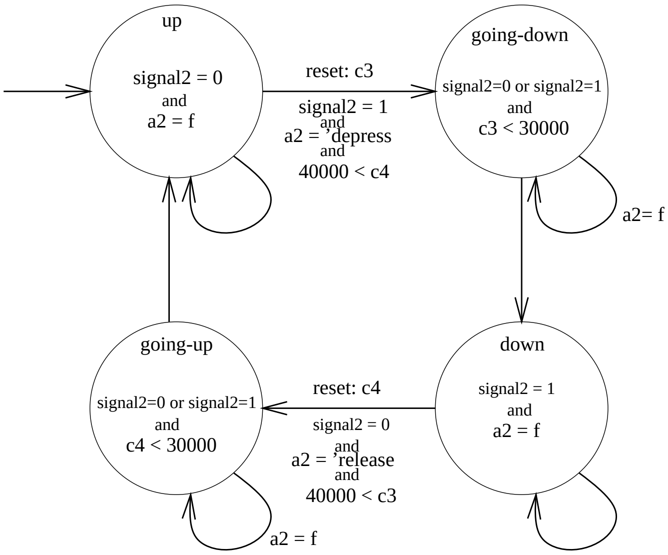

Figure 7.7: Model of button2

```
(defn button2-spec ()
  '(up
    ((up
      (and (equal signal2 0) (equal a2 'f))
      ((up () (true))
       (going-down ((c 3))
         (and (equal signal2 1)
              (and (equal a2 'depress) (lessp 40000 (c 4)))))))
     (going-down
      (and (or (equal signal2 0) (equal signal2 1))
           (lessp (c 3) 30000))
      ((going-down () (equal a2 'f)) (down () (true))))
     (down
      (and (equal signal2 1)  (equal a2 'f))
      ((down () (true))
       (going-up ((c 4))
         (and (equal signal2 0)
              (and (equal a2 'release) (lessp 40000 (c 3)))))))
     (going-up
      (and (or (equal signal2 0) (equal signal2 1))
           (lessp (c 4) 30000))
      ((going-up () (equal a2 'f)) (up () (true)))))))
```

Figure 7.8: Encoding of the button2 model

Definition:
extension-assignmentp $(a1,\ a2)$
$=$ **if** listp $(a1)$
then (cadr (assoc (caar $(a1)$, $a2$)) = cadar $(a1)$)
$\land$ extension-assignmentp (cdr $(a1)$, $a2$)
else t endif

Definition:
extension-historyp $(h1,\ h2)$
$=$ **if** listp $(h1)$
then listp $(h2)$
$\land$ extension-assignmentp (car $(h1)$, car $(h2)$)
$\land$ extension-historyp (cdr $(h1)$, cdr $(h2)$)
else $h2 \simeq$ **nil endif**

Definition:
extract-from-assignment (*list*, a)
$=$ **if** listp (*list*)
then cons (list (caar (*list*), cadr (assoc (cadar (*list*), a))),
extract-from-assignment (cdr (*list*), a))
else nil endif

Definition:
extract-from-history (*list*, h)
$=$ **if** listp (h)
then cons (extract-from-assignment (*list*, car (h)),
extract-from-history (*list*, cdr (h)))
else nil endif

Figure 7.9: Useful functions for formalizing FM9001 execution properties

a1 when each variable associated with a value in *a1* is associated with the same value in *a2*. History *h2* is an extension of history *h1* when it is the same length and each assignment of *h2* is an extension of the corresponding assignment in *h1*. We use the Nqthm execution facility to demonstrate this definition on examples below.

```
*(extension-historyp
   '(((y 2))        ((x off))       nil             ((x on) (y 2)))
   '(((x on) (y 2)) ((x off) (y 1)) ((x off) (y 2)) ((x on) (y 2))))
 T
*(extension-historyp
   '(((y 2))        ((x off))       nil             ((x on) (y 2)))
```

```
   '(((x on))       ((x off) (y 1)) ((x off) (y 2)) ((x on) (y 2))))
F
```

The function 'extract-from-history' takes as arguments a list of values to be extracted and a history. Each of the to-be-extracted values is a pair containing a new name and the current name of the value to be extracted. We demonstrate this function on an example below.

```
*(extract-from-history
   '((z x))
   '(((x on) (y 2)) ((x off) (y 1)) ((x off) (y 2)) ((x on) (y 2))))
 '(((Z ON)) ((Z OFF)) ((Z OFF)) ((Z ON)))
```

Recall from Section 7.1 that the button values are read by the FM9001 at memory locations $\mathtt{c0000000}_{16}$ and $\mathtt{c0000002}_{16}$ and the lights are controlled at memory locations $\mathtt{c0000001}_{16}$ and $\mathtt{c0000003}_{16}$, and also recall our development of an Nqthm model of FM9001 processor operation in Chapter 5. Figure 7.10 presents an Nqthm term that formalizes that a history h is consistent with regard to the two output memory locations when starting in state s with memory access time 3 and two inputs. The names in h correspond to memory in the following way: *signal1* corresponds to input location $\mathtt{c0000000}_{16}$, *signal2* to input location $\mathtt{c0000002}_{16}$, *l1* to output location $\mathtt{c0000001}_{16}$, and *l2* to output location $\mathtt{c0000003}_{16}$. In this term each microcycle of the inputs-list to the FM9001 model updates exactly the memory locations corresponding to the relevant input devices, in accordance with our assumption in Chapter 5 that the I/O configuration remains unchanged during execution and that each microcycle of input presented to the model has a value for each input device.

History h satisfies the term in Figure 7.10 when variables *l1* and *l2* have values that match exactly the values for memory locations $\mathtt{c0000001}_{16}$ and $\mathtt{c0000003}_{16}$ that our FM9001 model produces with initial state s, memory

```
extension-historyp (fm9001-rt-history (s,
                                       3,
                                       extract-from-history ('((c0000000₁₆
                                                                signal1)
                                                               (c0000002₁₆
                                                                signal2)),
                                                             h),
                                       '((c0000001₁₆ l1)
                                         (c0000003₁₆ l2))),
                   h)
```

Figure 7.10: History consistency with processor execution

access time 3, and inputs at locations $\texttt{c0000000}_{16}$ and $\texttt{c0000002}_{16}$ that match the values of *signal1* and *signal2*.

7.3.3 The Initial FM9001 State

We formalize when an FM9001 state is a reasonable initial state for the quiz-show system using Nqthm function ‘good-initial-fm9001-quiz-statep’, a predicate that returns whether its argument is an FM9001 state with the following properties:

- The loop of the program presented in Figure 7.4 is loaded in memory beginning at location $\texttt{4000000b}_{16}$.
- The memory-mapped input locations are 32-bit RAM. (The kind of memory at a memory location is a part of the FM9001 model.
- The memory-mapped output locations are RAM.
- Registers 0, 1, 2, 3, and 15 are 32-bit RAM.
- Registers 4, 5, 6, and 7 have as values the addresses of the locations used for memory-mapped I/O.

- Registers 1, 2, and 3 have value 0.

- Register 15 contains the value $\texttt{4000000b}_{16}$.

- The value of the memory locations $\texttt{c0000001}_{16}$ and $\texttt{c0000003}_{16}$ is 1.

The definition of 'good-initial-fm9001-quiz-statep' is complex, so we create an FM9001 state that contains our application and other appropriate values and demonstrate that the definition is satisfiable by showing that this example is a good initial state. This exercise assures that the function is satisfiable and holds of at least one reasonable FM9001 state. The Nqthm constant QUIZ-INITIAL-STATE has the list of values in Figure 7.4 loaded at location $\texttt{40000000}_{16}$ and correctly-sized RAM in the places we expect for the quiz application. We calculate that this FM9001 state, after execution of 7 instructions with register 15 designated the program counter, satisfies the good-state predicate.

```
*(good-initial-fm9001-quiz-statep
 (fm9001-step (fm9001-step (fm9001-step (fm9001-step
  (fm9001-step (fm9001-step (fm9001-step
    (quiz-initial-state)
    (nat-to-v 15 4)) (nat-to-v 15 4)) (nat-to-v 15 4))
  (nat-to-v 15 4)) (nat-to-v 15 4)) (nat-to-v 15 4)) (nat-to-v 15 4)))
 T
```

7.3.4 Desired Behavior

We specify the desired behavior of the quiz-show system using our formalism for describing abstract properties of real-time systems. In Section 7.1 we present informally the properties we wish for the quiz-show system, and in this section we present a behavior such that any satisfying history has those properties.

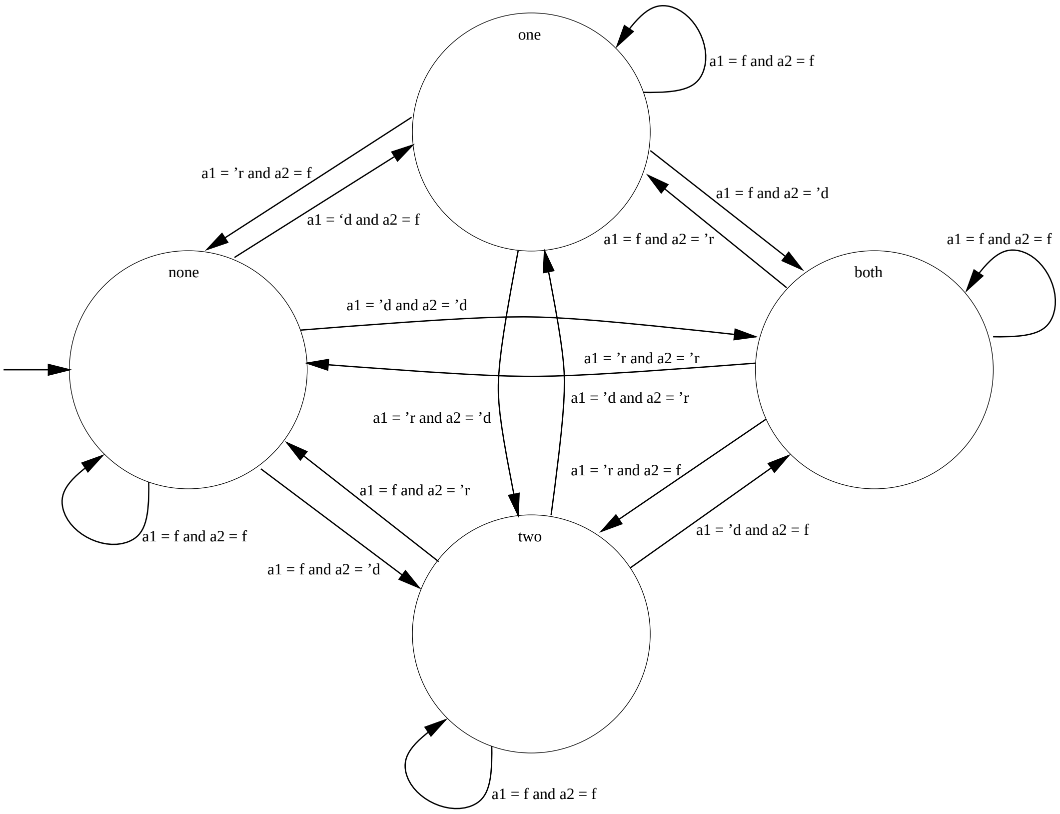

Figure 7.11: Quiz example specification structure

As with our formalization of the button model, the relationship between our formalization of the desired properties and the "real world" is not subject to proof. The informal properties we wish to maintain are — informal. We argue that there is a correspondence between our formalization and an intuitive notion of correct system behavior, but the appropriateness of this specification of desired behavior is not a matter of right and wrong but rather one of judgment.

The behavior we shall present as a formalization of the specification for the quiz-show system is complex, so for pedagogical reasons we present several behaviors that have the same structure whose predicates and counter resets when combined form the complete behavior. Figure 7.11 presents the first such behavior. In the diagrams related to the quiz-show specification DEPRESS is abbreviated D and RELEASE is abbreviated R. The behavior has four states that correspond to the four possible combinations of most-recent actions of the two buttons. The names of the states — **none**, **one**, **two**, and **both** — signify which of the buttons has had as its most recent action a depress. Each state has a transition leading to each state in the behavior, and each of these transitions has a predicate that requires that the button actions are consistent with the expected interpretation of that transition. We use the variables *a1* and *a2* to represent actions on button1 and button2 in the manner described in the button model presentation in Section 7.3.1.

As an example, note the transition from **one** to **both** in Figure 7.11. If two consecutive assignments in a satisfying history are mapped to these states in that order then the second assignment must associate **f** with *a1* and DEPRESS with *a2*. This is consistent with the informal understanding of satisfying histories suggested by the names of the states of the behavior.

Our ultimate aim is of course to relate the values of the lights of the quiz-show system to the most-recent actions on the buttons. Figure 7.12 presents a behavior that has constraints that ensure that the light values of satisfying histories are consistent with the most-recent actions on the buttons. Variable *l1* represents the value in the memory location corresponding to light1 and *l2* represents the value in the memory location corresponding to light2. Recall that value 1 means off and value 0 means on and that each time unit represents about 10^{-7} seconds. For each state, either the lights have values that are consistent with the most recent button actions, or the most recent action on one of the buttons occurred within 4 milliseconds.

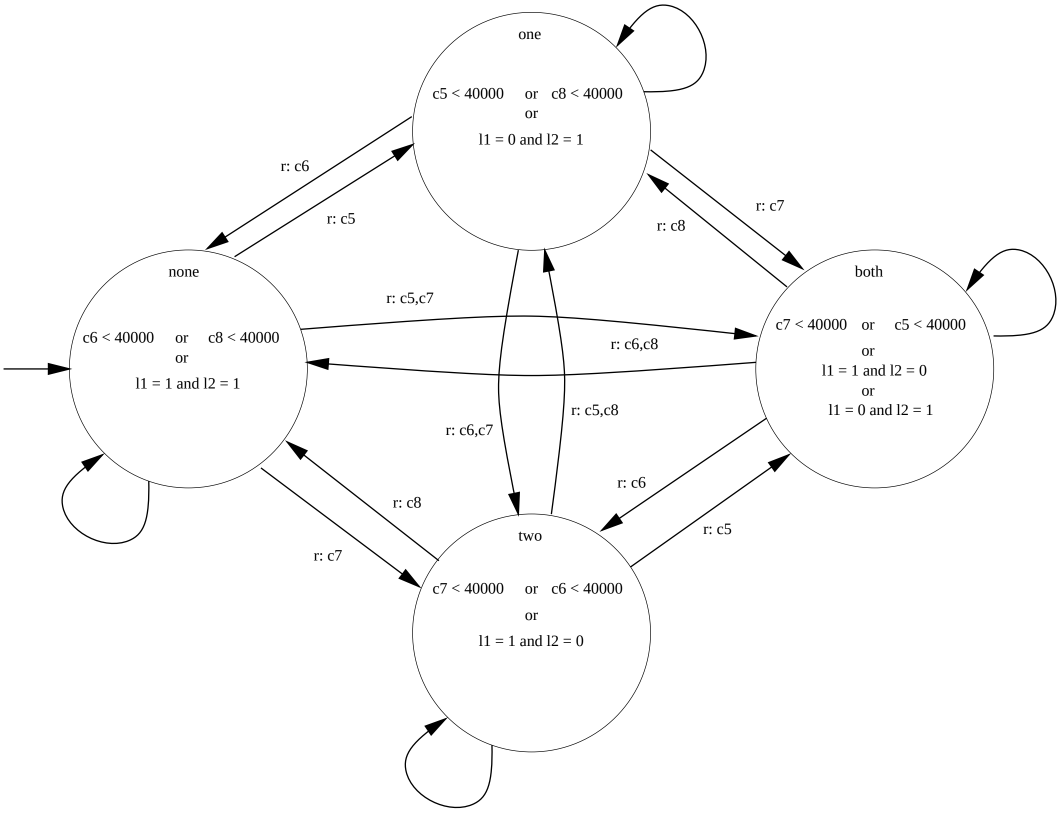

Figure 7.12: Quiz example lights constraints

As an example, note the state predicate of **one** in Figure 7.12. The counter (c 5) is reset on transitions that lead to states that correspond to assignments that associate DEPRESS with variable *a1* and is not reset on other transitions. So, for purposes of deciding whether a history satisfies this behavior, we can think of the value of counter (c 5) as the number of time units since button1 was depressed. (Or since the beginning of the history, if button1 has not been depressed.) Similarly, we think of (c 8) as the number of time units since button2 was released. If h is a history satisfying this behavior, then an assignment mapped to **one** either has light1 on and light2 off or the value of the counters indicates that button1 has been recently depressed or button2 recently released.

We wish the quiz-show system to have the property that once a button that has been depressed for at least 4 milliseconds is selected — that is, once the corresponding light is turned on — then the light remains on so long as the button is not released. Figure 7.13 presents a behavior with constraints that ensure that satisfying histories have this property. As an example, note the transitions in Figure 7.13 that begin and end in a state in which the most recent action on button1 is a depress. These are the transitions that begin and end in either state **one** or **both**. Each of these transitions has a transition predicate that requires that if the assignment corresponding to the original state has light1 on and the counter values indicate that the most recent depress of button1 was more than 4 milliseconds ago, then the next assignment also has light1 on. So, if button1 has been depressed for a while and light1 is on, light1 will remain on as long as button1 is not released.

Figure 7.14 presents a behavior that is identical in structure to the behaviors presented in this section where each of the predicates is the conjunction

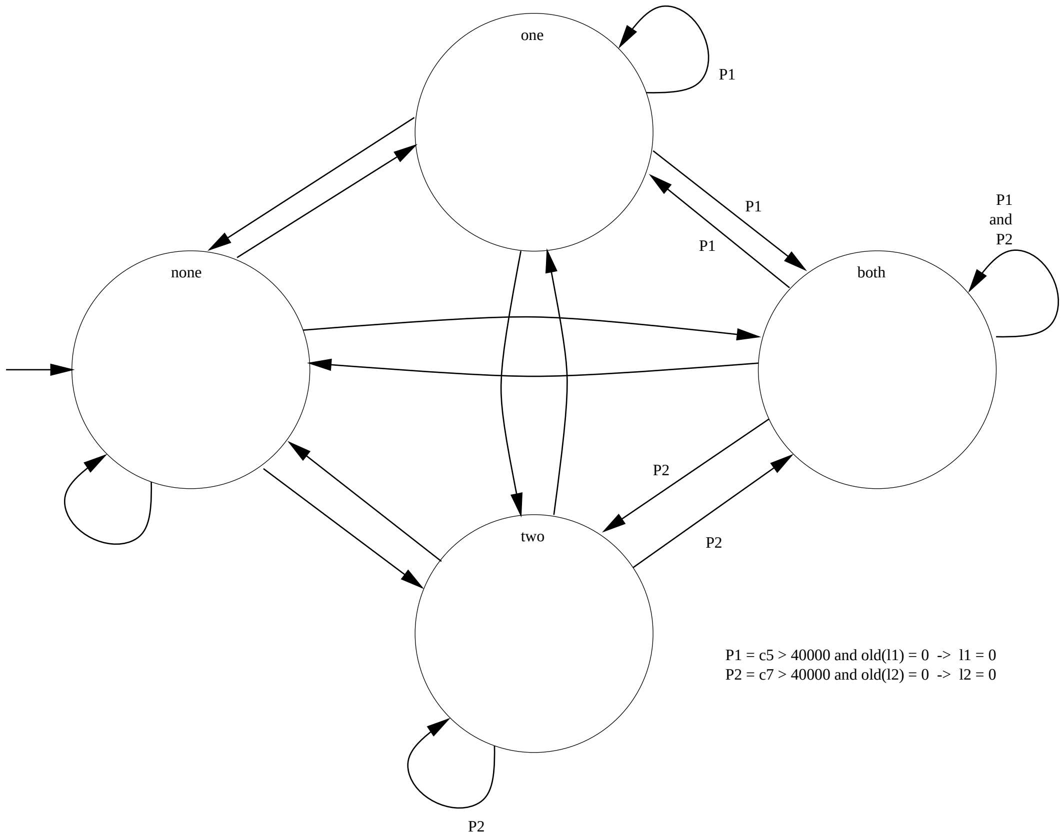

Figure 7.13: Quiz show steadiness constraint

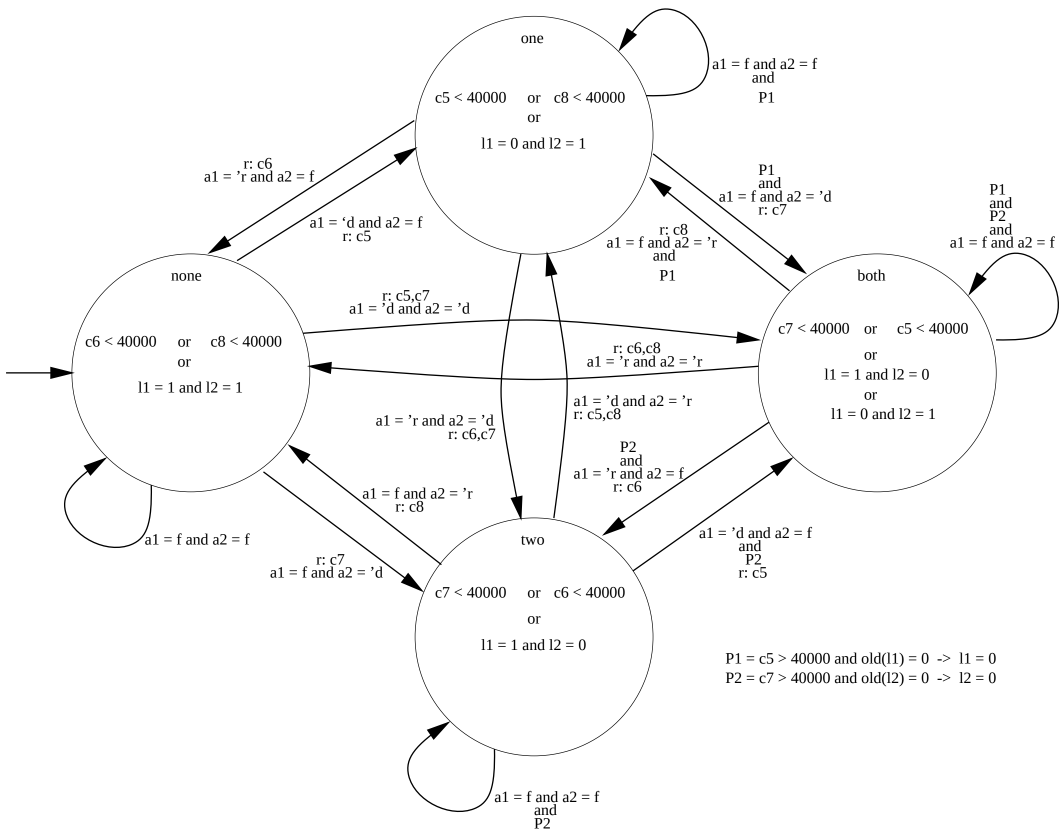

Figure 7.14: Quiz show specification

```
(defn quiz-spec ()
  '(none
    ((none
      (or (lessp (c 6) 40000) (or (lessp (c 8) 40000)
          (and (equal l1 1) (equal l2 1))))
      ((none () (and (equal a1 'f) (equal a2 'f)))
       (one ((c 5)) (and (equal a1 'depress) (equal a2 'f)))
       (two ((c 7)) (and (equal a1 'f) (equal a2 'depress)))
       (both ((c 5) (c 7)) (and (equal a1 'depress) (equal a2 'depress)))))
     (one
      (or (lessp (c 5) 40000) (or (lessp (c 8) 40000)
          (and (equal l1 0) (equal l2 1))))
      ((none ((c 6)) (and (equal a1 'release) (equal a2 'f)))
       (one () (and (equal a1 'f) (and (equal a2 'f)
                (or (not (and (lessp 40000 (c 5)) (equal (old l1) 0)))
                    (equal l1 0)))))
       (two ((c 6) (c 7)) (and (equal a1 'release) (equal a2 'depress)))
       (both ((c 7)) (and (equal a1 'f) (and (equal a2 'depress)
                 (or (not (and (lessp 40000 (c 5)) (equal (old l1) 0)))
                     (equal l1 0)))))))
     (two
      (or (lessp (c 7) 40000) (or (lessp (c 6) 40000)
          (and (equal l1 1) (equal l2 0))))
      ((none ((c 8)) (and (equal a1 'f) (equal a2 'release)))
       (one ((c 5) (c 8)) (and (equal a1 'depress) (equal a2 'release)))
       (two () (and (equal a1 'f) (and (equal a2 'f)
                (or (not (and (lessp 40000 (c 7)) (equal (old l2) 0)))
                    (equal l2 0)))))
       (both ((c 5)) (and (equal a1 'depress) (and (equal a2 'f)
                (or (not (and (lessp 40000 (c 7)) (equal (old l2) 0)))
                    (equal l2 0)))))))
     (both
      (or (lessp (c 7) 40000) (or (lessp (c 5) 40000)
      (or (and (equal l1 1) (equal l2 0)) (and (equal l1 0) (equal l2 1)))))
      ((none ((c 6) (c 8)) (and (equal a1 'release) (equal a2 'release)))
       (one ((c 8)) (and (equal a1 'f) (and (equal a2 'release)
                (or (not (and (lessp 40000 (c 5)) (equal (old l1) 0)))
                    (equal l1 0)))))
       (two ((c 6)) (and (equal a1 'release) (and (equal a2 'f)
                (or (not (and (lessp 40000 (c 7)) (equal (old l2) 0)))
                    (equal l2 0)))))
       (both () (and (equal a1 'f) (and (equal a2 'f)
            (and (or (not (and (lessp 40000 (c 5)) (equal (old l1) 0)))
                     (equal l1 0))
                 (or (not (and (lessp 40000 (c 7)) (equal (old l2) 0)))
                    (equal l2 0)))))))))))
```

Figure 7.15: Nqthm encoding of the quiz-show desired behavior

```
(good-initial-fm9001-quiz-statep (s)
 ∧  satisfiesp (h, BUTTON1-SPEC)
 ∧  satisfiesp (h, BUTTON2-SPEC)
 ∧  extension-historyp (fm9001-rt-history (s,
                                           3,
                                           extract-from-history ('((c0000000_16
                                                                    signal1)
                                                                   (c0000002_16
                                                                    signal2)),
                                                                 h),
                                           '((c0000001_16 l1)
                                             (c0000003_16 l2))),
                        h))
 →  satisfiesp (h, QUIZ-SPEC)
```

Figure 7.16: Quiz-show correctness theorem

of the predicates of those behaviors. We use this behavior as the formalization of what is expected in the quiz-show system. Figure 7.15 presents an Nqthm constant that is the encoding of this behavior. We claim that any history that satisfies the behavior presented in Figures 7.14 and 7.15 describes the operation of a system that meets the informal requirements in Section 7.1 for the quiz-show system. This is the converse of the claim about the button model, but like the button model claim it is not possible to "prove" since it is an informal statement. The state predicates of the behavior in Figure 7.14 require that a satisfying history achieves the first three informal goals of the system, and the transition predicates require that a satisfying history achieves the final goal.

7.3.5 A Correctness Theorem

Figure 7.16 presents a conjecture about the quiz-show system that makes use of the formalizations developed in this section. Hypothesis 1 requires variable s is a reasonable starting state for the quiz-show program. Variables *signal1* and *a1* in h are constrained in hypothesis 2 to reflect normal operation of button1 and variables *signal2* and *a2* in *h* are constrained in hypothesis 3 to reflect normal operation of button2. Variables *l1* and *l2* in *h* are constrained to be exactly what the FM9001 places in memory locations $\texttt{c0000001}_{16}$ and $\texttt{c0000003}_{16}$ respectively when executing beginning in state *s* with memory access time 3 and input values in locations $\texttt{c0000000}_{16}$ and $\texttt{c0000002}_{16}$ of *signal1* and *signal2* respectively. The conclusion guarantees that *l1*, *l2*, *a1*, and *a2* have the relationship we desire for the system.

The conjecture has been proved using Nqthm, and its proof is outlined in the next section. Since this theorem has been proved and it formalizes the notion of correct operation we desire for the quiz-show system, we say that the system is verified.

7.4 Guide to the Machine-checked Quiz-show Proof

Nqthm has been used to prove that the conjecture in Figure 7.16 is a theorem and the input to Nqthm that leads it to the proof of the correctness theorem is listed in Appendix E. Since Nqthm-checked proofs are generally considered reliable, and we discussed in previous sections why this correctness theorem is valuable for guaranteeing the behavior of the quiz-show system, the reader interested only in the verification of this one example need read no further to be assured that the quiz-show system is correct.

While the use of a mechanical theorem-prover to check a proof adds greatly to our confidence in its correctness, the completeness and precision of a mechanically-checked proof can obscure the proof structure. This section provides a guide to the proof by identifying the more-important subsidiary lemmas. In later sections we describe aspects of the proofs of these lemmas. We give Nqthm event names associated with some of the theorems so that the

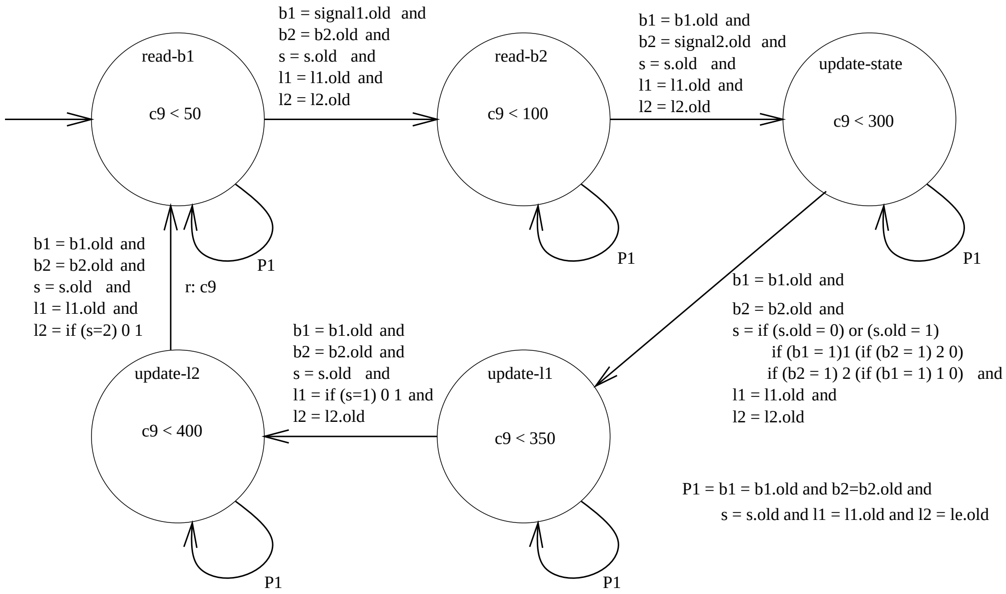

Figure 7.17: Quiz program specification

reader may find them (using the index) in Appendix E.

We introduce a behavior and a definition that are useful for factoring the correctness theorem into smaller, component lemmas. Figure 7.17 presents a behavior we call the quiz-show program specification. The Nqthm function 'quiz-program-operation' is an Nqthm function that calculates values just as the FM9001 does when executing the quiz-show application. Three main lemmas are proved in order to establish the correctness conjecture.

- The **abstract system** lemma states that the correctness theorem holds with the added assumption that history h satisfies the program specification instead of the FM9001-related assumptions.

- The **FM9001 reasonableness** lemma states that the FM9001 executing the application works as predicted by 'quiz-program-operation'.

$$
\begin{array}{l}
(((h \simeq \mathbf{nil}) \\
\quad \lor \quad ((\mathrm{cadr}\,(\mathrm{assoc}\,(\texttt{'s},\, \mathrm{car}\,(h))) = \texttt{0}) \\
\qquad\quad \land \quad (\mathrm{cadr}\,(\mathrm{assoc}\,(\texttt{'l1},\, \mathrm{car}\,(h))) = \texttt{1}) \\
\qquad\quad \land \quad (\mathrm{cadr}\,(\mathrm{assoc}\,(\texttt{'l2},\, \mathrm{car}\,(h))) = \texttt{1}))) \\
\ \land \quad \mathrm{satisfiesp}\,(h,\, \textsc{button1-spec}) \\
\ \land \quad \mathrm{satisfiesp}\,(h,\, \textsc{button2-spec}) \\
\ \land \quad \mathrm{satisfiesp}\,(h,\, \textsc{quiz-program-spec})) \\
\rightarrow \quad \mathrm{satisfiesp}\,(h,\, \textsc{quiz-spec})
\end{array}
$$

Figure 7.18: Abstract system lemma

- The **FM9001 program** lemma states that the history generated by 'quiz-program-operation' satisfies the program specification.

We describe each of these lemmas.

7.4.1 Abstract System Lemma

Figure 7.18 presents the abstract system lemma, which is prove-lemma event 'abstract-quiz-system-correctness' in Appendix E. This lemma is identical to the final correctness lemma except that the FM9001-related hypotheses have been replaced with a hypothesis about the values of the initial assignment of h and the hypothesis that h satisfies the program specification presented in Figure 7.17.

From the definition of 'satisfiesp' and the hypotheses involving the button models there exists mappings from the assignments in h to the states of the button model behaviors that have certain properties. We introduce a definition of a "state mapping" function that calculates using these mappings of assignments in h to states in BUTTON1-SPEC and BUTTON2-SPEC a mapping of assignments in h to the states of QUIZ-SPEC that generates a path in QUIZ-SPEC. We prove that this path demonstrates that the conclusion of the theorem holds. The definition of the state mapping function is straightforward and is presented in definition ‘abstract-quiz-map’. Given lists of the names of the states in the button specifications we calculate the appropriate state in the behavior describing the quiz-show’s desired properties. Note, for example, that when both buttons have been most recently depressed we select the state **both**.

7.4.2 FM9001 Reasonableness Proof

The real-time model of the FM9001 we use in our proof is complex, as it describes what happens to every bit of the FM9001 processor for every microcycle for every possible instruction. The FM9001 state is also rather large. Applications typically use processors in far simpler ways: much of memory is unused, programs don’t overwrite themselves so the program counter value determines what instruction is to be executed, etc. As with the code correctness proofs described in Section 4.6, we can simplify the proof by dispensing with the complexity of the FM9001 model.

We define the Nqthm function ‘quiz-program-operation’ that mimics the operation of the FM9001 when executing the quiz-show application and has 11 arguments that represent values from the FM9001 state and the inputs. These arguments are:

1. A natural number representation of the program counter’s value.
2. A natural number representation of register 3’s value.
3. A natural number representation of register 1’s value.
4. A natural number representation of register 2’s value.
5. A list of naturals representing the values of *signal1*.

```
(quiz-program-functioning-invariants (s)
 ∧   variable-not-overflowed-in-historyp ('signal1, h)
 ∧   variable-not-overflowed-in-historyp ('signal2, h)
 ∧   (mat = 3))
→   (fm9001-rt-history (s, mat,
                          extract-from-history ('((3221225472 signal1)
                                                  (3221225474 signal2)),
                                                h),
                          '(((15) pc) ((3) s) ((1) b1) ((2) b2)
                            (3221225472 signal1) (3221225474 signal2)
                            (3221225473 l1) (3221225475 l2)))
     =   quiz-program-operation
          (v-to-nat (read-mem (nat-to-v (15, 4), regs (car (s)))),
           v-to-nat (read-mem (nat-to-v (3, 4), regs (car (s)))),
           v-to-nat (read-mem (nat-to-v (1, 4), regs (car (s)))),
           v-to-nat (read-mem (nat-to-v (2, 4), regs (car (s)))),
           list-of-cadrs (list-of-cars (extract-from-history ('((3221225472 signal1)),
                                                              h))),
           list-of-cadrs (list-of-cars (extract-from-history ('((3221225474 signal2)),
                                                              h))),
           v-to-nat (read-mem (nat-to-v (3221225473, 32), cadr (s))),
           v-to-nat (read-mem (nat-to-v (3221225475, 32), cadr (s))),
           v-to-nat (read-mem (nat-to-v (0, 4), regs (car (s)))),
           z-flag (flags (car (s))),
           c-flag (flags (car (s)))))
```

Figure 7.19: FM9001 reasonableness lemma

6. A list of naturals representing the values of *signal2*.

7. A natural number representation of memory location $\texttt{c0000001}_{16}$'s value.

8. A natural number representation of memory location $\texttt{c0000003}_{16}$'s value.

9. A natural number representation of register 0's value.

10. A boolean representing the status of the processor "z" flag.

11. A boolean representing the status of the processor "c" flag.

We prove the FM9001 reasonableness lemma that appears as event 'fm9001-quiz-embedded' and which is presented in Figure 7.19. The hypotheses

$$\begin{array}{ll} & (\text{quiz-program-correctness-invariants}\,(pc,\ s,\ b1,\ b2,\ temp,\ z,\ c,\ \texttt{0}) \\ \wedge & (pc = \texttt{4000000b}_{\texttt{16}}) \\ \wedge & \text{list-of-zero-and-one}\,(s1\text{-}list) \\ \wedge & \text{list-of-zero-and-one}\,(s2\text{-}list)) \\ \rightarrow & \text{satisfiesp}\,(\text{quiz-program-operation}\,(pc,\ s,\ b1,\ b2,\ s1\text{-}list,\ s2\text{-}list,\ l1,\ l2,\ temp,\ z,\ c), \\ & \qquad\qquad \textsc{quiz-program-spec}) \end{array}$$

Figure 7.20: FM9001 program correctness lemma

of the theorem include the requirement that the values of variables *signal1* and *signal2* are naturals representable in 32 bits in history *h*.

An implication of the FM9001 reasonableness lemma is that the 11 elements listed above are all that is necessary to characterize the state of the computation during execution of the quiz-show application. The value of an element of memory containing a word of the program, for example, is irrelevant because it does not change. Processor register 9, for example, is irrelevant because it is never used. The state of the quiz-show application, like most applications, can be characterized in a much simpler manner than with the entire processor state, and the reasonableness lemma shows how to do that.

7.4.3 FM9001 Program Proof

Figure 7.20 presents the quiz-show program correctness lemma, which appears as 'satisfiesp-quiz-program-operation'. Assuming the lists of inputs are composed of zeros and ones and the variables that represent the FM9001 state have some properties we expect in the FM9001 state, the function 'quiz-program-operation' generates a history that satisfies the program specification of Figure 7.17.

We describe how one maps assignments generated by executing 'quiz-program-operation' to states in the program specification so that we may show that the behavior is satisfied. We calculate using the function 'quiz-program-spec-map-list' the appropriate state based on the value of the program counter. The mapping from program counter values of the FM9001 state to states in the program specification that is used by that function is listed below.

$\texttt{4000000b}_{16}$: read-b1
$\texttt{4000000c}_{16}$: read-b2
$\texttt{4000000d}_{16}$, $\texttt{4000000e}_{16}$, $\texttt{40000010}_{16}$, $\texttt{40000011}_{16}$, $\texttt{40000012}_{16}$, $\texttt{40000014}_{16}$, $\texttt{40000015}_{16}$, $\texttt{40000016}_{16}$, $\texttt{40000018}_{16}$, $\texttt{40000019}_{16}$, $\texttt{4000001b}_{16}$, $\texttt{4000001c}_{16}$, $\texttt{4000001d}_{16}$, $\texttt{4000001f}_{16}$, $\texttt{40000020}_{16}$, $\texttt{40000021}_{16}$, $\texttt{40000023}_{16}$, $\texttt{40000024}_{16}$: update-state
$\texttt{40000025}_{16}$, $\texttt{40000026}_{16}$, $\texttt{40000027}_{16}$, $\texttt{40000028}_{16}$: update-l1
$\texttt{40000029}_{16}$, $\texttt{4000002a}_{16}$, $\texttt{4000002b}_{16}$, $\texttt{4000002c}_{16}$: update-l2
$\texttt{4000002d}_{16}$: read-b1

This mapping allows proof of the program correctness lemma.

7.4.4 Deriving the Final Theorem

The lemmas of the previous section combine easily to prove the lemma in Figure 7.21. Roughly speaking, these lemmas presented in Figures 7.18, 7.19, and 7.20 constitute the ultimate goal, the correctness theorem in Figure 7.16, except that h is assumed to be an extension of the FM9001 execution in which the FM9001 model calculates values for extra elements of the FM9001 state. The extra elements are *signal1*, *signal2*, *pc*, *s*, *b1*, and *b2*. The variables *signal1* and *signal2* can be eliminated in the output list of the FM9001 model whose output is assumed extended by h because, given our assumptions about the FM9001 state, it is redundant to claim that h is an extension of the model whose output includes the values of the input locations that correspond to the values of *signal1* and *signal2* in h. We can justify the elimination of *pc*, *s*, *b1*, and *b2* from the output list since those variables are not mentioned elsewhere

$$
\begin{array}{ll}
\multicolumn{2}{l}{(\text{good-initial-fm9001-quiz-statep}\,(s)} \\
\wedge & \text{satisfiesp}\,(h,\ \textsc{button1-spec}) \\
\wedge & \text{satisfiesp}\,(h,\ \textsc{button2-spec}) \\
\wedge & \text{extension-historyp}\,(\text{fm9001-rt-history}\,(s, \\
& \qquad\qquad \texttt{3}, \\
& \qquad\qquad \text{extract-from-history}\,(\texttt{'((c0000000}_{16}\ \texttt{signal1)}\ \texttt{(c0000002}_{16}\ \texttt{signal2))},\ h), \\
& \qquad\qquad \texttt{'(((15) pc)}\ \texttt{((3) s)}\ \texttt{((1) b1)}\ \texttt{((2) b2)} \\
& \qquad\qquad\quad \texttt{(c0000000}_{16}\ \texttt{signal1)}\ \texttt{(c0000002}_{16}\ \texttt{signal2)} \\
& \qquad\qquad\quad \texttt{(c0000001}_{16}\ \texttt{l1)}\ \texttt{(c0000003}_{16}\ \texttt{l2)))}, \\
& \qquad h)) \\
\rightarrow & \text{satisfiesp}\,(h,\ \textsc{quiz-spec})
\end{array}
$$

Figure 7.21: A (nearly-final) quiz-show correctness result

in the final correctness theorem.

We have proved the final correctness lemma presented in Figure 7.16 using Nqthm. In the next section we describe some aspects of the proofs of these lemmas.

7.5 Invariants Proved in the Quiz-show Proof

In the previous section we presented an outline of the quiz-show correctness proof and described the three main sublemmas: the abstract system lemma, the FM9001 reasonableness lemma, and the program correctness lemma. In each of these proofs a crucial aspect is the construction of invariants and the proof that they are maintained during system execution. In this section we present the invariants for each of these lemmas.

These invariants were mostly generated by hand by the author as he proved these lemmas informally before constructing the Nqthm-checked proofs. However, perhaps 20of finding flaws in the informal argument when constructing the machine-checked proofs.

7.5.1 Abstract System Lemma Invariants

The invariants we prove as a part of the abstract system lemma proof are listed in Appendix E starting with 'quiz-invariant-counter-correspondence'. We list these invariants below where the value of each counter (c n) is represented in this list as *cn*, the name of the current state of the button1 specification, the button2 specification, and the program specification are represented with *p1*, *p2*, and *pp* respectively, and *s*, *l1*, *l2*, *b1*, and *b2* represent assignment values.

1. $c1 = c5$
2. $c2 = c6$
3. $c3 = c7$
4. $c4 = c8$
5. $(((pp = \texttt{'read-b2}) \vee (pp = \texttt{'update-state}))$
   $\wedge \quad ((c9 + 30000) < c2)$
   $\wedge \quad (p1 = \texttt{'up}))$
   $\rightarrow \quad (b1 = 0)$
6. $(((pp = \texttt{'read-b2}) \vee (pp = \texttt{'update-state}))$
   $\wedge \quad ((c9 + 30000) < c1)$
   $\wedge \quad (p1 = \texttt{'down}))$
   $\rightarrow \quad (b1 = 1)$
7. $((pp = \texttt{'update-state}) \wedge ((c9 + 30000) < c4) \wedge (p2 = \texttt{'up}))$
   $\rightarrow \quad (b2 = 0)$
8. $((pp = \texttt{'update-state}) \wedge ((c9 + 30000) < c3) \wedge (p2 = \texttt{'down}))$
   $\rightarrow \quad (b2 = 1)$
9. $(((pp = \texttt{'update-l1}) \vee (pp = \texttt{'update-l2}))$
   $\wedge \quad ((c9 + 30000) < c1)$
   $\wedge \quad (p1 = \texttt{'down}))$
   $\rightarrow \quad ((s = 1) \vee (s = 2))$

10. $(((pp = \texttt{'read-b1}) \lor (pp = \texttt{'read-b2}) \lor (pp = \texttt{'update-state}))$
$\land \quad ((30400 + c9) < c1)$
$\land \quad (p1 = \texttt{'down}))$
$\rightarrow \quad ((s = 1) \lor (s = 2))$

11. $(((pp = \texttt{'update-l1}) \lor (pp = \texttt{'update-l2}))$
$\land \quad ((c9 + 30000) < c2)$
$\land \quad (p1 = \texttt{'up}))$
$\rightarrow \quad ((s = 0) \lor (s = 2))$

12. $(((pp = \texttt{'read-b1}) \lor (pp = \texttt{'read-b2}) \lor (pp = \texttt{'update-state}))$
$\land \quad ((30400 + c9) < c2)$
$\land \quad (p1 = \texttt{'up}))$
$\rightarrow \quad ((s = 0) \lor (s = 2))$

13. $(((pp = \texttt{'update-l1}) \lor (pp = \texttt{'update-l2}))$
$\land \quad ((c9 + 30000) < c3)$
$\land \quad (p2 = \texttt{'down}))$
$\rightarrow \quad ((s = 1) \lor (s = 2))$

14. $(((pp = \texttt{'read-b1}) \lor (pp = \texttt{'read-b2}) \lor (pp = \texttt{'update-state}))$
$\land \quad ((30400 + c9) < c3)$
$\land \quad (p2 = \texttt{'down}))$
$\rightarrow \quad ((s = 1) \lor (s = 2))$

15. $(((pp = \texttt{'update-l1}) \lor (pp = \texttt{'update-l2}))$
$\land \quad ((c9 + 30000) < c4)$
$\land \quad (p2 = \texttt{'up}))$
$\rightarrow \quad ((s = 0) \lor (s = 1))$

16. $(((pp = \texttt{'read-b1}) \lor (pp = \texttt{'read-b2}) \lor (pp = \texttt{'update-state}))$
$\land \quad ((30400 + c9) < c4)$
$\land \quad (p2 = \texttt{'up}))$
$\rightarrow \quad ((s = 0) \lor (s = 1))$

17. $((pp = \texttt{'update-l2}) \land (p1 = \texttt{'up}) \land ((30000 + c9) < c2))$
$\rightarrow \quad (l1 = 1)$

18. $((p1 = \texttt{'up}) \land ((30400 + c9) < c2)) \rightarrow (l1 = 1)$

19. $((pp = \texttt{'update-l2})$
$\land \quad (p1 = \texttt{'down})$
$\land \quad (p2 = \texttt{'up})$
$\land \quad ((30000 + c9) < c1)$
$\land \quad ((30000 + c9) < c4))$
$\rightarrow \quad (l1 = 0)$

20. $((p1 = \texttt{'down})$
$\land \quad (p2 = \texttt{'up})$
$\land \quad ((30400 + c9) < c1)$
$\land \quad ((30400 + c9) < c4))$
$\rightarrow \quad (l1 = 0)$

21. $((pp = \texttt{'update-l2})$
$\wedge \ (p1 = \texttt{'down})$
$\wedge \ (p2 = \texttt{'down})$
$\wedge \ ((30000 + c9) < c1)$
$\wedge \ ((30000 + c9) < c3))$
$\rightarrow \ (l1 = \textbf{if } s = 1 \textbf{ then } 0$
$\quad \textbf{else } 1 \textbf{ endif})$

22. $((p1 = \texttt{'down})$
$\wedge \ (p2 = \texttt{'down})$
$\wedge \ ((30400 + c9) < c1)$
$\wedge \ ((30400 + c9) < c3))$
$\rightarrow \ (l1 = \textbf{if } s = 1 \textbf{ then } 0$
$\quad \textbf{else } 1 \textbf{ endif})$

23. $((p2 = \texttt{'up}) \wedge ((30400 + c9) < c4)) \rightarrow (l2 = 1)$

24. $((p2 = \texttt{'down})$
$\wedge \ (p1 = \texttt{'up})$
$\wedge \ ((30400 + c9) < c3)$
$\wedge \ ((30400 + c9) < c2))$
$\rightarrow \ (l2 = 0)$

25. $((p2 = \texttt{'down})$
$\wedge \ (p1 = \texttt{'down})$
$\wedge \ ((30400 + c9) < c3)$
$\wedge \ ((30400 + c9) < c1))$
$\rightarrow \ (l2 = \textbf{if } s = 2 \textbf{ then } 0$
$\quad \textbf{else } 1 \textbf{ endif})$

26. $((l1 = 0)$
$\wedge \ (p1 = \texttt{'down})$
$\wedge \ (pp = \texttt{'update-l2})$
$\wedge \ ((30000 + c9) < c1))$
$\rightarrow \ (s = 1)$

27. $((l1 = 0) \wedge (p1 = \texttt{'down}) \wedge ((30400 + c9) < c1)) \rightarrow (s = 1)$

28. $((l2 = 0) \wedge (p2 = \texttt{'down}) \wedge ((30400 + c9) < c3)) \rightarrow (s = 2)$

The proof that these invariants are in fact invariants is arduous but straightforward, so we focus on how these invariants are used to prove the theorem. As an example of the use of these invariants in the abstract system proof, suppose that history h has assignments h_i and h_{i+1} that are mapped to **up** and **going-down** in the behavior describing button1, **down** and **down** in the behavior describing button2, and **update-l2** and **read-b1** in the program specification. We show for this case that the invariants and the behaviors assumed satisfied by h are sufficient to show that the assignments satisfy the predicate of the transition between states to which they are mapped in the quiz-show specification behavior. We calculate that the assignments map to **two** and **both** in the quiz-show specification using 'abstract-quiz-map'. The transition predicate (as presented in Figures 7.14 and 7.15) using $c7$ to abbreviate counter (c 7) in the style of behavior diagrams is

$$(a1 = \texttt{'depress}) \wedge ((a2 = \texttt{'f}) \wedge ((\neg\ ((\texttt{40000} < c7) \wedge (\mathrm{old}\,(l2) = \texttt{0}))) \vee (l2 = \texttt{0})))$$

The value of $a1$ in assignment h_{i+1} is DEPRESS since these assignments are mapped to **up** and **going-down** in the button1 behavior, and the value of $a2$ is **f** since both assignments are mapped to **down** in the button2 behavior. Assume $(\texttt{40000} < c7) \wedge (\mathrm{old}\,(l2) = \texttt{0})$. From the program specification satisfaction hypothesis, (c 9) is reset on this transition and so has value 0. By invariants 3 and 28 and our assumption above, h_i associates the value 2 with s. By the program specification hypothesis, specifically the transition predicate on the transition from **update-l2** to **read-b1**, $l2$ in h_{i+1} has value 0, so the transition predicate is satisfied.

By this kind of reasoning for all possible system transitions, we prove that the abstract system theorem follows from the state mapping function and the proved invariants.

7.5.2 FM9001 Reasonableness Lemma Invariants

In order to prove the FM9001 reasonableness lemma we show that a set of invariants about the FM9001 running the quiz-show application is preserved throughout execution. These invariants are described in the Nqthm function ‘quiz-program-functioning-invariants’ which is a predicate that returns whether its argument is an FM9001 state that has the following properties:

1. Each of the 36 memory locations starting with address $\texttt{4000000b}_{16}$ contains the quiz-show program loop from Figure 7.4.

2. Both memory-mapped input locations are 32-bit RAM.

3. Each of the used registers is 32-bit RAM.

4. Each of the registers used to store memory-mapped I/O device addresses in the program contains the appropriate values.

5. The program counter register contains a value that points to an instruction in the quiz-show program loop.

The reasonableness lemma follows from the FM9001 definition and the proof that each of these invariants is preserved when an instruction is executed. This lemma is very useful because it allows reasoning about the operation of the FM9001 running the quiz-show application using the ‘quiz-program-operation’ function which is much simpler than the FM9001 model.

7.5.3 Program Correctness Lemma Invariants

In order to prove the program correctness lemma, we introduce in function ‘quiz-program-correctness-invariants’ a set of invariants that hold for each microcycle of program execution. We list these invariants below, where *pc* is the value of the program counter, *s* is the current selected button as stored in register 3, *b1* and *b2* are the values of the button signals as stored in registers 1 and 2, *temp* is the value of register 0, *z* and *c* are the values of the z and c flags, and *c9* is the value of (c 9) assuming we have followed the path in the program specification as indicated in Section 7.4.3.

1. $(s = 0) \vee (s = 1) \vee (s = 2)$
2. $(b1 = 0) \vee (b1 = 1)$
3. $(b2 = 0) \vee (b2 = 1)$
4. *pc* is a loop instruction location
5. $(pc = \mathtt{4000000b}_{16}) \rightarrow (c9 < 20)$
6. $(pc = \mathtt{4000000c}_{16}) \rightarrow (c9 < 36)$
7. $(pc = \mathtt{4000000d}_{16}) \rightarrow (c9 < 52)$
8. $(pc = \mathtt{4000000e}_{16}) \rightarrow ((c9 < 63) \wedge (z = (s = 2)))$
9. $(pc = \mathtt{40000010}_{16}) \rightarrow ((c9 < 79) \wedge (s \neq 2))$
10. $(pc = \mathtt{40000011}_{16}) \rightarrow ((c9 < 89) \wedge (s \neq 2) \wedge (c = (b1 = 1)))$
11. $(pc = \mathtt{40000012}_{16})$
    $\rightarrow \quad ((c9 < 99) \wedge (s \neq 2) \wedge (c = (b1 = 1)) \wedge (temp = 1))$
12. $(pc = \mathtt{40000014}_{16}) \rightarrow ((c9 < 115) \wedge (s \neq 2) \wedge (b1 = 0))$
13. $(pc = \mathtt{40000015}_{16})$
    $\rightarrow \quad ((c9 < 125) \wedge (s \neq 2) \wedge (b1 = 0) \wedge (c = (b2 = 1)))$
14. $(pc = \mathtt{40000016}_{16})$
    $\rightarrow \quad ((c9 < 135)$
    $\wedge \quad (s \neq 2)$
    $\wedge \quad (b1 = 0)$
    $\wedge \quad (c = (b2 = 1))$
    $\wedge \quad (temp = 2))$
15. $(pc = \mathtt{40000018}_{16})$
    $\rightarrow \quad ((c9 < 151) \wedge (s \neq 2) \wedge (b1 = 0) \wedge (b2 = 0))$
16. $(pc = \mathtt{40000019}_{16})$
    $\rightarrow \quad ((c9 < 161) \wedge (s \neq 2) \wedge (b1 = 0) \wedge (b2 = 0) \wedge (temp = 0))$
17. $(pc = \mathtt{4000001b}_{16}) \rightarrow ((c9 < 79) \wedge (s = 2))$
18. $(pc = \mathtt{4000001c}_{16}) \rightarrow ((c9 < 89) \wedge (s = 2) \wedge (c = (b2 = 1)))$
19. $(pc = \mathtt{4000001d}_{16})$
    $\rightarrow \quad ((c9 < 99) \wedge (s = 2) \wedge (c = (b2 = 1)) \wedge (temp = 2))$
20. $(pc = \mathtt{4000001f}_{16}) \rightarrow ((c9 < 115) \wedge (s = 2) \wedge (b2 = 0))$

21. $(pc = \mathtt{40000020}_{16})$
$\rightarrow \quad ((c9 < \mathtt{125}) \wedge (s = \mathtt{2}) \wedge (b2 = \mathtt{0}) \wedge (c = (b1 = \mathtt{1})))$

22. $(pc = \mathtt{40000021}_{16})$
$$\begin{array}{ll} \rightarrow & ((c9 < \mathtt{135}) \\ & \quad \wedge \quad (s = \mathtt{2}) \\ & \quad \wedge \quad (b2 = \mathtt{0}) \\ & \quad \wedge \quad (c = (b1 = \mathtt{1})) \\ & \quad \wedge \quad (temp = \mathtt{1})) \end{array}$$

23. $(pc = \mathtt{40000023}_{16})$
$\rightarrow \quad ((c9 < \mathtt{151}) \wedge (s = \mathtt{2}) \wedge (b2 = \mathtt{0}) \wedge (b1 = \mathtt{0}))$

24. $(pc = \mathtt{40000024}_{16})$
$$\begin{array}{ll} \rightarrow & ((c9 < \mathtt{177}) \\ & \quad \wedge \quad (temp = \textbf{if } (s = \mathtt{0}) \vee (s = \mathtt{1}) \\ & \qquad\qquad\qquad \textbf{then if } b1 = \mathtt{1} \textbf{ then } \mathtt{1} \\ & \qquad\qquad\qquad\qquad \textbf{elseif } b2 = \mathtt{1} \textbf{ then } \mathtt{2} \\ & \qquad\qquad\qquad\qquad \textbf{else } \mathtt{0} \textbf{ endif} \\ & \qquad\qquad\qquad \textbf{elseif } b2 = \mathtt{1} \textbf{ then } \mathtt{2} \\ & \qquad\qquad\qquad \textbf{elseif } b1 = \mathtt{1} \textbf{ then } \mathtt{1} \\ & \qquad\qquad\qquad \textbf{else } \mathtt{0} \textbf{ endif})) \end{array}$$

25. $(pc = \mathtt{40000025}_{16}) \rightarrow (c9 < \mathtt{187})$

26. $(pc = \mathtt{40000026}_{16}) \rightarrow ((c9 < \mathtt{197}) \wedge (temp = \mathtt{1}))$

27. $(pc = \mathtt{40000027}_{16})$
$\rightarrow \quad ((c9 < \mathtt{208}) \wedge (temp = \mathtt{1}) \wedge (z = (s = \mathtt{1})))$

28. $(pc = \mathtt{40000028}_{16})$
$$\begin{array}{ll} \rightarrow & ((c9 < \mathtt{218}) \\ & \quad \wedge \quad (temp = \textbf{if } s = \mathtt{1} \textbf{ then } \mathtt{0} \\ & \qquad\qquad\qquad \textbf{else } \mathtt{1} \textbf{ endif})) \end{array}$$

29. $(pc = \mathtt{40000029}_{16}) \rightarrow (c9 < \mathtt{234})$

30. $(pc = \mathtt{4000002a}_{16}) \rightarrow ((c9 < \mathtt{244}) \wedge (temp = \mathtt{1}))$

31. $(pc = \mathtt{4000002b}_{16})$
$\rightarrow \quad ((c9 < \mathtt{255}) \wedge (temp = \mathtt{1}) \wedge (z = (s = \mathtt{2})))$

32. $(pc = \mathtt{4000002c}_{16})$
$$\begin{array}{ll} \rightarrow & ((c9 < \mathtt{265}) \\ & \quad \wedge \quad (temp = \textbf{if } s = \mathtt{2} \textbf{ then } \mathtt{0} \\ & \qquad\qquad\qquad \textbf{else } \mathtt{1} \textbf{ endif})) \end{array}$$

33. $(pc = \mathtt{4000002d}_{16}) \rightarrow (c9 < \mathtt{4})$

Again, the proof that these invariants are in fact invariant is arduous but straightforward so we focus on how they are used to prove the program correctness lemma. The invariants are used to prove the program correctness theorem in Figure 7.20 given the mapping from assignments to the program specification. For example, suppose h is consistent with the execution of the quiz-show application for all relevant program variables. Consider whether the consecutive assignments h_i and h_{i+1} where the program counter variable has value $\mathtt{40000028}_{16}$ and $\mathtt{40000029}_{16}$ respectively satisfies the transition predicate of the program correctness lemma behavior. As can be seen from the program listed in Figure 7.3, these assignments represent the microcycles between which the effect of executing the assembled instruction `(move t f (r6) f0)` is seen. By definition of 'quiz-program-operation', *l1* is updated with the value of *temp*. These assignments are mapped to states **update-l1** and **update-l2** in the program specification respectively, so as can be seen in Figure 7.17 we must show that they satisfy the following which is the transition predicate of the transition between these two states:

$$
\begin{array}{l}
(\mathrm{old}\,(b1) = b1) \\
\wedge \quad ((\mathrm{old}\,(b2) = b2) \\
\qquad \wedge \quad ((s = \mathrm{old}\,(s)) \\
\qquad\qquad \wedge \quad ((l1 = \textbf{if } s = \mathtt{1} \textbf{ then } \mathtt{0} \\
\qquad\qquad\qquad\qquad \textbf{else } \mathtt{1} \textbf{ endif}) \\
\qquad\qquad\qquad \wedge \quad (\mathrm{old}\,(l2) = l2))))
\end{array}
$$

Invariant 28 and the effect of this instruction are sufficient to prove that this predicate is satisfied. We prove the quiz-show program correctness lemma by similar reasoning for each instruction.

In the next section we present another application for which we have proved a correctness lemma.

```
(satisfiesp (h, BUTTON-SPEC)
 ∧   variable-not-overflowed-in-historyp ('signal, h)
 ∧   extension-historyp (fm9001-rt-history (MIRROR-INITIAL-STATE,
                                            3,
                                            extract-from-history ('((3221225472
                                                                     signal)),
                                                                  h),
                                            '((3221225473 output))),
                         h))
→    satisfiesp (h, MIRROR-SYSTEM-SPEC)
```

Figure 7.22: The light-switch correctness theorem

7.6 The Light-Switch Example

We have constructed and verified another real-time system that uses the FM9001 board, a microprocessor-controlled light-switch. We prove that the system's light mirrors the status of a button promptly. The structure of the correctness lemma and the proof are similar to and simpler than that of the quiz-show example. We present three aspects of the light-switch example in this section: a brief description of the correctness lemma proved, a behavior that describes program execution that is used to factor the correctness proof, and an example execution of the system using the Nqthm execution facility.

7.6.1 A Correctness Lemma

The correctness lemma for the light-switch system is presented in Figure 7.22. The button model we use in this correctness lemma is presented in Figure 7.23. It is very similar to the model presented for the buttons in the quiz-show system, except that different variable names are used and it does not have the requirement that the signal be always 0 or 1. The correctness lemma also has a hypothesis that the signal from the button is representable in 32 bits. (We did not need this additional hypothesis in the quiz-show correctness theorem

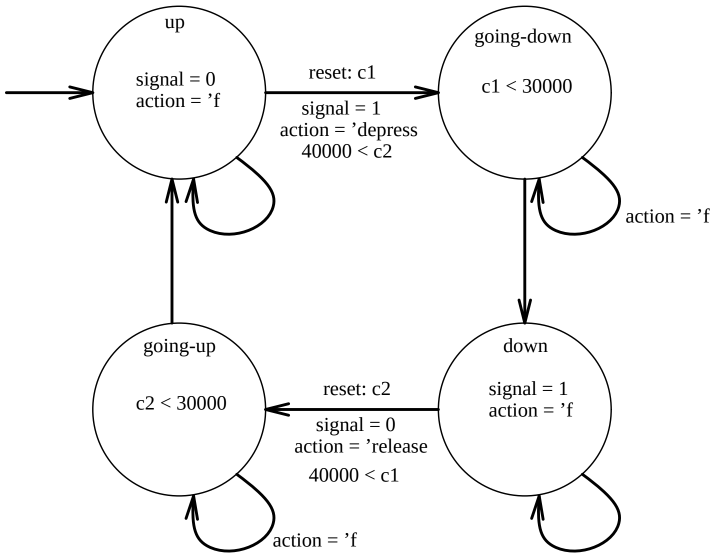

Figure 7.23: A button model

because in that example the button models have the additional requirement that the signal be 0 or 1.)

We assume that history h is an extension of an FM9001 execution, again in a manner analogous to the quiz-show system. In this case we only assume that there is one input location and one relevant output location. We provide a value for the lone input device for each microcycle, in accordance with our FM9001 model assumptions in Chapter 5. The interested reader is referred to Appendix E for a description of the initial state represented by Nqthm constant MIRROR-INITIAL-STATE. Among other things, it includes a program counter with value $\texttt{40000000}_{\texttt{16}}$ and a program loaded at that location that is the result of assembling the FM9001 assembly program presented in Figure 7.24.

The conclusion of the correctness theorem formalizes the notion that the light turns on and off soon after the button is depressed and released.

```
(move t f r1 (pc+))
#xC0000000
(move t f r0 (r1))
(move t f r1 (pc+))
#xC0000001
(move t f (r1) r0)
(move t f pc (pc+))
#x40000000
```

Figure 7.24: The light-switch application

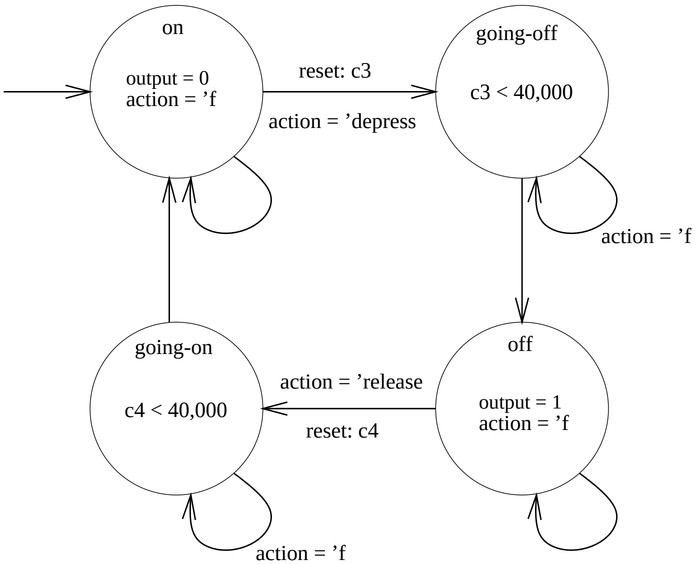

Figure 7.25: Light-switch desired property

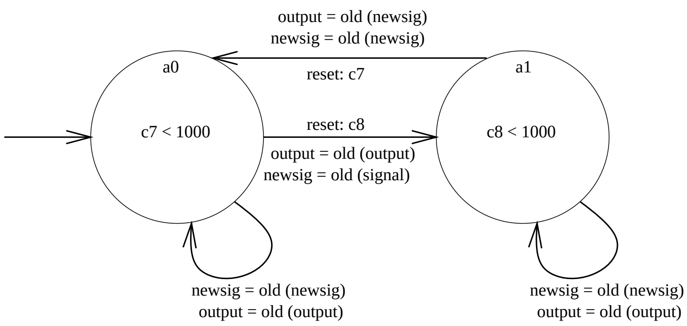

Figure 7.26: Light-switch program specification

Figure 7.25 presents a pictorial representation of this behavior. As we do with the quiz-show system specification, we claim that this behavior formalizes our informal notion of desirable system behavior. Except perhaps for brief periods just after the button is pressed or released, the processor-controlled light reflects the current status of the button. Any history that satisfies this conclusion represents an execution of the system that has the property we desire.

7.6.2 A Light-Switch Program Specification

As with the quiz-show proof, we use a program specification to factor the proof of the correctness lemma. Breaking the proof into smaller pieces in this manner makes the proof tractable. The program specification for the light-switch system is presented in Figure 7.26. This behavior allows us to prove an abstract lemma about the light-switch system, event 'mirror-satisfiesp'. We then prove that our application meets the program specification, which is proved in event 'mirror-program-meets-spec'.

The correctness theorem presented in Figure 7.22 has been proved using Nqthm using this approach. The correctness theorem is prove-lemma event 'light-switch-works'.

7.6.3 Example Execution of the Light-Switch System

The light-switch system is quite simple, so we take advantage of the executable nature of the Nqthm logic to examine the light-switch correctness theorem with respect to a history that represents one particular execution of the system. Since we have formalized all the relevant notions as definitions in the Nqthm logic we can evaluate the definitions on constants. Consider an example history that contains 50,000 microcycles of inactivity with the button released and the signal 0, then a button depress with an immediate steady signal 1 that lasts for 1,001 microcycles. Variable *output* has value 0 for the first 50,061 clock ticks then value 1 for the rest.

We use 'setq' in the Nqthm R-loop facility to associate the variable h with this history.

```
*(setq h
      (append
       (repeat 50000 '((action f) (signal 0) (output 0)))
      (append
       (repeat 1 '((action depress) (signal 1) (output 0)))
      (append
       (repeat 60 '((action f) (signal 1) (output 0)))
       (repeat 940 '((action f) (signal 1) (output 1)))))))
```

We check whether for this history the first hypothesis of our correctness result holds, which formalizes the assumption that the history satisfies the button model. (We use the function 'satisfiesp-eff' rather than 'satisfiesp' which, as can be seen in event 'satisfiesp-eff-equiv', have been proved equivalent. We use 'satisfiesp-eff' here because it executes far more efficiently.)

```
*(satisfiesp-eff h (button-spec))
 T
```

So, this example execution of the light-switch system satisfies our model of how the button operates.

The second hypothesis in our correctness result holds if each assignment in history *h* associates with *signal* a number that is less than 2^{32}. We evaluate this hypothesis for our example history to show that the example history satisfies this requirement.

```
*(variable-not-overflowed-in-historyp 'signal h)
 T
```

So, all values of *signal* in the history *h* defined above have values representable in 32 bits.

We also check that our example history satisfies the third hypothesis, which formalizes the assumption that the history is consistent with the execution of the FM9001 when executing the light-switch application. (The Nqthm function 'fm9001-rt-history-repeat-helper' is proved equivalent to 'fm9001-rt-history' in event 'fm9001-rt-repeat-helper-equiv', and we use it here because it is more efficient to execute.)

```
*(extension-historyp
         (fm9001-rt-history-repeat-helper (mirror-initial-state) 3
            (extract-from-history '((3221225472 signal)) h)
            '((3221225473 output)))
         h)
 T
```

Since the light-switch correctness theorem has been proved and our example history satisfies each of the hypothesis of the lemma, we know that the example satisfies the conclusion of the lemma as well. Even so, we determine this directly by evaluating the conclusion on our example history.

```
*(satisfiesp-eff h (mirror-system-spec))
 T
```

Chapter 8

Some Implications of the Proved Real-time System

Chapter 7 describes a formalization and associated machine-checked proof of the correctness of a real-time system. This chapter highlights some of the implications of that work.

8.1 Execution on the FM9001 Single-board Computer

Perhaps the most obvious and most satisfying implication of the theorems presented in Chapter 7 is that the applications described in that chapter may be dependably executed on the FM9001 board described in Section 5. (Recall the discussion in Section 5.9 about the relationship between the real-time FM9001 model and the fabricated FM9001.) The monitor program resident in the FM9001 ROM and a PERL script [50] running on a Sun Sparcstation are used to communicate with the FM9001 board [55]. To run the quiz-show program, one gives the following commands to the PERL program:

1. "!", which causes the FM9001 board to be reset and the monitor program to be executed;

2. "s 40000000" followed by the values of the program, which signals the monitor to load the program at location $\texttt{40000000}_{16}$. In the case of the quiz-show program the values of the assembled program are listed in Figure 7.4;

task name	task description	maximum duration
read-b1	read a value from button1	50 microcycles
read-b2	read a value from button2	50 microcycles
update-state	update current state	200 microcycles
update-l1	update light1	50 microcycles
update-l2	update light2	50 microcycles

Table 8.1: Quiz-show application partitioned into tasks

3. "x 40000000", which causes the program counter to be set to the location $\texttt{40000000}_{16}$, thereby ending the execution of the monitor program and beginning execution of the loaded program on the FM9001 board.

After all our efforts to formalize and verify the quiz-show program, we are pleased to report that the system appears to work as desired.

8.2 Comparison with Scheduling Theorem

We have proved several theorems in this dissertation about real-time systems. In Chapter 3 we prove a classic theorem about the EDF scheduling policy. In Chapter 7 we prove that an application succeeds in preserving certain desirable properties in a real-time system. These theorems were proved independently of each other, but both represent an approach to real-time system verification. In this section we discuss the relationship of these theorems.

Suppose we had constructed the quiz-show application within the framework of a scheduled real-time application. That is, suppose we had broken the program into tasks and specified the task request deadlines. The program specification for the quiz-show application presented in Figure 7.17 suggests a sensible partitioning of the application into five constituent tasks, and Table 8.1 presents these tasks. We could prove limits on the execution time of the tasks, much as we did in the correctness theorem proof we accomplished. For example, some of the proof obligations of the program correctness lemma of Section 7.4.3 are similar to the limits on execution time of the tasks of the application that are listed as durations in Table 8.1.

By application of the EDF optimality theorem from Chapter 3, we can show that each of the scheduled tasks meets its deadline assuming some sensible period for requests of needed tasks. So, for example, we can require that each task is executed every millisecond, or 10,000 microcycles, and show that the scheduled application does not fall behind. Note that the sum of the quotients of the duration and period for each task is less than 1, as is needed to apply the EDF optimality theorem.

How does the result of applying the scheduler policy theorem compare with the correctness result we actually proved? Using the scheduling theorem leads to a different notion of correctness than what we proved. The scheduling theorem focuses on task timing, so we can use it to conclude that the application never falls behind schedule. However, we proved something stronger in the correctness theorem of the last chapter, namely that the application does the right thing at the right time and thereby interacts successfully with the rest of a real-time system. The use of a real-time scheduler and the framework of scheduling theory requires we abstract somewhat from the actual bits to be run on the processor and prevents direct reasoning about what the application accomplishes. The use of a scheduler makes formalization and proof of application behavior other than timing behavior more difficult from the standpoint of formal verification.

Did we waste our time in Chapter 3 proving a theorem about schedulers if that kind of theorem can't be used to do the kind of verification we did for the quiz-show problem? Absolutely not. Schedulers are an extremely valuable real-time development tool whose correctness is vitally important to the many applications that use them. We were able to avoid the use of a scheduler in this example by performing the scheduling ourselves, thereby preserving the link with the underlying machine that allows us to reason directly about application execution. The quiz-show application presented in Figure 7.3 is a scheduled version of the tasks in Table 8.1 above.

8.3 Automating Abstract Proofs

Timed automata are attractive as a specification method because of their intuitive, graphical nature and their connection with computation. Behaviors as presented in Chapter 6 are similar to other approaches for specifying real-time systems. Much work has also been done to automate correctness proofs about automata. Alur and Dill present timed automata and a decision procedure for deciding interesting properties [3]. Subsequent algorithms have made these kinds of automatic inferences less impractical because of inefficiency. Although behaviors are syntactically somewhat different from Alur and Dill timed automata, they are similar in spirit and it appears likely that much of this work could be applied to that portion of our proofs we call "abstract" proofs in Section 7.4.

In this section we describe a prototype theorem proving system we have implemented that assists users in deriving behaviors that are known to be satisfied by histories that satisfy other, assumed system behaviors. This prover was not itself verified to be correct, and it was not used to derive any of the proofs reported in the previous chapter, but we briefly describe its design because it represents possible future work.

8.3.1 A Prototype Behaviors Prover

We have built a prototype prover for behaviors that implements several transformations on behaviors. Each of these transformations is believed to have certain useful properties with respect to the histories that satisfy the resulting behavior.

'remove-counter' The term remove-counter $(b,\ c)$ returns a behavior identical to b except that each state predicate list and transition predicate list has had removed any predicate in which counter c occurs, and each transition's list of resets has had removed each occurrence of counter c. History h satisfies remove-counter $(b,\ c)$ if h satisfies b.

'simplify' The term simplify (b) returns a behavior that is identical to b except that possibly some transitions that can not be satisfied by any pair of consecutive assignments in a history accepted by b are not included, and any state to which there is no sequence of transitions from the initial state is not included. The implementation of 'simplify' uses a search to eliminate irrelevant transitions and states of a behavior. A simple linear arithmetic package is implemented that detects some of the unsatisfiable transitions. History h satisfies simplify (b) if h satisfies b.

'counter-left' The term counter-left $(b,\ var,\ exp)$ is a behavior identical to b except that occurrences of var on the left side of the inequality in some state and transition predicates have been replaced by exp in states where $exp \leq var$ can be established. History h satisfies counter-left $(b,\ var,\ exp)$ if h satisfies b.

'counter-right' The term counter-right (b, var, exp) returns a behavior identical to b except that occurrences of var on the right side of the inequality in some state and transition predicates have been replaced by exp in states where $var \leq exp$ can be established. History h satisfies counter-right (b, var, exp) if h satisfies b.

'product' The term product ($b1$, $b2$) is a behavior with the following properties:

- The initial state is ($i1$ $i2$) where $i1$ is the initial state of $b1$ and $i2$ is the initial state of $b2$.
- The state-list consists of states whose name is of the form ($s1$ $s2$) where $s1$ is a state of $b1$ and $s2$ is a state of $b2$. Each state predicate is the conjunction of the predicates of the component states.
- The transition list of each state ($s1$ $s2$) contains a transition to the state ($s3$ $s4$) for each transition to $s3$ in $s1$ in $b1$ and $s4$ in $s2$ in $b2$. The set of resets for this transition is the set-union of the resets of the component transitions. The predicate of this transition is the conjunction of the component predlists.

Figure 8.1 presents two behaviors and their product. If h satisfies $b1$, h satisfies $b2$, and no counter occurs in a reset list or predicate in both $b1$ and $b2$, history h satisfies product ($b1$, $b2$).

'abstract' Let map be an assignment of states in the state-list of $b1$ to states in the state-list of $b2$. A behavior $b1$ is abstracted by $b2$ using map if the following hold:

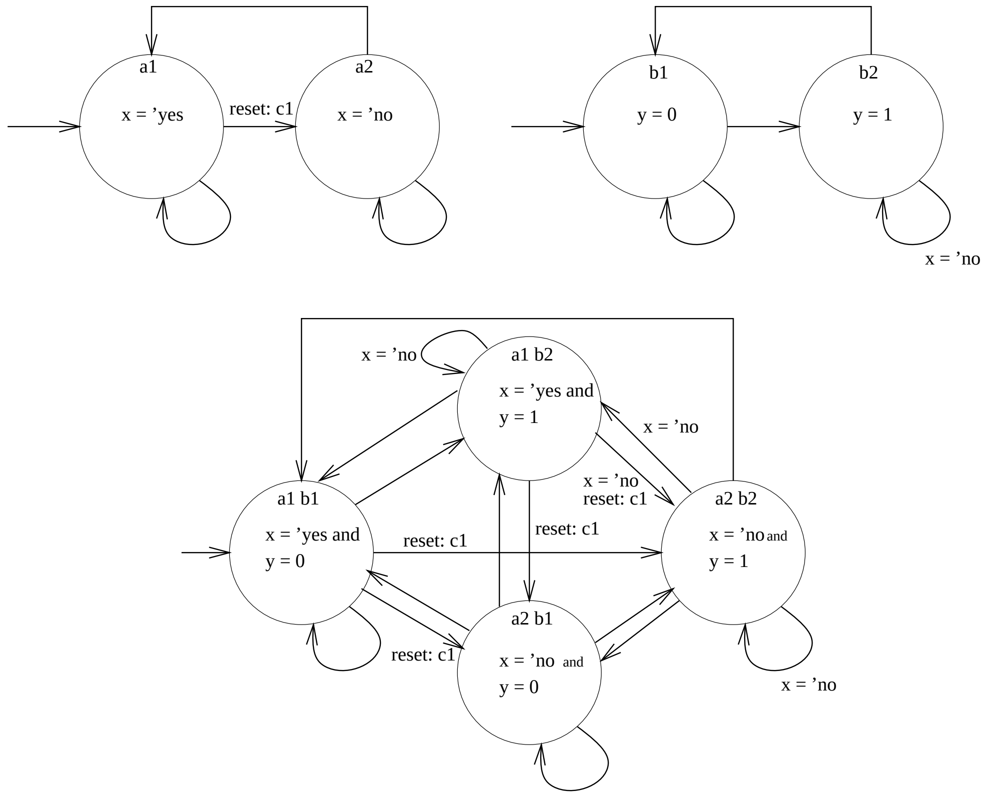

Figure 8.1: Two behaviors and their product

- map(initial state of *b1*) = initial state of *b2*.
- For each state *s1* in the state-list of *b1*, then the predicate of *s1* implies the predicate of map (*s1*).
- For each transition of each state *s1* in the state-list of *b1* there exists a transition in map (*s1*) where
  - the to-state is the name of map (*s2*), and
  - the resets is equal to the resets of the transition in *b1*, and
  - the predicate of this transition is implied by the predicate of the transition in *b1*.

The term abstract (*b1*, *b2*, *map*) is *b2* if *b1* is abstracted by *b2* using *map*, and otherwise *b1*. History h satisfies abstract (*b1*, *b2*, *map*) if h satisfies *b1*.

8.3.2 Overview

By using these transformations and associated theorems, one can derive the satisfiability of a behavior directly. We accomplish this by repeated transformation of the assumed behaviors until they are identical to the desired behavior. We have derived some small examples in this manner using the implemented prototype prover.

We do not use the prototype prover in our proofs because the verification of the transformation properties is a large task and these kinds of proofs are a relatively small part of our verification problem. As seen in Chapter 7 we were able to accomplish some verifications without these tools. Nevertheless, for larger problems, this approach may be useful.

8.4 Overview of Our Approach

8.4.1 Early History

The early history of this project was chaotic as it was a struggle to find a sensible way to represent real-time system properties and to understand what was needed in the statement of their correctness. We mention a few approaches that we pursued but abandoned.

We originally represented real-time properties directly as Nqthm definitions that were predicates on histories. These functions were very unsatisfying, however, as they were terribly complex and it was very difficult to explain why they represented anything interesting. This experience led us to invent behaviors. Although the invention of behaviors was perhaps unconsciously influenced by the work of other researchers in real-time verification, they were primarily a result of the experience of trying to explain how Nqthm functions represented interesting real-time system properties. Only later did it become clear how similar our specification approach is to others.

Another early approach was to add to the functionality of the Piton language in order to support the development of real-time applications, and to build our real-time applications in that language rather than directly as FM9001 machine code. This essentially meant adding timing information to the Piton interpreter, some I/O instructions to the Piton instruction set, and a new element to the Piton state having to do with memory-mapped I/O elements. Although what is needed in computer languages to support real-time programs is an interesting topic, the added complexity of working with "real-time Piton" did not help us do real-time verification of the kind we wanted. In fact, we would need a real-time FM9001 model similar to what we eventually used in order to support a verified version of "real-time Piton", so we were making our work more difficult rather than less by working with Piton rather than directly on the FM9001.

We developed a very simple real-time programming language named Tiny and wrote a Tiny compiler for the FM9001. We developed some programs using it that did real-time work, but because the compiler was unverified the approach was unsatisfying and we began using FM9001 machine code.

Another early effort involved building a specialized theorem proving program in the Nqthm logic that could be used to reason about behaviors. Using the Nqthm logic as a programming language, we built a system in which the user specified the outline of a proof about these abstract specifications and the system checked it. We present this prototype system briefly in Section 8.3. Since these tools are rather complex their verification would be a large effort, we chose not to pursue it because the approach was interfering rather than assisting our efforts to accomplish real-time verification.

8.4.2 Examples Specification and Proof History

Once we had settled on an approach for specification of real-time systems we specified and proved the examples described in Chapter 7.

The light-switch proof described in Section 7.6 required 4 months for the author to construct. Much of that time was required to prove theorems about the underlying theory of the real-time FM9001 model and the Nqthm formalization of behavior satisfaction. Different versions of the real-time model and different versions of satisfaction are proved equivalent, and many theorems are proved about histories. The form of the lemmas that lead Nqthm to the needed proofs changed many times as the proof developed. The needed invariants for the light-switch example, remarkably complicated for such a small system, required much work and it was not initially clear how to structure them so that they were both provable and applicable in the proofs.

The quiz-show application described in Chapter 7 required about 2 weeks of specification work, 1 day of application implementation, and 3 days of work on the FM9001 board. The quiz-show proof that is outlined in Section 7.4 required about 5 1/2 weeks to construct. One week was spent designing a proof outline on paper. Each of the three major lemmas described in Section 7.4 took nearly the same amount of time: about 1 1/2 weeks. One day was spent deriving the final theorem from these lemmas.

The proof and use of invariants in the quiz-show proof as indicated in Section 7.5 is at the heart of the proof of these lemmas, and the source of most of the mistakes uncovered during the proof. Each of the lemmas had numerous mistakes in the invariants constructed for its proof, most frequently omitted properties that we did not realize would be required. Several mistakes were uncovered in the quiz-show system specification. Several typographic mistakes involving counter names were fixed during the proof. Also, it was discovered during the proof that the Nqthm function 'or' used in a predicate in the quiz-show specification was not implemented in the definition of behavior satisfaction, so the specification was in error. (This specification mistake was fixed by changing the definition of behavior satisfaction to include 'or.') No bugs were found in either application as a result of doing the proofs.[1]

The quiz-show system is far more complex than the light-switch system yet it took about one-third the time to construct. There are two main

[1] A bug in the quiz-show program that was undetected by testing apparently as a result of a version-control mistake was fixed in the very early stages of its proof, but since it would have been immediately apparent when the application was executed we do not consider it a bug found by the proof process.

reasons for this: the development of a reusable theory in Nqthm for these kinds of proofs and a better understanding of the problem that led to a more efficient proof. We believe that the speedup in the development of these proofs seen in these two examples results from these two factors in approximately equal measure.

8.4.3 Effort to Accomplish Future Proofs

It is of course impossible to quantify precisely how complex a real-time system is to verify, but given the numbers of invariants and other kinds of proof obligations the quiz-show system correctness proof seems about five times more complex than the light-switch correctness proof. Since the more complex proof required about one-third the time required for the simpler proof, one might say roughly that we sped the proof of these kinds of theorems by a factor of 15. Might we continue to see similar improvement in future examples using these methods?

We expect that there would continue to be improvement, but not nearly so dramatic. Much of the time spent in the development of the quiz-show proof involved the substance of the quiz-show system rather than figuring out how to structure the proof. We guess that with more experience and better-developed tools we might triple our speed at these kinds of proofs. As Section 8.3 indicates there is potential to automate that portion of the proof we call the abstract lemma. But for the most part at the present time there seems to be no alternative for the developer to design the outline of the proof himself, so there is a limit to how fast these proofs can be constructed.

8.4.4 The Difficult Process of Specification

The specifications we develop in the examples in Chapter 7 guarantee the properties we desire. Although we have justified their reasonableness informally, and of course reasoned about their application to several systems, we have not described how we generated them. It is difficult to specify formally the operation of systems. The quiz-show specification in Figure 7.14 required much effort to design, for example. The first versions of this specification formalized the notion of debounced signal and related the light values to that signal. This specification was far more complex than the ultimate specification, but this was not immediately clear and as a result several weeks of work were wasted on it.

We believe that the graphical nature of behaviors facilitates the creation of good specifications.

8.5 Generality of Our Approach

We believe that our approach is applicable to problems besides the two real-time system examples we have verified so far. The real-time FM9001 model and behaviors, and the general form of the correctness theorems we state about real-time systems, is not tied to the particular examples we have pursued. We wish to highlight another aspect of the generality of the approach we have taken. The approach we have taken to the proofs of these examples is applicable to other problems besides the two we have accomplished.

Recall the outline presented in Section 7.4 of the proof of the quiz-show system. The outline of the proof does not involve the examples we have proved, and seems generally applicable to correctness proofs of these kinds of systems. Also, besides the development of a proof structure for approaching these kinds of theorems, we have proved many lemmas that are useful for proving theorems of this kind. A partial explanation for the speedup of the proof development between the two examples is the successful automation of large parts of these proofs. Lemmas about the FM9001 model, for example, would be useful for other proofs of systems that employ an FM9001.

Chapter 9

Related Work

We have described various projects related to the work reported in this dissertation in previous chapters. However, some projects relate to more than one aspect of our work, or are related to our goals in a general way, and have not been described. In this section we briefly present several such projects.

9.1 A Verified Round-robin Scheduler Implementation

Researchers at the Australian Software Verification Research Centre have developed a machine-checked proof of the correctness of a round-robin scheduler [17]. The theorem they prove is that each task of a task set is assigned a processor time slice during each large time period. Their implementation is coded on a MIPS R3000 processor for which they have a processor model [48] and they are developing a general-purpose theorem-proving system, Ergo, with which they check their proofs [49].

There are many differences in our work, besides obvious ones such as the different theorem proving system and different target machine. Perhaps most importantly, they are not focussed on proving the correctness of particular real-time systems as we are, for example, with the verification of the quiz-show system. Instead, they prove the correctness of system software that might be used in an application composed of several tasks. The round-robin scheduler correctness theorem they prove constrains program traces of applications that use the verified scheduler.

Another difference is that the MIPS R3000 processor model has different kinds of complexity than the FM9001 model we use in our real-time system proofs. Since we are focussed on the interaction of our processor with the outside system, our model is a detailed, microcycle-by-microcycle description of that interaction which the MIPS R3000 lacks. However, the MIPS R3000 processor model includes features lacked by the real-time FM9001 model, notably interrupts and a pipelined design that complicates instruction timing.

The scheduler theorem proved about the MIPS R3000 code is different from the kind of theorem we prove in Chapter 3 about EDF schedulers. A round-robin scheduler is much simpler, and their correctness theorem represents a simpler property about their scheduler than we prove about EDF schedulers. Of course, their computation model is more realistic than the model we use in our scheduler proofs since it is related to the MIPS processor while ours is very abstract.

Despite the differences between the round-robin scheduler verification work and that reported in this dissertation, the use of a model of a real processor and the use of a mechanical theorem prover are striking similarities of the projects. The Australian researchers deal with complex, nitty-gritty issues like those we are forced to confront. They add an axiom to their system, for example, that represents the assumption that tasks do not overwrite the scheduler code. Also, they rely on their theorem proving system to help ensure that the complexity of their proofs does not lead them into making an error. We know of no other project so similar to our work.

9.2 Program Verification

There have been several projects that have used mechanical theorem provers to do code proofs in a manner similar to the code proofs done in this dissertation. Programs were proved correct using an interpreter-based model of a commercial processor [13]. The interpreter in this project models the behavior of the user model of the Motorola 68020 microprocessor. KIT is a small operating system proved correct for a typical Von Neumann architecture [5]. Some of the proof obligations involve showing that code correctly implements aspects of the operating system. A compiler for a subset of Gypsy [1, 44] that has as a target Piton [38] is proved correct [59]. Again, code that implements the needed functionality is proved correct.

Like the nim proof in Chapter 4, timing behavior information is not dealt with in these projects as the models of computation associate with the execution of each instruction one "clock tick". One project that differentiates between timings of instructions is pursuing machine-checking of correctness [22]. Hoare logic annotations are added to a simple imperative programming language by the programmer and verified using an automatic theorem proving system. These properties include timing properties of the program. The axioms about the timing properties of individual language statements are justified by demonstrating a compiler for a small, lower-level language for which it is expected that a verified processor will be built. The system is based on the HOL theorem prover [23] and does not use an interpreter-based language semantics. Also, its focus is program execution properties and not on real-time systems. Even so, this work is similar to the aspect of our work that involves machine-checked code correctness proofs.

9.3 Real-time Application Development Tools

TAL (Timing Analysis Language) is a special-purpose language designed to help programmers derive timing information from their programs [15]. In addition to conventional language features like assignment and alternation, TAL has primitives that allow the programmer to describe a program's structure and perform calculations involving the timing properties of that program. A real-time programmer can use TAL to describe his application and derive timing information that may be helpful in ensuring that the program meets its timing constraints. TAL suggests a methodology for building more-reliable real-time programs. Each block of a real-time program has an associated TAL program that calculates an execution time upper bound. This technique makes the development of real-time programs somewhat less fragile: a change in one program block requires a change in the corresponding TAL program but not in the whole system, so timing upper bounds can be recalculated without great effort. TAL facilitates the development of sophisticated real-time development tools since it provides a language in which program timing information can be expressed.

Some work has been accomplished to support building software with this methodology. A prototype system for adding annotations to assembly language programs that allows the calculation of tighter timing bounds has been implemented [37]. Nevertheless, programs developed in this way using TAL will not achieve the reliability of a formally specified and mechanically checked application. For simple cases, the development of the corresponding TAL program is straightforward from an analysis of the syntactic structure of the target program. But, as is pointed out in [15], for realistic examples semantic information must be used to achieve tighter, more-reasonable execution time upper bounds. The process of developing non-obvious TAL programs to derive these bounds could introduce errors into the development process.

An improvement to the TAL approach would be the use of a language for timing information that had a precisely-defined semantics and was more general. This would allow proofs to be constructed about the timing behavior of programs and allow these proofs to be connected to a model of the hardware. The use of semantic information about the application could be applied in a sound manner rather than in an ad hoc fashion during development of timing routines.

The Mars design environment system [40] allows the user to do timing calculations similar to that of TAL. That is, the system can calculate upper bounds of program execution time, mostly automatically from the syntax. The user can add timing assertions — presumably that involve some higher-level understanding of the program — in order to make the system produce tighter execution time bounds. Like TAL, however, there is no formal semantics underlying the system, so there is no way to guarantee the validity of the user-provided assertions. Also, because of the lack of a precise language semantics it is not possible to connect the program formally to any higher-level program specification. Thus, though this methodology appears to be an improvement on current practice of coding and testing, it has drawbacks as a tool for producing verified real-time software.

Other real-time development tools use timed automata similar to behaviors to describe real-time system properties [26, 29]. We hope eventually to apply our work to a commercially-available system that supports this kind of development in order to make our formal verification approach more widely accessible.

9.4 Real-time Verification Research

Two particularly useful overviews of aspects of real-time verification research have been written: a "survey and taxonomy" of critical systems [42] and a description of the application of different kinds of schedulers [45].

One of the difficulties in producing reliable real-time software is that many levels of a system must be validated. A misunderstanding of the behavior of the hardware platform could cause a malfunction just as could a mistake in the application program itself. Reliability in an application is enhanced if the connection from level to level is explicit and exact. Halang and Stoyenko [25] focus on tools available to the real-time application developer at different levels of the system. They evaluate hardware and languages in terms of usefulness in constructing predictable real-time systems. They identify a large number of desirable features and introduce techniques to calculate good upper bounds on task execution times in the presence of resource contention.

Although approaches to produce higher-quality real-time systems at different levels are addressed by Halang and Stoyenko, there are several deficiencies from the standpoint of developing verified real-time applications. The languages and hardware they envision for real-time application development are very complicated. The complexity of these tools makes the development of good, precise models difficult, which makes problematic both formally tying different levels together and reasoning about the behavior of systems.

We describe our requirements informally and present a behavior that represents our formal specification. Another approach would be to describe our requirements using a formal language and demonstrate that they hold on our system models. A language similar to CTL, for example, could capture these requirements formally and relate them to a timed automata like behaviors. We have not pursued this kind of formalization because we are satisfied with the clarity of the specifications for the examples we have verified, but future, more complex examples may drive us to a different formalism for our specifications.

A purported Nqthm-checked correctness proof of a real-time kernel appears to have great relevance to our work [46] because its title suggests that an implementation of a real-time kernel had been reasoned about. Nevertheless, this project does not address the issue of real-time kernel implementation. Instead, a real-time kernel formalized in the Nqthm logic is presented. A step function models a transition in the system, and a proof is constructed that a predicate representing the notion of good state is preserved.

Unfortunately, the model of the "small real-time kernel" is extremely simple and corresponds only in the most shallow way to the notion of a real-time kernel. The characterization of kernel "correctness" is so trivial that it does not require any behavior with which in practice a real-time kernel builder or user is concerned [46]. The "correctness" theorem is proved automatically by Nqthm directly from the definitions introduced to describe the kernel. We believe that this work does not advance the cause of reliable real-time kernel development or use and, in particular, that the work presented in this dissertation is unrelated to the work presented in [46].

RTL (real-time logic) is a logic for reasoning about the requirements of real-time applications [28]. It uses an event-action model of real-time execution. Events in the environment and beginning and end of the execution of tasks are described in the logic along with various constraints on their relationships. The logic includes inference rules that allow the proof of interesting system properties. Some work has been done toward proving RTL properties automatically by using a modification of the Sup-Inf procedure [8, 28].

RTL has many interesting and useful features. The logic allows reasoning about an application and the proof of some timing correctness properties. Events in the environment are modeled simply and assumptions about when they occur are used in the proof. The specifications of the timing properties are simple and easy to understand, unlike some competing systems. RTL has also been used as a part of a larger design system that assists a developer in the construction of a real-time application [36]. The use of a special-purpose logic such as RTL has disadvantages in the verification of real-time systems that include properties unrelated to timing. Other theorems in which we are interested such as the correctness of the underlying support system cannot be formalized in RTL, and therefore can't be connected formally to it.

Modechart is a state-based specification technique for real-time systems whose semantics is defined using RTL [29]. Tools that facilitate use of Modechart for real-time specification that exploit the ability to view these kinds of specifications graphically have been built [41]. Modechart is similar to Statechart [26], although Modechart allows convenient specification of absolute timing constraints.

Behaviors, the specification formalism introduced in Chapter 6, are simpler than Modechart specifications. The hierarchical nature of Modechart specifications which allows simpler specification of complex systems is not part of behaviors, and the Modechart notions of events, modes, and actions are represented in behaviors simply as values on state variables. Even so, the two formalisms are similar. As an example of this similarity, Figure 9.1 presents a Modechart specification of the light-switch correctness property presented as a behavior in Figure 7.25. The notation ΩDEPRESS on the transition represents the requirement that the event DEPRESS occurs. Action A includes the action

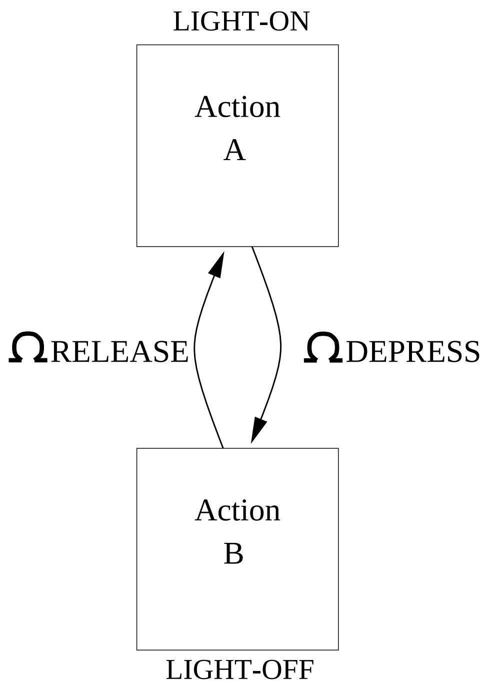

Figure 9.1: Modechart version of light-switch desired property

of turning the light on and action B includes the action of turning the light off. Constraints that require that the actions have a deadline of 4 milliseconds after the corresponding mode is entered by occurrence of an event are conveniently expressed as RTL formulae.

We have developed behaviors to express conveniently our real-time system properties and link us to a formal model of a processor. The essential difference between behaviors and Modechart specifications is the semantics of transitions. Unlike with Modechart specifications, a transition in a behavior represents a clock-tick. But at least for simple properties like those presented in Chapter 7 and the example in Figure 9.1, the two approaches allow the expression of similar specifications.

A model-checker that facilitates the proof of properties expressed as temporal formulae about systems described with Statechart specifications has been developed [16]. Simple properties of specified systems have been verified automatically with this system.

Some real-time system correctness theorems have been proved using a mechanical theorem prover. A simple model of a rowboat is constructed in which it travels down a river and the boat is acted upon by wind and force exerted by an actuator under program control [12]. Both the wind and actuator forces are integral units of unspecified size. There is a sensor that detects the position of the boat with respect to the centerline of the river, and the controller adjusts the actuator such that the boat stays close to this centerline. The control algorithm adjusts the actuator location based upon the two most recent sensor readings. Using a simple model of the boat and wind and assuming instantaneous updates of the actuator using the algorithm, several nice theorems are proved that suggest that this algorithm is effective in controlling the boat. The requirements on an implementation of the rowboat controller are not obviously derivable from the algorithm and model presented in [12]. For example, to conform to the simple model the program needs to take no time to execute.

Young presents an Nqthm-based specification of a simple real-time system [60]. Similar in style to the rowboat, it captures the requirements of a railroad crossing. The behavior of the train, a railway gate, and a controller are modeled using a sequence of states. The realities imposed by the physical system are used to constrain the behaviors, the needed behavior of the gate is specified, and a trivial algorithm is shown to meet all the specifications.

The Nqthm formalizations of the rowboat and railroad crossing are very interesting work as they succeed in formalizing classic real-time verification example problems. However, they both have a limitation from the standpoint of real-time system verification since no clear constraints on the execution properties of the real-time applications implicit in the examples are described. The approach used in these projects seems to have limited applicability to more complex systems where the interaction of an application with its environment is of vital concern.

Chapter 10

Conclusion

We put our trust in real-time systems with computer-controlled elements, and if they do not work correctly we risk catastrophe. The difficulties inherent in building real-time systems make ensuring their correctness challenging. We have formalized and proved several aspects of real-time systems in this dissertation, and show how we can overcome some of these difficulties. This type of approach holds promise for the development of systems that work reliably in the face of real-time system complexity.

Appendix A

Modifications to the FM9001 Board

The FM9001 board described in [2] and [55] has been modified to support real-time applications. Two new PLD's were designed, one a modification of memory address-handling hardware and one that drives the buttons and lights, and the board was rewired somewhat. This appendix contains the modified PLD design and lists the wiring changes. The PLD designs are in the form of PALASM input [43]. PALASM is a program that interprets designs and configures a programmable logic array that implements the design [4].

A.1 Design for FM_addr

The PLD FM_addr is designed to decode the FM9001 address lines and select appropriate inputs and delay lines was revised. Revision B simplified the old design and added the ability to select an I/O device.

```
Title    FM_addr
Pattern  fm_addr.pds
Revision B
Author   Ken Albin and Matt Wilding
Company  Computational Logic Inc.
Date     6/23/92 (mod 5/94 to simplify and add I/O devices)

;This pld is responsible for decoding the FM9001 address lines and using the
;results for chip selects and selecting the appropriate delay line tap
;for wait states.

;The strobe acknowledge signal is sychronized for the processor.

;Two inputs are the two most significant address lines.
;The third input is strobe from the FM.
;The fourth input is a ready line from the UART.
;Three inputs are delay taps from the delay line.
```

```
;Three outputs are chip selects.
;One output is the strobe acknowledge signal back to the FM.

CHIP FM_addr PAL22V10

CLK STROBE A30 A31 I4 IO_DELAY UART_DELAY RAM_DELAY ROM_DELAY RW I10 GND
I11 UART_READ UART_WRITE RAM_SELECT ROM_SELECT IO_SELECT DELAY_IN DTACK
WE IO8 IO9 VCC


STRING BOGUS_SELECT '(/A31 * /A30 * /RW * /STROBE)'


EQUATIONS

; Select the correct I/O device

/ROM_SELECT = /A31 * /A30 * RW * /STROBE    ;make sure not write to eprom

/RAM_SELECT = /A31 * A30 * /STROBE

UART_READ = A31 * /A30 * RW * /STROBE

UART_WRITE = A31 * /A30 * /RW * /STROBE

IO_SELECT = A31 * A30 * /STROBE


DELAY_IN = /STROBE

WE = RW           ; generate correct polarity for RAMs


; We use the delay PLD when we set the /DTACK signal back to the FM9001
; processor.  Note that the 9001 disasserts /STROBE at least 1 cycle
; after an I/O operation.  So, assuming the delay PLD uses a clock
; as least as fast as that used on the 9001, we are sure that the
; delay PLD will be "cleared" before we use it to delay the current
; I/O operation.

; Once the /DTACK signal is asserted (after waiting for the appropriate
; delay line) it remains asserted until the processor disasserts /STROBE


/DTACK :=     /STROBE *
                 (((UART_READ + UART_WRITE) * UART_DELAY)
                + (/RAM_SELECT * RAM_DELAY)
+ (IO_SELECT * IO_DELAY)
                + ((/ROM_SELECT + BOGUS_SELECT) * ROM_DELAY))
             + /DTACK * /STROBE


SIMULATION

TRACE_ON STROBE A30 A31 UART_DELAY RAM_DELAY ROM_DELAY IO_DELAY RW
 UART_READ UART_WRITE RAM_SELECT ROM_SELECT IO_SELECT DELAY_IN DTACK WE

PRLDF DTACK


SETF STROBE /A30 /A31 /UART_DELAY /RAM_DELAY /ROM_DELAY /IO_DELAY RW
```

```
CLOCKF CLK
SETF /STROBE
CLOCKF CLK
CLOCKF CLK
SETF ROM_DELAY
CLOCKF CLK
CLOCKF CLK
SETF STROBE
CLOCKF CLK
SETF /ROM_DELAY
CLOCKF CLK

SETF A30; read RAM
CLOCKF CLK
SETF /STROBE
CLOCKF CLK
CLOCKF CLK
SETF RAM_DELAY
CLOCKF CLK
CLOCKF CLK
SETF STROBE
CLOCKF CLK
CLOCKF CLK
SETF /RAM_DELAY

SETF /A30 A31   ; read uart immediately
CLOCKF CLK
SETF /STROBE
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF UART_DELAY
CLOCKF CLK
CLOCKF CLK
SETF STROBE
CLOCKF CLK
SETF /UART_DELAY

SETF A30 A31    ; READ I/O device
CLOCKF CLK
SETF /STROBE
CLOCKF CLK
CLOCKF CLK
SETF UART_DELAY
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF STROBE
CLOCKF CLK
SETF /UART_DELAY
CLOCKF CLK

TRACE_OFF
```

A.2 Design for FM_button

```
Title    FM_button
Pattern  button.pds
Revision B
Author   Ken Albin and Matt Wilding
Company  Computational Logic Inc.
Date     5/11/94 (mod 5/7/95 to add extra button/light)

; This pld is works as an I/O interface between the FM address-decoding
; PLD and a button and light

; inputs:

; CLK  - processor clock
; IO_SELECT - generated by address-decoding chip based on memory address
; RW - read/write signal from processor - read high, write low
; SIGNAL1 - signal from button1
; SIGNAL2 - signal from button2
; DEVICE_SELECT - a1, the 2nd lsb of address

; input/output:

; D0 - lowest data bit.  The button value for reads, the light for writes
; LIGHT1 - register that controls light1
; LIGHT2 - register that controls light2



CHIP button PAL22V10

CLK IO_SELECT RW SIGNAL1 SIGNAL2 DEVICE_SELECT I6 I7 I8 I9 I10 GND
I11 D0_TEMP LIGHT1 D0 LIGHT2 IO4 IO5 IO6 IO7 IO8 OLD_IO_SELECT VCC



EQUATIONS

; OLD_IO_SELECT will be used as a register containing the most recent
; value of IO_SELECT.  (This allows us to latch the "first" value of
; SIGNAL when IO_SELECT goes high)

OLD_IO_SELECT := IO_SELECT

; D0 is the first value of the signal after IO_SELECT becomes active
; note that we assume that IO_SELECT becomes inactive for at least
; one cycle to clear D0
;
; We use D0_TEMP and D0 so that we can use the signal when it is
; disconnected.  (Probably would work anyway, but not doing it
; this way confuses the simulator.)

D0_TEMP := (/OLD_IO_SELECT * IO_SELECT * ((device_select * signal2) +
           (/device_select * signal1))) + (D0_TEMP * IO_SELECT)

; Don't screw up D0 unless we are really reading the button

D0 = D0_TEMP
D0.TRST = RW * IO_SELECT
```

```
; LIGHTs are updated by a write to the I/O device, and stay constant
; until the next write
LIGHT1 := (IO_SELECT * /RW * /device_select * D0) +
          (LIGHT1 * (/IO_SELECT + RW + device_select))

LIGHT2 := (IO_SELECT * /RW * device_select * D0) +
          (LIGHT2 * (/IO_SELECT + RW + /device_select))


SIMULATION

TRACE_ON CLK IO_SELECT RW SIGNAL1 signal2 DEVICE_SELECT D0_TEMP D0 LIGHT1 light2 OLD_IO_SELECT

SETF /IO_SELECT DEVICE_SELECT RW SIGNAL1 /signal2 D0

PRLDF LIGHT1 LIGHT2 D0_TEMP OLD_IO_SELECT

;reads of the buttons
CLOCKF CLK
CLOCKF CLK
SETF IO_SELECT
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF /IO_SELECT
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK

SETF /device_select
SETF signal2
CLOCKF CLK
CLOCKF CLK
SETF IO_SELECT
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF /IO_SELECT
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK

; Write to the lights

CLOCKF CLK
SETF /RW /D0
CLOCKF CLK
SETF IO_SELECT
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF /IO_SELECT
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK

setf device_select
CLOCKF CLK
SETF /RW /D0
CLOCKF CLK
SETF IO_SELECT
```

```
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK
SETF /IO_SELECT
CLOCKF CLK
CLOCKF CLK
CLOCKF CLK

TRACE_OFF
```

A.3 Wiring Changes

The following wiring changes were made to the FM9001 board. The pin numbers are references to the FM9001 board [2].

1. REMOVE UART_TXRDY (UART Pin 24) from FM_addr Pin 5
2. REMOVE UART_RXRDY (UART Pin 29) from FM_addr Pin 6
3. ADD DEL80 (delay Pin 16) to IO_DELAY (FM_addr Pin 6)
4. ADD IO_SELECT (button Pin 2) to IO_SELECT (FM_addr Pin 18)
5. ADD VCC to button Pin 24
6. ADD ground to button Pin 12
7. ADD D0 (connector J1 pin B7) to D0 (button 16)
8. ADD RW (connector J2 pin A9) to RW (button 3)
9. ADD clk (FM_addr 1) to clk (button 1)
10. ADD physical button pin 1 to SIGNAL (button 4)
11. ADD physical button pin 14 to VCC
12. ADD physical button pin 7 to GND
13. ADD LIGHT (button 15) to LS244 pin 2
14. ADD LS244 pin 1 to GND
15. ADD LS244 pin 10 to GND
16. ADD LS244 pin 20 to VCC
17. ADD LS244 pin 18 to physical light pin 7
18. ADD physical light pin 2 to VCC
19. REMOVE UART_DELAY (FM_addr Pin 7) to delay tap
20. REMOVE RAM_DELAY (FM_addr Pin 8) to delay tap
21. REMOVE ROM_DELAY (FM_addr Pin 9) to delay tap
22. ADD IO_DELAY (FM_addr Pin 6) to RAM_DELAY (FM_addr Pin 8)
23. ADD UART_DELAY (FM_addr Pin 7) to ROM_DELAY (FM_addr Pin 9)
24. ADD RAM_DELAY (FM_addr Pin 8) to ROM_DELAY (FM_addr Pin 9)
25. REMOVE CLK (delay Pin 1) to 25 Mz clock
26. ADD CLK (delay Pin 1) to CLK (FM_addr Pin 1)

27. REMOVE CLK (FM_addr Pin 1) to CLK (FM_RESET Pin 1)
28. REMOVE CLK (FM_addr Pin 1) to CLK (10 Mz clock)
29. ADD CLK (FM_delay Pin 1) to CLK (FM_RESET Pin 1)
30. ADD CLK (FM_delay Pin 1) to CLK (10 Mz clock)

Appendix B

Nim-playing Piton Program Listing

This appendix contains a listing of the nim Piton program. The listing in this appendix has been changed into a more-standard syntax for readability. Appendix D contains the "official" version about which the correctness proofs have been accomplished in function 'cm-prog'.

```
SUBROUTINE XOR-BVS (VECS-ADDR NUMVECS)
          PUSH-LOCAL VECS-ADDR
          FETCH
          PUSH-LOCAL NUMVECS
          SUB1-NAT
          POP-LOCAL NUMVECS
   LOOP:  PUSH-LOCAL NUMVECS
          TEST-NAT-AND-JUMP ZERO DONE
          PUSH-LOCAL NUMVECS
          SUB1-NAT
          POP-LOCAL NUMVECS
          PUSH-LOCAL VECS-ADDR
          PUSH-CONSTANT (NAT 1)
          ADD-ADDR
          SET-LOCAL VECS-ADDR
          FETCH XOR-BITV
          JUMP LOOP
   DONE:  RET

SUBROUTINE PUSH-1-VECTOR NIL
          PUSH-CONSTANT (BITV (0 0 0 0 0 0 0 1))
          RET

SUBROUTINE NAT-TO-BV (VALUE) (CURRENT-BIT TEMP)
          CALL PUSH-1-VECTOR
          POP-LOCAL CURRENT-BIT
          CALL PUSH-1-VECTOR
          RSH-BITV
   LOOP:  PUSH-LOCAL VALUE
          TEST-NAT-AND-JUMP ZERO DONE
          PUSH-LOCAL VALUE
          DIV2-NAT
          POP-LOCAL TEMP
          POP-LOCAL VALUE
          PUSH-LOCAL TEMP
          TEST-NAT-AND-JUMP ZERO LAB
          PUSH-LOCAL CURRENT-BIT
```

```
          XOR-BITV
   LAB:   PUSH-LOCAL CURRENT-BIT
          LSH-BITV
          POP-LOCAL CURRENT-BIT
          JUMP LOOP
   DONE:  RET

SUBROUTINE BV-TO-NAT
            (BV) (CURRENT-BIT CURRENT-2POWER)
          PUSH-CONSTANT (NAT 1)
          POP-LOCAL CURRENT-2POWER
          CALL PUSH-1-VECTOR
          POP-LOCAL CURRENT-BIT
          PUSH-CONSTANT (NAT 0)
   LOOP:  PUSH-LOCAL BV
          PUSH-LOCAL CURRENT-BIT
          AND-BITV
          TEST-BITV-AND-JUMP ALL-ZERO LAB
          PUSH-LOCAL CURRENT-2POWER
          ADD-NAT
   LAB:   PUSH-LOCAL CURRENT-BIT
          LSH-BITV
          SET-LOCAL CURRENT-BIT
          TEST-BITV-AND-JUMP ALL-ZERO DONE
          PUSH-LOCAL CURRENT-2POWER
          MULT2-NAT
          POP-LOCAL CURRENT-2POWER
          JUMP LOOP
   DONE:  RET

SUBROUTINE NUMBER-WITH-AT-LEAST
            (NUMS-ADDR NUMNUMS MIN) (I)
          PUSH-CONSTANT (NAT 0)
          SET-LOCAL I
   LOOP:  PUSH-LOCAL NUMS-ADDR
          FETCH
          PUSH-LOCAL MIN
          LT-NAT
          TEST-BOOL-AND-JUMP T LAB
          ADD1-NAT
   LAB:   PUSH-LOCAL NUMNUMS
          PUSH-LOCAL I
          ADD1-NAT
          SET-LOCAL I
          SUB-NAT
          TEST-NAT-AND-JUMP ZERO DONE
          PUSH-LOCAL NUMS-ADDR
          PUSH-CONSTANT (NAT 1)
          ADD-ADDR
          POP-LOCAL NUMS-ADDR
          JUMP LOOP
   DONE:  RET

SUBROUTINE HIGHEST-BIT (BV) (CB)
          CALL PUSH-1-VECTOR
          SET-LOCAL CB
          RSH-BITV
   LOOP:  PUSH-LOCAL CB
          TEST-BITV-AND-JUMP ALL-ZERO DONE
          PUSH-LOCAL BV
          PUSH-LOCAL CB
          AND-BITV
```

```
         TEST-BITV-AND-JUMP ALL-ZERO LAB
         POP
         PUSH-LOCAL CB
   LAB:  PUSH-LOCAL CB
         LSH-BITV
         POP-LOCAL CB
         JUMP LOOP
   DONE: RET

SUBROUTINE MATCH-AND-XOR
           (VECS NUMVECS MATCH XOR-VECTOR) (I)
         PUSH-CONSTANT (NAT 0)
         POP-LOCAL I
   LOOP: PUSH-LOCAL VECS
         FETCH
         PUSH-LOCAL MATCH
         AND-BITV
         TEST-BITV-AND-JUMP NOT-ALL-ZEROS FOUND
         PUSH-LOCAL I
         ADD1-NAT
         SET-LOCAL I
         PUSH-LOCAL NUMVECS
         LT-NAT
         TEST-BOOL-AND-JUMP F DONE
         PUSH-LOCAL VECS
         PUSH-CONSTANT (NAT 1)
         ADD-ADDR
         POP-LOCAL VECS
         JUMP LOOP
   FOUND: PUSH-LOCAL VECS
         FETCH
         PUSH-LOCAL XOR-VECTOR
         XOR-BITV
         PUSH-LOCAL VECS
         DEPOSIT
   DONE: RET

SUBROUTINE REPLACE-VALUE (LIST OLDVAL NEWVAL)
   LOOP: PUSH-LOCAL LIST
         FETCH
         PUSH-LOCAL OLDVAL
         EQ
         TEST-BOOL-AND-JUMP T DONE
         PUSH-LOCAL LIST
         PUSH-CONSTANT (NAT 1)
         ADD-ADDR
         POP-LOCAL LIST
         JUMP LOOP
   DONE: PUSH-LOCAL NEWVAL
         PUSH-LOCAL LIST
         DEPOSIT RET

SUBROUTINE NAT-TO-BV-LIST
           (NAT-LIST BV-LIST LENGTH) (I 0)
   LOOP: PUSH-LOCAL NAT-LIST
         FETCH
         CALL NAT-TO-BV
         PUSH-LOCAL BV-LIST
         DEPOSIT
         PUSH-LOCAL I
         ADD1-NAT
         SET-LOCAL I
```

```
         PUSH-LOCAL LENGTH
         EQ
         TEST-BOOL-AND-JUMP T DONE
         PUSH-LOCAL NAT-LIST
         PUSH-CONSTANT (NAT 1)
         ADD-ADDR
         POP-LOCAL NAT-LIST
         PUSH-LOCAL BV-LIST
         PUSH-CONSTANT (NAT 1)
         ADD-ADDR
         POP-LOCAL BV-LIST
         JUMP LOOP
   DONE: RET

SUBROUTINE BV-TO-NAT-LIST
           (BV-LIST NAT-LIST LENGTH) (I 0)
   LOOP: PUSH-LOCAL BV-LIST
         FETCH     CALL BV-TO-NAT
         PUSH-LOCAL NAT-LIST
         DEPOSIT     PUSH-LOCAL I
         ADD1-NAT
         SET-LOCAL I
         PUSH-LOCAL LENGTH     EQ
         TEST-BOOL-AND-JUMP T DONE
         PUSH-LOCAL NAT-LIST
         PUSH-CONSTANT (NAT 1)
         ADD-ADDR
         POP-LOCAL NAT-LIST
         PUSH-LOCAL BV-LIST
         PUSH-CONSTANT (NAT 1)
         ADD-ADDR
         POP-LOCAL BV-LIST
         JUMP LOOP
   DONE: RET

SUBROUTINE MAX-NAT (NAT-LIST LENGTH) (I 0 J)
         PUSH-CONSTANT (NAT 0)
   LOOP: SET-LOCAL J
         PUSH-LOCAL J
         PUSH-LOCAL NAT-LIST
         FETCH
         SET-LOCAL J    LT-NAT
         TEST-BOOL-AND-JUMP F LAB
         POP
         PUSH-LOCAL J
   LAB:  PUSH-LOCAL I
         ADD1-NAT
         SET-LOCAL I
         PUSH-LOCAL LENGTH
         EQ
         TEST-BOOL-AND-JUMP T DONE
         PUSH-LOCAL NAT-LIST
         PUSH-CONSTANT (NAT 1)
         ADD-ADDR
         POP-LOCAL NAT-LIST
         JUMP LOOP
   DONE: RET

SUBROUTINE SMART-MOVE
           (STATE NUMPILES WORK-AREA) (I)
         PUSH-LOCAL STATE
         PUSH-LOCAL NUMPILES
```

```
         PUSH-CONSTANT (NAT 2)
         CALL NUMBER-WITH-AT-LEAST
         PUSH-CONSTANT (NAT 2)
         LT-NAT
         TEST-BOOL-AND-JUMP T LAB
         PUSH-LOCAL STATE
         PUSH-LOCAL WORK-AREA
         PUSH-LOCAL NUMPILES
         CALL NAT-TO-BV-LIST
         PUSH-LOCAL WORK-AREA
         PUSH-LOCAL NUMPILES
         PUSH-LOCAL WORK-AREA
         PUSH-LOCAL NUMPILES
         CALL XOR-BVS
         SET-LOCAL I
         CALL HIGHEST-BIT
         PUSH-LOCAL I
         CALL MATCH-AND-XOR
         PUSH-LOCAL WORK-AREA
         PUSH-LOCAL STATE
         PUSH-LOCAL NUMPILES
         CALL BV-TO-NAT-LIST
         RET
   LAB:  PUSH-LOCAL STATE
         PUSH-LOCAL STATE
         PUSH-LOCAL NUMPILES
         CALL MAX-NAT
         PUSH-LOCAL STATE
         PUSH-LOCAL NUMPILES
         PUSH-CONSTANT (NAT 1)
         CALL NUMBER-WITH-AT-LEAST
         DIV2-NAT
         POP-LOCAL I
         POP
         PUSH-LOCAL I
         CALL REPLACE-VALUE
         RET

SUBROUTINE DELAY (TIME)
   LAB:  PUSH-LOCAL TIME
         SUB1-NAT
         SET-LOCAL TIME
         NO-OP NO-OP NO-OP NO-OP
         TEST-NAT-AND-JUMP ZERO DONE
         NO-OP
         JUMP LAB
   DONE: RET

SUBROUTINE COMPUTER-MOVE
           (STATE NUMPILES WORK-AREA) (I)
         PUSH-CONSTANT (NAT 250)
         CALL DELAY
         PUSH-CONSTANT (NAT 250)
         CALL DELAY
         PUSH-CONSTANT (NAT 250)
         CALL DELAY
         PUSH-CONSTANT (NAT 250)
         CALL DELAY
         PUSH-LOCAL STATE
         PUSH-LOCAL NUMPILES
         PUSH-CONSTANT (NAT 2)
         CALL NUMBER-WITH-AT-LEAST
```

```
        TEST-NAT-AND-JUMP ZERO LAB
        PUSH-LOCAL STATE
        PUSH-LOCAL WORK-AREA
        PUSH-LOCAL NUMPILES
        CALL NAT-TO-BV-LIST
        PUSH-LOCAL WORK-AREA
        PUSH-LOCAL NUMPILES
        CALL XOR-BVS
        TEST-BITV-AND-JUMP ALL-ZERO LAB
        PUSH-LOCAL STATE
        PUSH-LOCAL NUMPILES
        PUSH-LOCAL WORK-AREA
        CALL SMART-MOVE
        RET
LAB:    PUSH-LOCAL STATE
        PUSH-LOCAL STATE
        PUSH-LOCAL NUMPILES
        CALL MAX-NAT
        POP-LOCAL I
        PUSH-LOCAL I
        PUSH-LOCAL I
        SUB1-NAT
        CALL REPLACE-VALUE
        RET
```

Appendix C

"scheduler.events"

```
(proveall "scheduler"
'(
#|
 Copyright (C) 1995 by Matthew Wilding and Computational Logic, Inc.
 All Rights Reserved.

 This script is hereby placed in the public domain, and therefore unlimited
 editing and redistribution is permitted.

 NO WARRANTY

 Matthew Wilding and Computational Logic, Inc. PROVIDES ABSOLUTELY NO
 WARRANTY.  THE EVENT SCRIPT IS PROVIDED "AS IS" WITHOUT WARRANTY OF
 ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
 ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE.  THE ENTIRE RISK AS TO THE QUALITY AND
 PERFORMANCE OF THE SCRIPT IS WITH YOU.  SHOULD THE SCRIPT PROVE
 DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR
 CORRECTION.

 IN NO EVENT WILL Matthew Wilding and Computational Logic, Inc. BE
 LIABLE TO YOU FOR ANY DAMAGES, ANY LOST PROFITS, LOST MONIES, OR
 OTHER SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE
 USE OR INABILITY TO USE THIS SCRIPT (INCLUDING BUT NOT LIMITED TO
 LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY
 THIRD PARTIES), EVEN IF YOU HAVE ADVISED US OF THE POSSIBILITY OF
 SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

This file contains the events that lead to the proof of the optimality
of an earliest-deadline-first scheduler on any set of periodic tasks.

|#

;; We use naturals library that is part of the Nqthm-1992 distribution
;; available from Computational Logic, Inc.
(note-lib "/slocal/src/nqthm-1992/examples/numbers/naturals" t)

(defn tk-name (pt) (car pt))

(defn tk-period (pt) (cadr pt))

(defn tk-duration (pt) (caddr pt))

; periodic task is a triple (name period duration)
(defn periodic-taskp (pt)
  (and
   (listp pt)
   (litatom (tk-name pt))
   (not (equal (tk-name pt) nil))
   (lessp 0 (tk-period pt))
   (lessp 0 (tk-duration pt))
```

```
   (equal nil (cdr (cdr (cdr pt))))))

(defn periodic-tasksp (pts)
  (if (listp pts)
      (and
       (periodic-taskp (car pts))
       (not (assoc (tk-name (car pts)) (cdr pts)))
       (periodic-tasksp (cdr pts)))
    (equal pts nil)))

;; a task request has the form (name request-time deadline-time duration)

;; generate the requests for a periodic task during a time period
(defn periodic-task-requests (pt starting-time ending-time)
  (if (periodic-taskp pt)
      (if (lessp starting-time ending-time)
          (cons (list (tk-name pt) starting-time
                      (plus starting-time (tk-period pt)) (tk-duration pt))
                (periodic-task-requests pt (plus starting-time (tk-period pt))
                                        ending-time))
        nil)
    nil)
  ((lessp (difference ending-time starting-time))))

;; we generate a list of task request lists
(defn periodic-tasks-requests (pts starting-time ending-time)
  (if (periodic-tasksp pts)
      (if (listp pts)
          (append
           (periodic-task-requests (car pts) starting-time ending-time)
           (periodic-tasks-requests (cdr pts) starting-time ending-time))
        nil)
    nil))

(defn repeat (n val)
  (if (zerop n)
      nil
    (cons val (repeat (sub1 n) val))))

;; produce an "obvious" schedule of length c that consists of
;; (duration/period)*c calls of each task in a periodic task list
(defn substring-schedule (pts bigp)
  (if (listp pts)
      (append
       (repeat (quotient (times bigp (caddar pts)) (cadar pts)) (caar pts))
       (substring-schedule (cdr pts) bigp))
    nil))

(defn make-length (length list fill)
  (if (zerop length) nil
    (if (listp list)
        (cons (car list) (make-length (sub1 length) (cdr list) fill))
      (cons fill (make-length (sub1 length) nil fill)))))

(defn repeat-list (list times)
  (if (zerop times) nil
    (append list (repeat-list list (sub1 times)))))

(defn make-simple-schedule (pts bigp length)
  (repeat-list (make-length bigp (substring-schedule pts bigp) nil)
             (quotient length bigp)))
```

```
(defn firstn (n list)
  (if (zerop n) nil
    (cons (car list) (firstn (sub1 n) (cdr list)))))

(defn nth (n list)
  (if (zerop n) (car list)
    (nth (sub1 n) (cdr list))))

(defn nthcdr (n list)
  (if (zerop n) list
    (nthcdr (sub1 n) (cdr list))))


(defn length (list)
  (if (listp list)
      (add1 (length (cdr list)))
    0))

(prove-lemma length-append (rewrite)
  (equal
   (length (append x y))
   (plus (length x) (length y))))

(prove-lemma length-firstn (rewrite)
  (equal (length (firstn n list)) (fix n)))

(prove-lemma length-nthcdr (rewrite)
  (equal (length (nthcdr n list)) (difference (length list) n)))

(prove-lemma equal-length-0 (rewrite)
  (equal
   (equal (length x) 0)
   (not (listp x))))

;; each task request period is a multiple of bigp, and bigp*duration
;; is a multiple of period

(defn expanded-tasksp (ts P)
  (if (listp ts)
      (and
       (zerop (remainder (cadar ts) P))
       (zerop (remainder (caddar ts) P))
       (zerop (remainder (times P (caddar ts)) (cadar ts)))
       (expanded-tasksp (cdr ts) P))
    t))

(defn plist (list)
  (if (listp list)
      (cons (car list) (plist (cdr list)))
    nil))

(defn name (r) (car r))

(defn request-time (r) (cadr r))

(defn deadline (r) (caddr r))

(defn duration (r) (cadddr r))

(deftheory task-abbr (name request-time deadline duration tk-name tk-duration
                           tk-period))
```

```
(defn good-schedule (s r)
  (if (listp r)
      (and
       (equal
        (occurrences
         (name (car r))
         (firstn (difference (deadline (car r)) (request-time (car r)))
                 (nthcdr (request-time (car r)) s)))
         (duration (car r)))
       (good-schedule s (cdr r)))
    t))

(defn big-period (pts)
  (if (listp pts)
      (times (tk-period (car pts)) (big-period (cdr pts)))
    1))

(defn active-task-requests (time r)
  (if (listp r)
      (if (and (lessp time (deadline (car r)))
               (not (lessp time (request-time (car r)))))
          (cons (car r) (active-task-requests time (cdr r)))
        (active-task-requests time (cdr r)))
    nil))

(defn unfulfilled (time s r)
  (if (listp r)
      (if (equal
           (occurrences
            (name (car r))
            (firstn (difference time (request-time (car r)))
                    (nthcdr (request-time (car r)) s)))
           (duration (car r)))
          (unfulfilled time s (cdr r))
        (cons (car r) (unfulfilled time s (cdr r))))
    nil))

;; return a task request with least deadline
(defn least-deadline (r)
  (if (listp r)
      (if (listp (cdr r))
          (if (lessp (deadline (car r)) (deadline (car (cdr r))))
              (least-deadline (cons (car r) (cdr (cdr r))))
            (least-deadline (cdr r)))
        (car r))
    nil)
  ((lessp (length r))))

;; return location of first instance of task in s no earlier than time
(defn first-instance (time task s)
  (if (lessp time (length s))
      (if (equal (nth time s) task)
          time
        (first-instance (add1 time) task s))
    f)
  ((lessp (difference (length s) time))))

(defn replace-nth (n val list)
  (if (zerop n) (cons val (cdr list))
    (cons (car list) (replace-nth (sub1 n) val (cdr list)))))

;; swap locations i and j in list
```

```
(defn swap (i j list)
  (replace-nth i (nth j list) (replace-nth j (nth i list) list)))

(prove-lemma length-replace-nth (rewrite)
  (equal
   (length (replace-nth i val list))
   (if (lessp i (length list)) (length list) (add1 i))))

(prove-lemma length-swap (rewrite)
  (equal (length (swap i j list))
         (if (lessp i (length list))
             (if (lessp j (length list))
                 (length list)
               (add1 j))
           (if (lessp i j)
               (add1 j)
             (add1 i)))))

(defn make-element-edf (s r time)
  (let ((unfulfilled (unfulfilled time s (active-task-requests time r))))
    (let ((first (first-instance time (car (least-deadline unfulfilled)) s)))
      (if (and (listp unfulfilled) first)
          (swap time first s)
        s))))

(prove-lemma lessp-first-instance (rewrite)
  (implies
   (listp s)
   (lessp (first-instance time task s) (length s))))

(prove-lemma length-make-element-edf (rewrite)
  (implies
   (lessp time (length s))
   (equal (length (make-element-edf s r time)) (length s))))

(disable make-element-edf)

(defn make-schedule-edf (s r first)
  (if (lessp first (length s))
      (make-schedule-edf (make-element-edf s r first) r (add1 first))
    s)
  ((lessp (difference (length s) first))))

(enable make-element-edf)

(prove-lemma plist-repeat-list (rewrite)
  (equal (plist (repeat-list s n)) (repeat-list s n)))

(prove-lemma length-repeat-list (rewrite)
  (equal (length (repeat-list s n)) (times n (length s))))

(prove-lemma append-nil (rewrite)
  (equal (append list nil) (plist list)))

(prove-lemma good-schedule-append (rewrite)
  (equal
   (good-schedule s (append r1 r2))
   (and
    (good-schedule s r1)
    (good-schedule s r2))))

;; introduce the useful notion of sublist
```

```
(defn sublistp (a b)
  (if (listp b)
      (if (listp a)
          (if (equal (car a) (car b))
              (sublistp (cdr a) (cdr b))
            (sublistp a (cdr b)))
        t)
    (nlistp a)))

(defn remove-until (v l)
  (if (listp l)
      (if (equal v (car l))
          (cdr l)
        (remove-until v (cdr l)))
    l))

(prove-lemma sublistp-cons-rewrite (rewrite)
  (equal (sublistp (cons c x) y)
         (and
          (member c y)
          (sublistp x (remove-until c y)))))

(prove-lemma listp-remove-until-means-listp (rewrite)
  (implies
   (listp (remove-until a z))
   (listp z)))

(prove-lemma lessp-remove-until (rewrite)
  (equal
   (lessp (length (remove-until a y)) (length y))
   (listp y)))

(prove-lemma remove-until-append (rewrite)
  (equal
   (remove-until a (append x y))
   (if (member a x)
       (append (remove-until a x) y)
     (remove-until a y))))

(defn list-until (v l)
  (if (listp l)
      (if (equal v (car l))
          (list v)
        (cons (car l) (list-until v (cdr l))))
    l))

(prove-lemma append-remove-until-list-until (rewrite)
  (equal (append (list-until v l) (remove-until v l)) l))

;; amazingly complex - easier way? (took me ~3 hours to prove this
;; small lemma)
(defn sublistp-append-induct (a b y z)
  (if (listp a)
      (sublistp-append-induct
       (cdr a) b (append y (list-until (car a) z))
       (remove-until (car a) z))
    (if (nlistp b) t
      (if (listp y)
          (if (member (car b) y)
              (sublistp-append-induct
               (list (car b)) (cdr b) (remove-until (car b) y) z)
            t)
```

```
        t)))
  ((ord-lessp (cons (add1 (length b)) (length a)))))

(prove-lemma member-append
             (rewrite)
             (equal (member a (append x y))
                    (or (member a x) (member a y))))

(prove-lemma listp-bagint-with-singleton-implies-member
             (rewrite)
             (implies (listp (bagint y (list z)))
                      (member z y)))

(prove-lemma bagint-singleton
             (rewrite)
             (equal (bagint x (list y))
                    (if (member y x) (list y) nil)))

(prove-lemma transitivity-of-append
             (rewrite)
             (equal (append (append a b) c)
                    (append a (append b c))))

(prove-lemma sublistp-append (rewrite)
  (implies
   (sublistp (append a b) z)
   (sublistp b (append y z)))
  ((induct (sublistp-append-induct a b y z))))

(prove-lemma sublistp-cdr1 (rewrite)
   (implies
    (sublistp x y)
    (sublistp (cdr x) y))
   ((use (sublistp-append (a (list (car x))) (b (cdr x)) (z y) (y nil)))))

(prove-lemma sublistp-cdr2 (rewrite)
   (implies
    (sublistp x (cdr y))
    (sublistp x y))
   ((use (sublistp-append (a nil) (b x) (y (list (car y))) (z (cdr y))))))

(prove-lemma remainder-big-period-sublist (rewrite)
  (implies
   (and
    (periodic-tasksp y)
    (sublistp x y))
   (equal (remainder (big-period y) (big-period x)) 0)))

(prove-lemma remainder-big-period-cdr (rewrite)
  (implies
   (equal (remainder n (big-period pts)) 0)
   (equal (remainder n (big-period (cdr pts))) 0)))

(prove-lemma repeat-list-plus (rewrite)
  (equal
   (repeat-list l (plus a b))
   (append (repeat-list l a) (repeat-list l b))))

(prove-lemma nthcdr-append (rewrite)
  (equal (nthcdr n (append l1 l2))
        (if (lessp n (length l1))
            (append (nthcdr n l1) l2)
```

```
          (nthcdr (difference n (length l1)) l2))))

(prove-lemma firstn-append (rewrite)
  (equal (firstn n (append l1 l2))
         (if (lessp (length l1) n)
             (append l1 (firstn (difference n (length l1)) l2))
           (firstn n l1))))

(defn nthcdr-repeat-list-induct (n1 n2 list)
  (if (zerop n1) t
    (nthcdr-repeat-list-induct (sub1 n1) (difference n2 (length list)) list)))

(prove-lemma nthcdr-repeat-list (rewrite)
  (implies
   (and
    (equal (remainder n1 (length list)) 0)
    (not (lessp (times n2 (length list)) n1)))
  (equal
   (nthcdr n1 (repeat-list list n2))
   (repeat-list list (difference n2 (quotient n1 (length list))))))
  ((induct (nthcdr-repeat-list-induct n2 n1 list))
   (disable quotient-difference quotient-difference1)))


(prove-lemma length-make-length (rewrite)
  (equal (length (make-length n list fill)) (fix n)))

(prove-lemma member-expanded (rewrite)
  (implies
   (and
    (member tk pts)
    (expanded-tasksp pts bigp))
   (equal (remainder (cadr tk) bigp) 0)))

(prove-lemma firstn-length-list (rewrite)
  (equal (firstn (length x) x) (plist x)))

(prove-lemma firstn-repeat-list (rewrite)
  (implies
   (and
    (equal (remainder n1 (length list)) 0)
    (not (lessp (times n2 (length list)) n1)))
   (equal
    (firstn n1 (repeat-list list n2))
    (repeat-list list (quotient n1 (length list)))))
  ((induct (nthcdr-repeat-list-induct n2 n1 list))
   (disable quotient-difference quotient-difference1)))

(prove-lemma lessp-remainder-special (rewrite)
  (implies
   (and
    (equal (remainder x z) 0)
    (equal (remainder y z) 0))
   (equal (lessp y (plus x z))
          (and
           (not (zerop z))
           (not (lessp x y)))))
  ((induct (double-remainder-induction z x y))))

(prove-lemma lessp-difference-special (rewrite)
  (implies
   (and
```

```
    (equal (remainder x z) 0)
    (equal (remainder y z) 0))
   (equal (lessp (difference x y) z)
          (and
           (not (zerop z))
           (not (lessp y x)))))
  ((induct (double-remainder-induction z x y))))

(prove-lemma occurrences-append
             (rewrite)
             (equal (occurrences a (append x y))
                    (plus (occurrences a x)
                          (occurrences a y))))

(prove-lemma occurrences-repeat-list (rewrite)
  (equal
   (occurrences v (repeat-list list n))
   (times n (occurrences v list))))

(prove-lemma occurrences-make-length (rewrite)
  (equal (occurrences v (make-length size list fill))
         (if (lessp size (length list))
             (occurrences v (firstn size list))
           (if (equal v fill)
               (plus (occurrences v list) (difference size (length list)))
             (occurrences v list))))
  ((expand (make-length 1 list fill))))

(prove-lemma occurrences-repeat (rewrite)
  (equal (occurrences x (repeat n y))
         (if (equal x y) (fix n) 0)))

(prove-lemma member-repeat (rewrite)
  (equal
   (member x (repeat n v))
   (and
    (lessp 0 n)
    (equal x v))))

(prove-lemma member-substring-schedule (rewrite)
  (implies
   (and
    (periodic-tasksp z)
    (expanded-tasksp z bigp)
    (not (equal v nil)))
   (iff (member v (substring-schedule z bigp))
        (assoc v z)))
  ((induct (assoc v z))))

(prove-lemma member-car-x-x
  (rewrite)
  (equal (member (car x) x) (listp x)))

(prove-lemma occurrences-substring-schedule (rewrite)
  (implies
   (and
    (member tk pts)
    (lessp 0 bigp)
    (periodic-tasksp pts)
    (expanded-tasksp pts bigp))
   (equal (occurrences (car tk) (substring-schedule pts bigp))
          (quotient (times bigp (caddr tk)) (cadr tk)))))
```

```
(prove-lemma times-quotient-quotient-special (rewrite)
  (implies
   (and
    (equal (remainder x bigp) 0)
    (equal (remainder (times bigp y) x) 0)
    (lessp 0 bigp))
   (equal (times (quotient x bigp)
                 (quotient (times bigp y) x))
          (fix y))))

(prove-lemma good-schedule-periodic-task-requests (rewrite)
  (implies
   (and
    (member tk pts)
    (periodic-tasksp pts)
    (expanded-tasksp pts bigp)
    (lessp 0 bigp)
    (not (lessp n2 n1))
    (numberp n2)
    (numberp n1)
    (zerop (remainder (cadr tk) bigp))
    (zerop (remainder (times bigp (caddr tk)) (cadr tk)))
    (zerop (remainder n1 bigp))
    (zerop (remainder n1 (cadr tk)))
    (zerop (remainder n2 bigp))
    (zerop (remainder n2 (big-period pts)))
    (zerop (remainder n2 (cadr tk)))
    (not (lessp bigp (length (substring-schedule pts bigp)))))
   (good-schedule (repeat-list
                   (make-length bigp (substring-schedule pts bigp) nil)
                   (quotient n2 bigp))
                  (periodic-task-requests tk n1 n2)))
  ((induct (periodic-task-requests tk n1 n2))))

(prove-lemma member-sublistp (rewrite)
  (implies
   (and
    (sublistp x y)
    (member e x))
   (member e y)))

(prove-lemma member-expanded-tasksp-means (rewrite)
  (implies
   (and
    (expanded-tasksp pts bigp)
    (member tk pts))
   (and
    (equal (remainder (cadr tk) bigp) 0)
    (equal (remainder (times bigp (caddr tk)) (cadr tk)) 0))))

(prove-lemma remainder-period-if-remainder-big-period (rewrite)
  (implies
   (and
    (periodic-tasksp pts)
    (member tk pts)
    (zerop (remainder n (big-period pts))))
   (equal (remainder n (cadr tk)) 0)))

(prove-lemma good-simple-schedule-sublist nil
           (implies
            (and
```

```
             (not (lessp bigp (length (substring-schedule pts2 bigp))))
             (equal n1 0)
             (sublistp pts1 pts2)
             (periodic-tasksp pts2)
             (expanded-tasksp pts2 bigp)
             (lessp 0 bigp)
             (numberp n2)
             (zerop (remainder n2 bigp))
             (zerop (remainder n2 (big-period pts2))))
            (good-schedule (make-simple-schedule pts2 bigp n2)
                           (periodic-tasks-requests pts1 n1 n2)))
           ((induct (periodic-tasks-requests pts1 n1 n2))))

(prove-lemma sublistp-x-x (rewrite)
  (sublistp x x))

(prove-lemma periodic-tasks-requests-simple (rewrite)
  (implies
   (not (lessp n1 n2))
   (equal (periodic-tasks-requests pts n1 n2) nil)))

(prove-lemma good-simple-schedule (rewrite)
           (implies
            (and
             (not (lessp bigp (length (substring-schedule pts bigp))))
             (periodic-tasksp pts)
             (expanded-tasksp pts bigp)
             (lessp 0 bigp)
             (zerop (remainder n bigp))
             (zerop (remainder n (big-period pts))))
            (good-schedule (make-simple-schedule pts bigp n)
                           (periodic-tasks-requests pts 0 n)))
           ((use (good-simple-schedule-sublist (n2 n) (n1 0) (pts2 pts)
                                               (pts1 pts)))
            (disable-theory t)
            (enable sublistp-x-x periodic-tasks-requests-simple good-schedule)
            (enable-theory ground-zero task-abbr)))


(defn expand-tasks (pts bigp)
  (if (listp pts)
      (cons
       (list (caar pts) (times bigp (cadar pts)) (times bigp (caddar pts)))
       (expand-tasks (cdr pts) bigp))
    nil))

;;;; expanded-tasksp identifies expand-tasks
(prove-lemma expanded-tasksp-expand-task-helper (rewrite)
  (implies
   (and
    (equal (remainder x (big-period pts1)) 0)
    (not (zerop x)))
   (expanded-tasksp (expand-tasks pts1 x) x)))

(prove-lemma zerop-big-period (rewrite)
  (implies
   (periodic-tasksp pts)
   (lessp 0 (big-period pts))))

(prove-lemma expanded-tasksp-expand-task (rewrite)
  (implies
   (periodic-tasksp pts)
```

```
   (expanded-tasksp (expand-tasks pts (big-period pts)) (big-period pts))))

(prove-lemma assoc-expand-tasks (rewrite)
  (implies
   (periodic-tasksp x)
  (iff
   (assoc v (expand-tasks x n))
   (assoc v x))))

(prove-lemma periodic-tasksp-expand-tasks (rewrite)
  (implies
   (periodic-tasksp pts)
  (equal
   (periodic-tasksp (expand-tasks pts n))
   (or
    (not (zerop n))
    (not (listp pts))))))

(defn non-overlapping-requests3 (request request-list)
  (if (listp request-list)
      (and
       (or
        (not (equal (car request) (caar request-list)))
        (not (lessp (cadr request) (caddar request-list)))
        (not (lessp (cadar request-list) (caddr request)))
        (equal (car request-list) request))
       (non-overlapping-requests3 request (cdr request-list)))
    t))


(defn non-overlapping-requests2 (r1 r2)
  (if (listp r1)
      (and
       (non-overlapping-requests3 (car r1) r2)
       (non-overlapping-requests2 (cdr r1) r2))
    t))

(defn non-overlapping-requests (r)
  (non-overlapping-requests2 r r))

(defn double-cdr-induction (a b)
  (if (listp a)
      (double-cdr-induction (cdr a) (cdr b))
    t))

(prove-lemma plist-firstn (rewrite)
  (equal (plist (firstn n l)) (firstn n l)))

(prove-lemma cons-nth-nthcdr (rewrite)
  (implies
   (lessp n (length l))
  (equal
   (cons (nth n l) (nthcdr n (cdr l)))
   (nthcdr n l))))

(prove-lemma equal-append (rewrite)
  (equal
   (equal (append a b) y)
   (if (nlistp y)
       (and
        (nlistp a)
        (equal b y))
```

```
    (and
     (not (lessp (length y) (length a)))
     (equal (firstn (length a) y) (plist a))
     (equal (nthcdr (length a) y) b))))
  ((induct (double-cdr-induction a y))))


(prove-lemma replace-nth-replace-nth (rewrite)
  (implies
   (not (equal (fix i) (fix j)))
   (equal
    (replace-nth i x (replace-nth j y l))
    (replace-nth j y (replace-nth i x l)))))

(prove-lemma replace-nth-idempotent(rewrite)
  (implies
   (equal (fix i) (fix j))
   (equal (replace-nth i x (replace-nth j y l))
          (replace-nth i x l))))

(prove-lemma member-nth (rewrite)
  (implies
   (lessp n (length list))
   (member (nth n list) list)))

(prove-lemma swap-commutative (rewrite)
  (equal
   (swap j i s)
   (swap i j s)))

(prove-lemma member-replace-nth (rewrite)
  (implies
   (not (member x l))
   (equal
    (member x (replace-nth i v l))
    (or
     (equal x v)
     (and (equal x 0) (lessp (length l) i))))))

(prove-lemma occurrences-replace-nth (rewrite)
  (equal
   (occurrences x (replace-nth n v list))
   (difference
    (plus
     (occurrences x list)
     (if (equal v x) 1 0)
     (if (equal x 0) (difference n (length list)) 0))
    (if (and (equal x (nth n list)) (lessp n (length list))) 1 0)))
  ((do-not-generalize t)))

(prove-lemma nthcdr-1 (rewrite)
  (equal (nthcdr 1 x) (cdr x)))

(prove-lemma nlistp-nthcdr (rewrite)
  (implies
   (not (listp s))
   (equal (nthcdr n s) (if (zerop n) s 0))))

(prove-lemma nthcdr-firstn-plus (rewrite)
  (and
   (equal (nthcdr n (firstn (plus n x) s))
          (firstn x (nthcdr n s)))
```

```
   (equal (nthcdr n (firstn (plus x n) s))
          (firstn x (nthcdr n s)))))

(prove-lemma cdr-nthcdr-cons (rewrite)
  (equal
   (cdr (nthcdr x (cons a b)))
   (nthcdr x b)))

(defn add1-sub1-induct (a b)
  (if (zerop b) t
    (add1-sub1-induct (add1 a) (sub1 b))))

(prove-lemma nth-nthcdr (rewrite)
  (equal
   (nth n1 (nthcdr n2 s))
   (nth (plus n1 n2) s))
  ((induct (add1-sub1-induct n1 n2))))

(prove-lemma nthcdr-nthcdr (rewrite)
  (equal
   (nthcdr n1 (nthcdr n2 s))
   (nthcdr (plus n1 n2) s))
  ((induct (add1-sub1-induct n1 n2))))

(prove-lemma equal-append-a-append-a (rewrite)
  (equal
   (equal (append a x) (append a y))
   (equal x y)))

(prove-lemma nthcdr-replace-nth (rewrite)
  (equal
   (nthcdr n (replace-nth i v l))
   (if (lessp i n)
       (nthcdr n l)
     (replace-nth (difference i n) v (nthcdr n l)))))

(prove-lemma firstn-replace-nth (rewrite)
  (equal
   (firstn n (replace-nth i v l))
   (if (lessp i n)
       (replace-nth i v (firstn n l))
     (firstn n l))))

(prove-lemma cdr-firstn-cons (rewrite)
  (equal
   (cdr (firstn n (cons a b)))
   (if (zerop n) 0 (firstn (sub1 n) b))))

(prove-lemma nth-firstn (rewrite)
  (equal
   (nth n (firstn n2 s))
   (if (lessp n n2) (nth n s) 0)))

(prove-lemma firstn-cons (rewrite)
  (equal
   (firstn n (cons a b))
   (if (zerop n) nil
     (cons a (firstn (sub1 n) b)))))

(defn double-sub1-induction (a b)
  (if (zerop a) t
    (double-sub1-induction (sub1 a) (sub1 b))))
```

```
(prove-lemma equal-repeat-repeat (rewrite)
  (equal
   (equal (repeat n v) (repeat n2 v2))
   (and
    (equal (fix n) (fix n2))
    (or
     (equal v v2)
     (zerop n))))
  ((induct (double-sub1-induction n n2))))

(prove-lemma firstn-too-big (rewrite)
  (implies
   (lessp (length x) n)
   (equal (firstn n x) (append x (repeat (difference n (length x)) 0)))))

(prove-lemma firstn-firstn (rewrite)
  (equal
   (firstn a (firstn b x))
   (if (lessp b a)
       (append (firstn b x) (repeat (difference a b) 0))
     (firstn a x))))

(prove-lemma plist-repeat (rewrite)
  (equal (plist (repeat n x)) (repeat n x)))

(prove-lemma firstn-1 (rewrite)
  (equal (firstn 1 s) (list (car s))))

(prove-lemma nthcdr-cons-firstn (rewrite)
  (equal
   (nthcdr w (cons a (firstn (plus w x) b)))
   (if (zerop w) (cons a (firstn x b))
     (nthcdr (sub1 w) (firstn (plus w x) b)))))

(prove-lemma nthcdr-sub1-firstn-plus (rewrite)
  (implies
   (not (zerop w))
   (equal
    (nthcdr (sub1 w) (firstn (plus w x) b))
    (firstn (add1 x) (nthcdr (sub1 w) b))))
  ((use (nthcdr-firstn-plus (n (sub1 w)) (x (add1 x)) (s b)))
   (disable nthcdr-firstn-plus)))

(prove-lemma nthcdr-repeat (rewrite)
  (equal (nthcdr n1 (repeat n2 x))
         (if (lessp n2 n1) 0 (repeat (difference n2 n1) x))))

(prove-lemma firstn-nlistp (rewrite)
  (implies
   (nlistp l)
   (equal (firstn n l) (repeat n 0))))

(defn member-nth-firstn-induction (i y s)
  (if (zerop i) t
    (member-nth-firstn-induction (sub1 i) (sub1 y) (cdr s))))

(prove-lemma member-nth-firstn (rewrite)
  (implies
   (lessp i j)
   (member (nth i s) (firstn j s))))
```

```
(prove-lemma member-car-firstn (rewrite)
  (equal
   (member (car x) (firstn n x))
   (not (zerop n))))

(prove-lemma member-nth-firstn-nthcdr (rewrite)
  (implies
   (and
    (not (lessp i y))
    (lessp i (plus x y)))
  (member (nth i s) (firstn x (nthcdr y s))))
  ((induct (member-nth-firstn-induction i y s))))

(prove-lemma non-overlapping-requests-means (rewrite)
  (implies
   (non-overlapping-requests r)
   (non-overlapping-requests3 (car r) (cdr r))))

(prove-lemma non-overlapping-requests2-cdr (rewrite)
  (implies
   (non-overlapping-requests2 r1 r2)
   (non-overlapping-requests2 (cdr r1) (cdr r2))))

(prove-lemma non-overlapping-requests-cdr (rewrite)
  (implies
   (non-overlapping-requests r)
   (non-overlapping-requests (cdr r))))

(prove-lemma member-firstn-only-if-member (rewrite)
  (implies
   (not (member x l))
   (equal
    (member x (firstn n l))
    (and (equal x 0) (lessp (length l) n)))))

(prove-lemma member-nthcdr-only-if-member (rewrite)
  (implies
   (not (member x l))
   (not (member x (nthcdr n l)))))

(prove-lemma nth-replace-nth (rewrite)
  (equal
   (nth i (replace-nth j v l))
   (if (equal (fix i) (fix j)) v (nth i l))))

(prove-lemma member-x-firstn-cons-x (rewrite)
  (equal
   (member x (firstn n (cons x y)))
   (lessp 0 n)))

(defn cars-non-nil-litatoms (list)
  (if (listp list)
      (and
       (litatom (caar list))
       (not (equal (caar list) nil))
       (cars-non-nil-litatoms (cdr list)))
    t))

(prove-lemma swap-preserves-good-schedule nil
  (implies
   (and
    (non-overlapping-requests3 task-request-a r)
```

```
    (non-overlapping-requests3 task-request-b r)
    (cars-non-nil-litatoms r)
    (not (lessp i (cadr task-request-a)))
    (not (lessp j (cadr task-request-a)))
    (lessp i (caddr task-request-a))
    (lessp j (caddr task-request-a))
    (not (lessp i (cadr task-request-b)))
    (not (lessp j (cadr task-request-b)))
    (lessp i (caddr task-request-b))
    (lessp j (caddr task-request-b))
    (equal (nth i s) (car task-request-a))
    (equal (nth j s) (car task-request-b))
    (litatom (nth i s))
    (litatom (nth j s))
    (good-schedule s r))
   (good-schedule (swap i j s) r))
  ((induct (good-schedule s r))
   (disable-theory t)
   (enable swap non-overlapping-requests3 good-schedule occurrences-replace-nth
           firstn-replace-nth nth-firstn nth-nthcdr nth nthcdr firstn member-nth-firstn
           member-nthcdr-only-if-member member-firstn-only-if-member member-x-firstn-cons-x
           member-nth-firstn-nthcdr cars-non-nil-litatoms equal-occurrences-zero
           length-replace-nth length-firstn nth-replace-nth nthcdr-replace-nth)
   (enable-theory ground-zero task-abbr naturals)))

; return task request corresponding to task at time
(defn corresponding-request (task time r)
  (if (listp r)
      (if (and
           (equal (caar r) task)
           (not (lessp time (cadar r)))
           (lessp time (caddar r)))
          (car r)
        (corresponding-request task time (cdr r)))
    f))

(defn all-non-nil-corresponding (s r time)
  (if (lessp time (length s))
      (and
       (or
        (corresponding-request (nth time s) time r)
        (equal (nth time s) nil))
       (all-non-nil-corresponding s r (add1 time)))
    t)
  ((lessp (difference (length s) time))))

(prove-lemma non-overlapping-requests2-member (rewrite)
  (implies
   (and
    (non-overlapping-requests2 r1 r2)
    (sublistp r1 r2)
    (member e r1))
   (non-overlapping-requests3 e r2)))

(prove-lemma non-overlapping-requests3-member (rewrite)
  (implies
   (and
    (non-overlapping-requests r)
    (member e r))
   (non-overlapping-requests3 e r)))

(prove-lemma member-corresponding-request-nth (rewrite)
```

```
  (implies
    (corresponding-request task time r)
    (member (corresponding-request task time r) r)))


(defn all-litatoms (list)
  (if (listp list)
      (and
       (litatom (car list))
       (all-litatoms (cdr list)))
    t))

(prove-lemma litatom-nth (rewrite)
  (implies
   (all-litatoms s)
   (equal (litatom (nth i s)) (lessp i (length s)))))

(prove-lemma lessp-first-instance-s (rewrite)
  (equal
   (lessp (first-instance time task s) (length s))
   (listp s)))

(prove-lemma member-nthcdr-from-cdr (rewrite)
  (implies
   (and
    (not (member x (nthcdr n (cdr s))))
    (lessp n (length s)))
   (equal (member x (nthcdr n s))
          (equal x (nth n s)))))

(prove-lemma listp-nthcdr (rewrite)
  (equal
   (listp (nthcdr n x))
   (lessp n (length x))))

(prove-lemma car-nthcdr (rewrite)
  (equal
   (car (nthcdr n l))
   (if (lessp n (length l)) (nth n l) 0)))

(prove-lemma nth-first-instance-simple (rewrite)
  (equal
   (equal (nth (first-instance n x s) s) x)
   (or
    (member x (nthcdr n s))
    (equal (car s) x))))


(prove-lemma equal-occurrences-firstn-nthcdr (rewrite)
  (implies
   (lessp n1 n2)
   (equal
    (equal (occurrences x (firstn n1 (nthcdr n list)))
           (occurrences x (firstn n2 (nthcdr n list))))
    (not (member x (firstn (difference n2 n1) (nthcdr (plus n n1) list)))))))

(prove-lemma member-firstn-lessp (rewrite)
  (implies
   (and
    (member x (firstn n1 l))
    (not (lessp n2 n1)))
   (member x (firstn n2 l))))
```

```
(prove-lemma member-least-deadline (rewrite)
  (equal
   (member (least-deadline r) r)
   (listp r)))

(prove-lemma unfulfilled-task-later-in-good-schedule (rewrite)
  (implies
   (and
    (good-schedule s r)
    (all-litatoms s)
    (cars-non-nil-litatoms r)
    (member tr (unfulfilled element s (active-task-requests element r))))
   (member (car tr) (nthcdr element s))))

(prove-lemma car-corresponding-request (rewrite)
  (implies
   (corresponding-request task time r)
   (equal (car (corresponding-request task time r)) task)))

(prove-lemma active-task-has-later-deadline nil
  (implies
   (member tr (active-task-requests element r))
   (lessp element (caddr tr))))

(prove-lemma sublistp-unfulfilled (rewrite)
  (sublistp (unfulfilled e s r) r))

(prove-lemma least-deadline-has-later-deadline (rewrite)
  (implies
   (listp (unfulfilled element s (active-task-requests element r)))
   (equal
    (lessp
     element
     (caddr (least-deadline
      (unfulfilled element s (active-task-requests element r)))))
    t))
  ((use (active-task-has-later-deadline
         (tr (least-deadline
              (unfulfilled element s (active-task-requests element r)))))
        (member-sublistp
         (e (least-deadline
              (unfulfilled element s (active-task-requests element r))))
         (x (unfulfilled element s (active-task-requests element r)))
         (y (active-task-requests element r))))))


(prove-lemma active-task-hasnt-earlier-start nil
  (implies
   (member tr (active-task-requests element r))
   (not (lessp element (cadr tr)))))

(prove-lemma lessp-corresponding-request-deadline (rewrite)
  (implies
   (corresponding-request (nth element s) element r)
   (equal
    (lessp element (caddr (corresponding-request (nth element s) element r)))
    t)))

(prove-lemma least-deadline-hasnt-earlier-start (rewrite)
  (implies
   (listp (unfulfilled element s (active-task-requests element r)))
```

```
   (equal
    (lessp
     element
     (cadr (least-deadline
      (unfulfilled element s (active-task-requests element r)))))
    f))
  ((use (active-task-hasnt-earlier-start
         (tr (least-deadline
              (unfulfilled element s (active-task-requests element r)))))
        (member-sublistp
         (e (least-deadline
              (unfulfilled element s (active-task-requests element r))))
         (x (unfulfilled element s (active-task-requests element r)))
         (y (active-task-requests element r))))))

(prove-lemma lessp-corresponding-request-start (rewrite)
  (implies
   (corresponding-request x time r)
   (not (lessp time (cadr (corresponding-request x time r))))))

(prove-lemma lessp-corresponding-request-deadline-linear (rewrite)
  (implies
   (corresponding-request x time r)
   (lessp time (caddr (corresponding-request x time r)))))

(prove-lemma sublistp-active-task-requests (rewrite)
  (sublistp (active-task-requests time r) r))

(prove-lemma member-means-all-cars-not-litatoms (rewrite)
  (implies
   (and
    (member x r)
    (not (litatom (car x))))
   (not (cars-non-nil-litatoms r))))

(prove-lemma member-least-deadline-unfulfilled (rewrite)
  (implies
   (cars-non-nil-litatoms r)
   (equal
    (member
     (least-deadline (unfulfilled element s (active-task-requests element r)))
     r)
    (listp (unfulfilled element s (active-task-requests element r)))))
   ((use (member-sublistp
          (e (least-deadline
              (unfulfilled element s (active-task-requests element r))))
          (x (unfulfilled element s (active-task-requests element r)))
          (y (active-task-requests element r)))
         (member-sublistp
          (e (least-deadline
              (unfulfilled element s (active-task-requests element r))))
          (y r)
          (x (active-task-requests element r))))))

(prove-lemma not-numberp-corresponding (rewrite)
  (implies
   (not (numberp element))
   (equal (corresponding-request x element r)
          (corresponding-request x 0 r))))

(prove-lemma member-firstn-means-lessp-first-instance (rewrite)
  (implies
```

```
   (member v (firstn (difference n time) (nthcdr time s)))
   (equal (lessp (first-instance time v s) n) t)))


(prove-lemma first-instance-member-unfulfilled-not-past-deadline (rewrite)
  (implies
   (and
    (good-schedule s r)
    (member tr (unfulfilled element s (active-task-requests element r))))
   (equal
    (lessp (first-instance element (car tr) s)
           (caddr tr))
    t)))

(prove-lemma lessp-firstn-instance-time (rewrite)
  (implies
   (not (lessp time n))
  (equal
   (lessp (first-instance time v s) n)
   (and
    (not (member v (nthcdr time s)))
    (not (zerop n))))))

(prove-lemma first-instance-member-unfulfilled-not-before-start (rewrite)
  (implies
   (and
    (good-schedule s r)
    (cars-non-nil-litatoms r)
    (member tr (unfulfilled element s (active-task-requests element r))))
   (equal
    (lessp (first-instance element (car tr) s)
           (cadr tr))
    f)))

(prove-lemma member-corresponding-request (rewrite)
  (implies
   (and
    (good-schedule s r)
    (corresponding-request (nth element s) element r))
   (member
    (corresponding-request (nth element s) element r)
    (unfulfilled element s (active-task-requests element r)))))

(defn member-deadline-induct (x l)
  (if (listp l)
      (if (listp (cdr l))
          (if (lessp (caddar l) (caddadr l))
              (if (equal x (cadr l))
                  (member-deadline-induct (car l) (cons (car l) (cddr l)))
                (member-deadline-induct x (cons (car l) (cddr l))))
            (if (equal x (car l))
                (member-deadline-induct (cadr l) (cons (cadr l) (cddr l)))
              (member-deadline-induct x (cons (cadr l) (cddr l)))))
        t)
    t)
  ((lessp (length l))))


(prove-lemma member-deadline-not-less-than-least-deadline (rewrite)
  (implies
   (member x l)
   (equal (lessp (caddr x) (caddr (least-deadline l))) f))
```

```
  ((induct (member-deadline-induct x l))))

(prove-lemma first-instance-member-unfulfilled-not-past-deadline-better
  (rewrite)
  (implies
   (and
    (good-schedule s r)
    (member tr (unfulfilled element s (active-task-requests element r)))
    (not (lessp (caddr tr2) (caddr tr))))
   (equal
    (lessp (first-instance element (car tr) s)
           (caddr tr2))
    t))
  ((use (first-instance-member-unfulfilled-not-past-deadline))
   (disable first-instance-member-unfulfilled-not-past-deadline)))

(prove-lemma non-overlapping-requests3-simple (rewrite)
  (implies
   (and
    (cars-non-nil-litatoms r)
    (not (listp tr)))
   (non-overlapping-requests3 tr r)))


(prove-lemma swap-preserves-good-schedule-simple nil
  (implies
   (and
    (non-overlapping-requests3 task-request r)
    (cars-non-nil-litatoms r)
    (not (lessp i (cadr task-request)))
    (not (lessp j (cadr task-request)))
    (lessp i (caddr task-request))
    (lessp j (caddr task-request))
    (equal (nth j s) (car task-request))
    (equal (nth i s) nil)
    (litatom (nth j s))
    (good-schedule s r))
   (good-schedule (swap i j s) r))
  ((induct (good-schedule s r))
   (disable-theory t)
   (enable swap non-overlapping-requests3 good-schedule occurrences-replace-nth
           firstn-replace-nth nth-firstn nth-nthcdr nth nthcdr firstn member-nth-firstn
           member-nthcdr-only-if-member member-firstn-only-if-member member-x-firstn-cons-x
           member-nth-firstn-nthcdr cars-non-nil-litatoms equal-occurrences-zero
           length-replace-nth length-firstn nth-replace-nth nthcdr-replace-nth)
   (enable-theory ground-zero task-abbr naturals)))

(prove-lemma nth-too-big (rewrite)
  (implies
   (not (lessp i (length s)))
   (equal (nth i s) 0)))

(prove-lemma not-corresponding-request-means-nil (rewrite)
  (implies
   (and
    (all-non-nil-corresponding s r n)
    (not (corresponding-request (nth i s) i r))
    (not (lessp i n)))
   (equal (nth i s) (if (lessp i (length s)) nil 0))))


;; takes 1/2 hr to process without disable-theory t
```

```
(prove-lemma make-element-edf-preserves-good-schedule (rewrite)
  (implies
   (and
    (lessp element (length s))
    (non-overlapping-requests r)
    (cars-non-nil-litatoms r)
    (all-litatoms s)
    (all-non-nil-corresponding s r 0)
    (good-schedule s r))
   (good-schedule (make-element-edf s r element) r))
  ((disable-theory t)
   (enable-theory ground-zero task-abbr)
   (enable nth length good-schedule first-instance make-element-edf non-overlapping-requests
           swap-commutative cars-non-nil-litatoms all-non-nil-corresponding
           non-overlapping-requests3-member member-corresponding-request-nth all-litatoms
           litatom-nth lessp-first-instance-s nth-first-instance-simple member-least-deadline
           unfulfilled-task-later-in-good-schedule car-corresponding-request
           least-deadline-has-later-deadline least-deadline-hasnt-earlier-start
           lessp-corresponding-request-start lessp-corresponding-request-deadline-linear
           member-least-deadline-unfulfilled lessp-firstn-instance-time
           first-instance-member-unfulfilled-not-before-start member-corresponding-request
           member-deadline-not-less-than-least-deadline
           first-instance-member-unfulfilled-not-past-deadline-better
           non-overlapping-requests3-simple nth-too-big not-corresponding-request-means-nil)
   (use
    (swap-preserves-good-schedule
     (task-request-a
      (if (corresponding-request (nth element s) element r)
          (corresponding-request (nth element s) element r)
        nil))
     (task-request-b
      (least-deadline
       (unfulfilled element s (active-task-requests element r))))
     (time element)
     (i element)
     (j
      (first-instance  element
       (car (least-deadline
             (unfulfilled element s (active-task-requests element r))))
       s)))
    (swap-preserves-good-schedule-simple
     (task-request
      (least-deadline
       (unfulfilled element s (active-task-requests element r))))
     (time element)
     (i element)
     (j
      (first-instance  element
       (car (least-deadline
             (unfulfilled element s (active-task-requests element r))))
       s))))))

(prove-lemma all-litatoms-replace-nth (rewrite)
  (equal
   (all-litatoms (replace-nth n v x))
   (and
    (litatom v)
    (not (lessp (length x) n))
    (all-litatoms (firstn n x))
    (all-litatoms (nthcdr (add1 n) x)))))

(prove-lemma all-litatoms-nthcdr (rewrite)
```

```
  (implies
   (all-litatoms s)
   (all-litatoms (nthcdr n s))))

(prove-lemma all-litatoms-repeat (rewrite)
  (equal
   (all-litatoms (repeat n v))
   (or
    (zerop n)
    (litatom v))))

(prove-lemma all-litatoms-firstn (rewrite)
  (implies
   (all-litatoms s)
   (equal (all-litatoms (firstn n s)) (not (lessp (length s) n)))))

(prove-lemma make-element-edf-preserves-all-litatoms (rewrite)
  (implies
   (and
    (lessp element (length s))
    (non-overlapping-requests r)
    (cars-non-nil-litatoms r)
    (all-litatoms s)
    (all-non-nil-corresponding s r 0)
    (good-schedule s r))
   (all-litatoms (make-element-edf s r element)))
  ((disable non-overlapping-requests)))

(prove-lemma equal-lessp-sub1x-y-x-y (rewrite)
  (equal
   (equal (lessp (sub1 x) y) (lessp x y))
   (or
    (zerop x)
    (not (equal (fix x) (fix y))))))


(prove-lemma all-non-nil-corresponding-replace-simple (rewrite)
  (implies
   (and
    (all-non-nil-corresponding s r n)
    (lessp n1 (length s)))
   (equal
    (all-non-nil-corresponding (replace-nth n1 v s) r n)
    (or (corresponding-request v n1 r) (equal v nil) (lessp n1 n)))))

(prove-lemma car-replace-nth (rewrite)
  (equal (car (replace-nth n v s)) (if (zerop n) v (car s)))
  ((expand (replace-nth 1 v s))))

(prove-lemma not-corresponding-request-means (rewrite)
  (implies
   (and
    (not (corresponding-request v1 n1 r))
    (not (equal v1 nil))
    (equal (nth n1 s) v1)
    (lessp n1 (length s))
    (not (lessp n1 n)))
   (not (all-non-nil-corresponding s r n))))

(prove-lemma all-non-nil-corresponding-cons (rewrite)
  (implies
   (all-non-nil-corresponding (cons a b) r n)
```

```
   (equal
    (all-non-nil-corresponding (cons x b) r n)
    (or (equal x nil) (corresponding-request x 0 r) (not (zerop n))))))


(prove-lemma all-non-nil-corresponding-replace-replace (rewrite)
  (implies
   (and
    (all-non-nil-corresponding s r n)
    (lessp n1 (length s))
    (lessp n2 (length s)))
   (equal
    (all-non-nil-corresponding (replace-nth n2 v2 (replace-nth n1 v1 s)) r n)
    (and
     (or (corresponding-request v1 n1 r) (equal v1 nil) (lessp n1 n)
         (equal (fix n1) (fix n2)))
     (or (corresponding-request v2 n2 r) (equal v2 nil) (lessp n2 n))))))

(prove-lemma nth-first-instance (rewrite)
  (implies
   (member v (nthcdr time s))
   (equal (nth (first-instance time v s) s) v)))

(prove-lemma car-corresponding-request-better (rewrite)
  (equal
   (car (corresponding-request v n r))
   (if (corresponding-request v n r) v 0)))

(prove-lemma equivalent-corresponding-requests nil
  (implies
   (and
    (non-overlapping-requests3 (corresponding-request v n r) r)
    (not (lessp n1 (cadr (corresponding-request v n r))))
    (lessp n1 (caddr (corresponding-request v n r))))
   (equal (corresponding-request v n1 r) (corresponding-request v n r))))


(prove-lemma lessp-n-1 (rewrite)
  (equal
   (lessp n 1)
   (zerop n)))

(prove-lemma member-corresponding-request-simplify (rewrite)
  (implies
   (cars-non-nil-litatoms r)
   (iff
    (member (corresponding-request v n r) r)
    (corresponding-request v n r))))

(prove-lemma member-corresponding-request2 (rewrite)
  (implies
   (and
    (all-non-nil-corresponding s r n1)
    (not (lessp n n1))
    (lessp n (length s))
    (cars-non-nil-litatoms r)
    (not (equal (nth n s) nil)))
   (member (corresponding-request (nth n s) n r) r)))


;;; the best theorem would be about make-element-edf's preserving
;;; all-non-nil-corresponding.  But we'll punt on that for now,
```

```
;;; and use the fact that periodic request schedules are "full"
;;; and thus all reasonable schedules have corresponding requests

(defn all-nils-or-cars (s r)
  (if (listp s)
      (and
       (or (equal (car s) nil) (assoc (car s) r))
       (all-nils-or-cars (cdr s) r))
    t))

(prove-lemma corresponding-request-append (rewrite)
  (iff
   (corresponding-request v n (append r1 r2))
   (or
   (corresponding-request v n r1)
   (corresponding-request v n r2))))

(prove-lemma corresponding-request-different-name (rewrite)
  (implies
   (not (equal (car pt) tk))
   (not (corresponding-request tk n (periodic-task-requests pt n1 n2)))))

(prove-lemma corresponding-request-periodic-task (rewrite)
  (implies
   (and
    (periodic-taskp pt)
    (equal (remainder n1 (cadr pt)) 0)
    (lessp n n2)
    (not (lessp n n1)))
   (iff (corresponding-request tk n (periodic-task-requests pt n1 n2))
        (equal (car pt) tk))))

(prove-lemma all-nils-or-cars-nlistp (rewrite)
  (implies
   (not (listp x))
   (equal (all-nils-or-cars l x) (equal (plist l) (repeat (length l) nil)))))

(prove-lemma assoc-nth-pts (rewrite)
  (implies
   (and
    (lessp n (length s))
    (all-nils-or-cars s pts)
    (not (equal (nth n s) nil)))
   (assoc (nth n s) pts)))

(prove-lemma corresponding-request-periodic-tasks (rewrite)
  (implies
   (and
    (assoc v pts)
    (periodic-tasksp pts)
    (equal (remainder n1 (cadr (assoc v pts))) 0)
    (lessp n n2)
    (not (lessp n n1)))
   (corresponding-request v n (periodic-tasks-requests pts n1 n2)))
  ((induct (length pts))))

(prove-lemma all-nils-or-cars-replace-nth (rewrite)
  (implies
   (and
    (all-nils-or-cars s r)
    (lessp n (length s)))
   (equal
```

```
    (all-nils-or-cars (replace-nth n v s) r)
    (or
     (equal v nil)
     (assoc v r)))))

(prove-lemma all-non-nil-corresponding-periodic-requests (rewrite)
  (implies
   (and
    (all-nils-or-cars s pts)
    (periodic-tasksp pts)
    (equal (fix n1) (length s)))
   (all-non-nil-corresponding s (periodic-tasks-requests pts 0 n1) n)))

(defn double-sub1-cdr-induct (n1 n2 s)
  (if (zerop n2) t
    (double-sub1-cdr-induct (sub1 n1) (sub1 n2) (cdr s))))

(prove-lemma all-nils-or-cars-replace-nth-replace-nth (rewrite)
  (implies
   (and
    (all-nils-or-cars s r)
    (lessp n (length s))
    (lessp n2 (length s)))
   (equal
    (all-nils-or-cars (replace-nth n v (replace-nth n2 v2 s)) r)
    (and
     (or (equal v nil) (assoc v r))
     (or (equal v2 nil) (assoc v2 r) (equal (fix n) (fix n2))))))
  ((induct (double-sub1-cdr-induct n n2 s))))

(prove-lemma lessp-0-length-means-listp (rewrite)
  (implies
   (lessp n (length l))
   (listp l)))

(prove-lemma all-nils-or-cars-make-element-edf (rewrite)
  (implies
   (and
    (all-nils-or-cars s pts)
    (listp s)
    (lessp n (length s)))
   (all-nils-or-cars (make-element-edf s pts1 n) pts)))

(prove-lemma make-schedule-edf-preserves-good-schedule (rewrite)
  (implies
   (and
    (non-overlapping-requests r)
    (cars-non-nil-litatoms r)
    (all-litatoms s)
    (all-non-nil-corresponding s r 0)
    (all-nils-or-cars s pts)        ; not needed but useful
    (good-schedule s r)
    (equal r (periodic-tasks-requests pts 0 (length s)))
    (periodic-tasksp pts))
   (good-schedule (make-schedule-edf s r n) r))
  ((induct (make-schedule-edf s r n))
   (disable make-element-edf non-overlapping-requests)))

(prove-lemma non-overlapping-requests2-append (rewrite)
  (equal
   (non-overlapping-requests2 (append a b) r)
   (and
```

```
    (non-overlapping-requests2 a r)
    (non-overlapping-requests2 b r))))

(prove-lemma all-nils-or-cars-plist (rewrite)
  (equal
   (all-nils-or-cars (plist a) pts)
   (all-nils-or-cars a pts)))

(prove-lemma all-nils-or-cars-append (rewrite)
  (equal
   (all-nils-or-cars (append a b) pts)
   (and
    (all-nils-or-cars a pts)
    (all-nils-or-cars b pts))))

(prove-lemma all-nils-or-cars-repeat-list (rewrite)
  (equal
   (all-nils-or-cars (repeat-list l n) pts)
   (or
    (zerop n)
    (all-nils-or-cars l pts))))

(prove-lemma all-nils-or-cars-make-length (rewrite)
  (equal
   (all-nils-or-cars (make-length n l nil) pts)
   (if (lessp n (length l))
     (all-nils-or-cars (firstn n l) pts)
     (all-nils-or-cars l pts))))

(prove-lemma all-nils-or-cars-repeat (rewrite)
  (equal
   (all-nils-or-cars (repeat n v) pts)
   (or (equal v nil) (assoc v pts) (zerop n))))

(prove-lemma all-nils-or-cars-substring-schedule (rewrite)
  (implies
   (cars-non-nil-litatoms pts)
   (all-nils-or-cars (substring-schedule pts n) pts)))

(prove-lemma all-nils-or-cars-firstn (rewrite)
  (implies
   (and
    (all-nils-or-cars l pts)
    (lessp n (length l)))
   (all-nils-or-cars (firstn n l) pts)))

(prove-lemma all-nils-or-cars-make-simple-schedule (rewrite)
  (implies
   (cars-non-nil-litatoms pts)
   (all-nils-or-cars (make-simple-schedule pts bigp n) pts)))

(prove-lemma length-make-simple-schedule (rewrite)
  (implies
   (equal (remainder n bigp) 0)
   (equal (length (make-simple-schedule pts bigp n)) (fix n))))

(prove-lemma periodic-tasksp-means-cars-non-nil-litatoms (rewrite)
  (implies
   (periodic-tasksp pts)
   (cars-non-nil-litatoms pts)))

(prove-lemma all-litatoms-append (rewrite)
```

```
  (equal
   (all-litatoms (append a b))
   (and
    (all-litatoms a)
    (all-litatoms b))))

(prove-lemma all-litatoms-repeat-list (rewrite)
  (equal
   (all-litatoms (repeat-list l n))
   (or
    (zerop n)
    (all-litatoms l))))

(prove-lemma all-litatoms-make-length (rewrite)
  (equal
   (all-litatoms (make-length n l v))
   (if (lessp (length l) n)
       (and
        (all-litatoms l)
        (litatom v))
     (all-litatoms (firstn n l)))))

(prove-lemma all-litatoms-substring-schedule (rewrite)
  (implies
   (cars-non-nil-litatoms pts)
   (all-litatoms (substring-schedule pts bigp))))

(prove-lemma all-litatoms-make-simple-schedule (rewrite)
  (implies
   (cars-non-nil-litatoms pts)
   (all-litatoms (make-simple-schedule pts bigp n))))

(prove-lemma cars-non-nil-litatoms-periodic-tasks (rewrite)
  (implies
   (cars-non-nil-litatoms pts)
   (cars-non-nil-litatoms (periodic-tasks-requests pts n1 n2))))

(defn value-all-cars (v list)
  (if (listp list)
      (and
       (equal (caar list) v)
       (value-all-cars v (cdr list)))
    t))

(prove-lemma non-overlapping-requests3-append (rewrite)
  (equal
   (non-overlapping-requests3 v (append r1 r2))
   (and
    (non-overlapping-requests3 v r1)
    (non-overlapping-requests3 v r2))))

(prove-lemma value-all-cars-periodic-task-requests (rewrite)
  (equal
   (value-all-cars x (periodic-task-requests req n1 n2))
   (or (not (listp (periodic-task-requests req n1 n2)))
       (equal x (car req)))))

(prove-lemma non-overlapping-requests2-value-all-cars (rewrite)
  (implies
   (and
    (value-all-cars v list)
    (not (assoc v r))
```

```
    (cars-non-nil-litatoms r))
   (equal (non-overlapping-requests2 r (append list r2))
          (non-overlapping-requests2 r r2))))

(prove-lemma assoc-append (rewrite)
  (implies
   (not (equal v 0))
  (iff
   (assoc v (append x y))
   (or
    (assoc v x)
    (assoc v y)))))


(prove-lemma assoc-periodic-task-requests (rewrite)
  (implies
    (not (equal v (car req)))
    (not (assoc v (periodic-task-requests req n1 n2)))))


(prove-lemma assoc-periodic-tasks-requests (rewrite)
  (implies
   (and
    (not (equal v 0))
    (not (assoc v list)))
   (not (assoc v (periodic-tasks-requests list n1 n2)))))


(prove-lemma non-overlapping-requests2-append-arg2 (rewrite)
  (implies
   (and
    (not (assoc (car req) list))
    (not (equal (car req) 0))
    (cars-non-nil-litatoms list))
   (equal
    (non-overlapping-requests2
     (periodic-tasks-requests list n1 n2)
     (append (periodic-task-requests req n3 n4) y))
    (non-overlapping-requests2 (periodic-tasks-requests list n1 n2) y)))
  ((use (non-overlapping-requests2-value-all-cars
         (v (car req)) (list (periodic-task-requests req n3 n4))
         (r (periodic-tasks-requests list n1 n2))
         (r2 y)))))

(prove-lemma non-overlapping-requests3-periodic-task (rewrite)
  (implies
   (not (lessp n1 end))
   (non-overlapping-requests3 (list tk start end duration)
                              (periodic-task-requests pt n1 n2))))

(prove-lemma non-overlapping-requests3-task-name-difference (rewrite)
  (implies
   (not (equal (car pt) (car req)))
   (non-overlapping-requests3 req (periodic-task-requests pt n1 n2))))

(prove-lemma non-overlapping-requests3-name-difference (rewrite)
  (implies
   (and
    (litatom (car req))
    (not (assoc (car req) list)))
   (non-overlapping-requests3 req (periodic-tasks-requests list n1 n2))))
```

```
(prove-lemma non-overlapping-requests2-cons-too-big (rewrite)
  (implies
   (not (lessp n1 (caddr req)))
  (equal
   (non-overlapping-requests2 (periodic-task-requests tr n1 n2) (cons req r))
   (non-overlapping-requests2 (periodic-task-requests tr n1 n2) r))))

(prove-lemma non-overlapping-requests2-periodic-task (rewrite)
  (implies
   (not (assoc (car req) x))
  (non-overlapping-requests2
   (periodic-task-requests req n1 n2)
   (append
    (periodic-task-requests req n1 n2)
    (periodic-tasks-requests x n3 n4)))))

(prove-lemma non-overlapping-requests2-periodic-tasks (rewrite)
  (implies
   (periodic-tasksp pts)
   (non-overlapping-requests2
    (periodic-tasks-requests pts 0 n)
    (periodic-tasks-requests pts 0 n))))

(prove-lemma non-overlapping-requests-periodic-tasks-requests (rewrite)
  (implies
   (periodic-tasksp pts)
   (non-overlapping-requests (periodic-tasks-requests pts 0 n))))


(prove-lemma edf-schedule-good-for-expanded (rewrite)
  (implies
   (and
    (not (lessp bigp (length (substring-schedule pts bigp))))
    (periodic-tasksp pts)
    (expanded-tasksp pts bigp)
    (lessp 0 bigp)
    (zerop (remainder n bigp))
    (zerop (remainder n (big-period pts))))
    (good-schedule
     (make-schedule-edf (make-simple-schedule pts bigp n)
                        (periodic-tasks-requests pts 0 n)
                        0)
     (periodic-tasks-requests pts 0 n)))
  ((disable make-schedule-edf make-simple-schedule non-overlapping-requests)
   (use (make-schedule-edf-preserves-good-schedule
         (s (make-simple-schedule pts bigp n))
         (r (periodic-tasks-requests pts 0 n))
         (n 0)))))

(defn every-nth (n list)
  (if (zerop n) nil
    (if (listp list)
        (cons (car list)
              (every-nth n (nthcdr n list)))
      nil))
  ((lessp (length list))))

(defn expand-tasks-requests (tasks-requests n)
  (if (listp tasks-requests)
      (cons (list (caar tasks-requests)
                  (times n (cadar tasks-requests))
                  (times n (caddar tasks-requests))
```

```
                  (times n (cadddar tasks-requests)))
            (expand-tasks-requests (cdr tasks-requests) n))
    nil))

(prove-lemma expand-tasks-requests-append (rewrite)
  (equal
   (expand-tasks-requests (append a b) n)
   (append (expand-tasks-requests a n) (expand-tasks-requests b n))))

(prove-lemma listp-append (rewrite)
  (equal
   (listp (append a b))
   (or (listp a) (listp b))))

(prove-lemma listp-expand-tasks-requests (rewrite)
  (equal
   (listp (expand-tasks-requests l n))
   (listp l)))

(prove-lemma listp-task-requests (rewrite)
  (equal
   (listp (periodic-task-requests task n1 n2))
   (and
    (lessp n1 n2)
    (periodic-taskp task))))

(prove-lemma length-expand-tasks-requests (rewrite)
  (equal
   (length (expand-tasks-requests l n))
   (length l)))

(prove-lemma different-lengths-mean-different nil
  (implies
   (equal x y)
   (equal (length x) (length y))))

(prove-lemma equal-nthcdr-x-x (rewrite)
  (equal
   (equal (nthcdr n x) x)
   (or
    (zerop n)
    (equal x 0)))
  ((use (different-lengths-mean-different (x (nthcdr n x)) (y x)))))

(defn length-periodic-task-request-induct (n1 period bigp n2)
  (if (lessp 0 (times bigp period))
      (if (lessp n1 n2)
          (length-periodic-task-request-induct (plus n1 (times bigp period))
                                               period bigp n2)
        t)
    t)
  ((lessp (difference n2 n1))))

(prove-lemma length-periodic-task-requests (rewrite)
  (implies
   (and
    (lessp 0 bigp)
    (lessp 0 period)
    (equal (remainder n1 bigp) 0)
    (equal (remainder n2 bigp) 0))
   (equal
    (length (periodic-task-requests (list name (times bigp period)
```

```
                                        duration) n1 n2))
    (length (periodic-task-requests (list name period duration)
                                    (quotient n1 bigp) (quotient n2 bigp)))))
  ((induct (length-periodic-task-request-induct n1 period bigp n2))))

(prove-lemma plist-expand-tasks-requests (rewrite)
  (equal
   (plist (expand-tasks-requests task n))
   (expand-tasks-requests task n)))

(prove-lemma equal-plist-nil (rewrite)
  (equal
   (equal (plist l) nil)
   (nlistp l)))

(prove-lemma length-cons (rewrite)
  (equal (length (cons a b)) (add1 (length b))))

(prove-lemma cons-append-hack (rewrite)
  (not (equal (cons a (append b c)) c))
  ((use (different-lengths-mean-different (x c) (y (cons a (append b c)))))))

(prove-lemma equal-append-b-append-b (rewrite)
  (equal
   (equal (append x b) (append y b))
   (equal (plist x) (plist y)))
  ((induct (double-cdr-induction x y))))

(prove-lemma plist-periodic-task-requests (rewrite)
  (equal
   (plist (periodic-task-requests task n1 n2))
   (periodic-task-requests task n1 n2)))

(prove-lemma lessp-equal-times-x-a-x (rewrite)
  (equal (lessp (times a b) a)
         (and
          (not (zerop a))
          (zerop b))))

(prove-lemma periodic-task-requests-expand (rewrite)
  (implies
   (and
    (periodic-taskp pt)
    (equal (remainder n1 bigp) 0)
    (equal (remainder n2 bigp) 0)
    (equal (remainder (cadr pt) bigp) 0)
    (equal (remainder (caddr pt) bigp) 0)
    (numberp n1))
   (equal
    (periodic-task-requests pt n1 n2)
    (expand-tasks-requests
     (periodic-task-requests (cons (car pt)
                                   (cons (quotient (cadr pt) bigp)
                                         (cons (quotient (caddr pt) bigp)
                                               (cdddr pt))))
                             (quotient n1 bigp) (quotient n2 bigp))
     bigp)))
  ((induct (periodic-task-requests pt n1 n2))))


(prove-lemma periodic-tasks-requests-expand (rewrite)
  (implies
```

```
   (and
    (equal (remainder n1 bigp) 0)
    (equal (remainder n2 bigp) 0)
    (periodic-tasksp pts)
    (numberp n1))
   (equal
    (periodic-tasks-requests (expand-tasks pts bigp) n1 n2)
    (expand-tasks-requests
     (periodic-tasks-requests pts (quotient n1 bigp) (quotient n2 bigp))
     bigp)))
  ((induct (length pts))
   (disable equal-append)))

(defn edf (lengthschedule r)
  (if (zerop lengthschedule) nil
    (let ((s (edf (sub1 lengthschedule) r)))
      (let ((unfulfilled
             (unfulfilled (sub1 lengthschedule) s
                          (active-task-requests (sub1 lengthschedule) r))))
        (if (listp unfulfilled)
            (append s (list (name (least-deadline unfulfilled))))
          (append s (list nil)))))))

(prove-lemma length-edf-simple (rewrite)
  (equal (length (edf n r)) (fix n)))

(defn valid-requests (reqs)
  (if (listp reqs)
      (and
       (litatom (caar reqs))
       (not (equal (caar reqs) nil))
       (numberp (cadar reqs))
       (numberp (caddar reqs))
       (lessp 0 (cadddar reqs))
       (equal (cddddar reqs) nil)
       (valid-requests (cdr reqs)))
    (equal reqs nil)))

(prove-lemma valid-requests-periodic-task (rewrite)
   (equal
    (valid-requests (periodic-task-requests pt n1 n2))
    (or
     (numberp n1)
     (not (lessp n1 n2))
     (not (periodic-taskp pt)))))

(prove-lemma valid-requests-append (rewrite)
  (implies
   (equal (plist a) a)
   (equal
    (valid-requests (append a b))
    (and
     (valid-requests a)
     (valid-requests b)))))

(prove-lemma valid-requests-periodic-tasks (rewrite)
   (equal
    (valid-requests (periodic-tasks-requests pts n1 n2))
    (or
     (numberp n1)
     (not (lessp n1 n2))
     (not (periodic-tasksp pts))
```

```
    (not (listp pts)))))

(prove-lemma listp-active-task-requests-0 (rewrite)
  (implies
   (lessp n n2)
   (equal
    (listp (active-task-requests n (periodic-task-requests pt n1 n2)))
    (and (periodic-taskp pt) (not (lessp n n1)))))
  ((induct (periodic-task-requests pt n1 n2))))

(prove-lemma active-task-requests-append (rewrite)
  (equal
   (active-task-requests n (append r1 r2))
   (append
    (active-task-requests n r1)
    (active-task-requests n r2))))

(prove-lemma listp-active-task-requests-0-multiple (rewrite)
  (implies
   (lessp n n2)
   (equal
    (listp (active-task-requests n (periodic-tasks-requests pts n1 n2)))
    (and
     (periodic-tasksp pts)
     (not (lessp n n1))
     (listp pts))))
  ((induct (periodic-tasks-requests pts n1 n2))))

(prove-lemma unfulfilled-0 (rewrite)
  (implies
   (valid-requests r)
   (equal (unfulfilled 0 s r) r)))

(prove-lemma valid-requests-active-task-requests (rewrite)
  (implies
   (valid-requests r)
   (valid-requests (active-task-requests n r))))

(prove-lemma lessp-0-length-means (rewrite)
  (equal (lessp 0 (length l)) (listp l)))

(prove-lemma listp-repeat (rewrite)
  (equal (listp (repeat n v)) (not (zerop n))))

(prove-lemma length-repeat (rewrite)
  (equal (length (repeat n v)) (fix n)))

(prove-lemma plist-append (rewrite)
  (equal (plist (append x y)) (append x (plist y))))

(prove-lemma length-plist (rewrite)
  (equal (length (plist x)) (length x)))

(prove-lemma listp-plist (rewrite)
  (equal (listp (plist x)) (listp x)))

(prove-lemma plist-edf-simple (rewrite)
  (equal (plist (edf n r)) (edf n r)))


(prove-lemma firstn-repeat (rewrite)
  (equal
```

```
   (firstn n1 (repeat n2 v))
   (if (lessp n2 n1)
       (append (repeat n2 v) (repeat (difference n1 n2) 0))
     (repeat n1 v))))

(prove-lemma edf-simple-nlistp  (rewrite)
  (implies
   (not (listp r))
   (equal (edf n r) (repeat n nil)))
  ((expand (edf 1 r))))

(prove-lemma make-schedule-edf-nlistp (rewrite)
  (implies
   (not (listp r))
   (equal (make-schedule-edf s r n) s)))

(prove-lemma replace-nth-nlistp (rewrite)
  (implies
   (nlistp l)
   (equal (replace-nth n v l) (append (repeat n 0) (cons v 0)))))

(prove-lemma plist-replace-nth2 (rewrite)
  (equal (equal (plist (replace-nth n v s)) (replace-nth n v s))
         (and
          (equal (plist s) s)
          (lessp n (length s)))))

(prove-lemma plist-swap (rewrite)
  (equal
   (equal (plist (swap n1 n2 s)) (swap n1 n2 s))
   (and (equal (plist s) s) (lessp n1 (length s)) (lessp n2 (length s)))))

(prove-lemma plist-make-element-edf (rewrite)
  (implies
   (equal (plist s) s)
   (equal (plist (make-element-edf s r n)) (make-element-edf s r n)))
  ((disable swap)))

(prove-lemma length-make-schedule-edf (rewrite)
  (equal (length (make-schedule-edf s r n)) (length s)))

(prove-lemma equal-nthcdr-nthcdr-from-nthcdr-plus1 (rewrite)
  (implies
   (and
    (lessp n (length s1))
    (lessp n (length s2))
    (equal (nthcdr n (cdr s1)) (nthcdr n (cdr s2))))
   (equal
    (equal (nthcdr n s1) (nthcdr n s2))
    (equal (nth n s1) (nth n s2)))))

(prove-lemma lessp-first-instance2 (rewrite)
  (implies
   (first-instance n2 v l)
   (not (lessp (first-instance n2 v l) n2))))

(prove-lemma nth-make-schedule-edf-simple (rewrite)
  (implies
   (lessp n1 n2)
   (equal (nth n1 (make-schedule-edf s r n2)) (nth n1 s))))

(prove-lemma car-append (rewrite)
```

```
  (equal
   (car (append x y))
   (if (listp x) (car x) (car y))))

(prove-lemma listp-edf-simple (rewrite)
  (equal (listp (edf n r)) (not (zerop n))))

(prove-lemma unfulfilled-schedule-first-part-only (rewrite)
  (implies
   (lessp n (length s))
   (equal (unfulfilled n s r) (unfulfilled n (firstn n s) r))))

(prove-lemma firstn-n-edf-simple-n nil
  (implies
   (equal (fix n1) (fix n2))
   (equal (firstn n1 (edf n2 r)) (edf n2 r))))

(prove-lemma firstn-edf-simple-regular nil
  (implies
   (lessp n1 n2)
   (equal (firstn n1 (edf n2 r)) (edf n1 r)))
  ((induct (edf n2 r))))

(prove-lemma firstn-edf-simple-help nil
  (implies
   (not (lessp n2 n1))
   (equal (firstn n1 (edf n2 r)) (edf n1 r)))
  ((use (firstn-edf-simple-regular) (firstn-n-edf-simple-n))))

(prove-lemma firstn-edf-simple (rewrite)
  (equal
   (firstn n1 (edf n2 r))
   (if (lessp n2 n1)
       (append (edf n2 r) (repeat (difference n1 n2) 0))
     (edf n1 r)))
  ((use (firstn-too-big (x (edf n2 r)) (n n1))
        (firstn-edf-simple-help))))

(prove-lemma nth-append (rewrite)
  (equal (nth n (append x y))
         (if (lessp n (length x))
             (nth n x)
           (nth (difference n (length x)) y))))

; dangerous - must use in special rules
(prove-lemma nth-edf-simple-simpler nil
  (implies
   (lessp n n2)
   (equal (nth n (edf n2 r)) (nth n (edf (add1 n) r))))
  ((induct (edf n2 r))))

(prove-lemma active-task-requests-nnumberp (rewrite)
  (implies
   (not (numberp n))
   (equal (active-task-requests n r) (active-task-requests 0 r))))

(prove-lemma unfulfilled-nnumberp (rewrite)
  (implies
   (not (numberp n))
   (equal (unfulfilled n s r) (unfulfilled 0 s r))))

(prove-lemma nth-edf-simple (rewrite)
```

```
  (equal (nth n (edf n2 r))
         (if (lessp n n2)
             (if (listp (unfulfilled n (edf n r)
                                       (active-task-requests n r)))
                 (car (least-deadline
                       (unfulfilled n (edf n r)
                                      (active-task-requests n r))))
               nil)
           0))
  ((use (nth-edf-simple-simpler))
   (expand (edf (add1 n) r))
   (enable edf length-edf-simple nth-too-big nth-append nth unfulfilled-nnumberp
           active-task-requests-nnumberp)
   (disable-theory t)
   (enable-theory ground-zero task-abbr naturals)))

(prove-lemma equal-cdr-cdr-means (rewrite)
  (implies
   (and
    (equal (cdr x) (cdr y))
    (listp x)
    (listp y))
   (equal
    (equal x y)
    (equal (car x) (car y)))))

(disable equal-cdr-cdr-means)

(prove-lemma first-instance-same-as-member (rewrite)
  (implies
   n
  (iff
   (first-instance n v s)
   (member v (nthcdr n s)))))

(prove-lemma nth-make-element-edf (rewrite)
  (implies
   n
  (equal
   (nth n (make-element-edf s r n))
   (if (and (listp (unfulfilled n s (active-task-requests n r)))
            (first-instance n (car (least-deadline
                                    (unfulfilled n s
                                                 (active-task-requests n r))))
                            s))
       (car (least-deadline
             (unfulfilled n s (active-task-requests n r))))
     (nth n s)))))

(prove-lemma unfulfilled-append (rewrite)
  (equal
   (unfulfilled n s (append r1 r2))
   (append (unfulfilled n s r1) (unfulfilled n s r2))))

(defn no-unfulfilled-active-task-induct (pts oldpts)
  (if (nlistp pts) t
    (no-unfulfilled-active-task-induct (cdr pts)
                                       (append oldpts (list (car pts))))))

(prove-lemma periodic-tasksp-append-car (rewrite)
  (implies
   (and
```

```
    (periodic-tasksp (append x y))
    (listp y))
   (periodic-tasksp (append x (list (car y))))))

(prove-lemma assoc-plist (rewrite)
  (equal (assoc v (plist l)) (assoc v l)))

(prove-lemma all-nils-or-cars-plist2 (rewrite)
  (equal (all-nils-or-cars l1 (plist l2))
         (all-nils-or-cars l1 l2)))

(prove-lemma assoc-append-simple (rewrite)
  (implies
   (and
    (not (assoc v l))
    (periodic-tasksp l))
   (equal (assoc v (append l l2)) (assoc v l2))))

(prove-lemma listp-unfulfilled-if-schedule-contains (rewrite)
  (implies
   (and
    (periodic-taskp pt)
    (good-schedule s (periodic-task-requests pt n1 n2))
    (equal (nth n s) (car pt))
    (lessp n n2)
    (not (lessp n n1)))
   (listp (unfulfilled
           n
           (firstn n s)
           (active-task-requests n (periodic-task-requests pt n1 n2))))))

(prove-lemma no-unfulfilled-active-task-if-nil-help nil
  (implies
   (and
    (not (equal (nth n s) nil))
    (periodic-tasksp (append oldpts pts))
    (periodic-tasksp oldpts)  ; speeds proof
    (periodic-tasksp pts)     ; speeds proof
    (good-schedule s (periodic-tasks-requests pts n1 n2))
    (all-nils-or-cars s (append oldpts pts))
    (lessp n n2)
    (lessp n (length s))
    (not (lessp n n1)))
   (or
    (listp
     (unfulfilled
      n
      (firstn n s)
      (active-task-requests n (periodic-tasks-requests pts n1 n2))))
    (assoc (nth n s) oldpts)))
  ((induct (no-unfulfilled-active-task-induct pts oldpts))))

(prove-lemma no-unfulfilled-active-task-if-nil (rewrite)
  (implies
   (and
    (not (equal (nth n s) nil))
    (periodic-tasksp pts)
    (good-schedule s (periodic-tasks-requests pts n1 n2))
    (all-nils-or-cars s pts)
    (lessp n n2)
    (lessp n (length s))
    (not (lessp n n1)))
```

```
   (listp
    (unfulfilled
     n
     (firstn n s)
     (active-task-requests n (periodic-tasks-requests pts n1 n2)))))
  ((use (no-unfulfilled-active-task-if-nil-help (oldpts nil)))))

(prove-lemma replace-nth-first-instance-nnumberp (rewrite)
  (implies
   (not (numberp n))
   (equal (replace-nth (first-instance n v s) v2 l)
          (replace-nth (first-instance 0 v s) v2 l))))

(prove-lemma make-element-nnumberp (rewrite)
  (implies
   (not (numberp n))
   (equal (make-element-edf s r n)
          (make-element-edf s r 0))))

;; versions of nth- theorems with n = 0

(prove-lemma car-make-schedule-edf-simple (rewrite)
  (implies
   (lessp 0 n2)
   (equal (car (make-schedule-edf s r n2)) (car s)))
  ((use (nth-make-schedule-edf-simple (n1 0)))
   (disable nth-make-schedule-edf-simple)))

(prove-lemma car-edf-simple (rewrite)
  (equal (car (edf n2 r))
         (if (lessp 0 n2)
             (if (listp (unfulfilled 0 (edf 0 r)
                                     (active-task-requests 0 r)))
                 (car (least-deadline
                       (unfulfilled 0 (edf 0 r)
                                    (active-task-requests 0 r))))
               nil)
           0))
  ((use (nth-edf-simple (n 0)))
   (disable nth-edf-simple)))


(prove-lemma car-repeat (rewrite)
  (equal
   (car (repeat n v))
   (if (zerop n) 0 v)))

(prove-lemma numberp-first-instance (rewrite)
  (implies
   (numberp n)
   (equal (numberp (first-instance n v l))
          (member v (nthcdr n l)))))

(prove-lemma nth-make-element-simple (rewrite)
  (implies (lessp n1 n2)
           (equal (nth n1 (make-element-edf s r n2))
                  (nth n1 s))))

(prove-lemma car-make-element-simple (rewrite)
  (implies
   (not (zerop n))
   (equal (car (make-element-edf s r n))
```

```
         (car s)))
  ((use (nth-make-element-simple (n2 n) (n1 0)))
   (disable nth-make-element-simple)))


(prove-lemma car-make-element-edf (rewrite)
  (equal
   (car (make-element-edf s r n))
   (if (and (zerop n)
            (listp (unfulfilled 0 s (active-task-requests 0 r)))
            (first-instance 0 (car (least-deadline
                                    (unfulfilled 0 s
                                                 (active-task-requests 0 r))))
                            s))
       (car (least-deadline
             (unfulfilled 0 s (active-task-requests 0 r))))
     (car s)))
  ((use (nth-make-element-edf (n 0)))
   (disable nth-make-element-edf)))

(prove-lemma firstn-make-element-simple (rewrite)
  (implies
   (not (lessp n2 n1))
   (equal (firstn n1 (make-element-edf s r n2))
          (firstn n1 s))))

(prove-lemma firstn-sub1-cdr-make-element (rewrite)
  (implies
   (not (zerop n))
  (equal
   (firstn (sub1 n) (cdr (make-element-edf s r n)))
   (firstn (sub1 n) (cdr s))))
  ((use (firstn-make-element-simple (n1 n) (n2 n)))
   (disable-theory t)
   (enable-theory ground-zero task-abbr)
   (enable firstn)))

(prove-lemma nthcdr-n-cons-firstn-n (rewrite)
  (equal
   (nthcdr n (cons a (firstn n l)))
   (if (zerop n) (list a) (list (nth (sub1 n) l)))))

(prove-lemma nth-make-element-edf-sub1 (rewrite)
  (implies
   (not (zerop n))
  (equal
   (nth (sub1 n) (cdr (make-element-edf s r n)))
   (if (and (listp (unfulfilled n s (active-task-requests n r)))
            (first-instance n (car (least-deadline
                                    (unfulfilled n s
                                                 (active-task-requests n r))))
                            s))
       (car (least-deadline
             (unfulfilled n s (active-task-requests n r))))
     (nth n s))))
  ((use (nth-make-element-edf))
   (disable-theory t)
   (enable-theory ground-zero task-abbr)
   (enable nth)))

(prove-lemma equal-repeat-when-nil-not (rewrite)
  (implies
```

```
   (not (equal (car s) v))
   (equal (equal s (repeat (length s) v))
          (equal s nil))))

(prove-lemma member-car-schedule (rewrite)
  (implies
   (and
    (good-schedule s rs)
    (valid-requests rs)
    (member r rs))
   (member (car r) s)))

(prove-lemma sublistp-remove-until (rewrite)
  (implies
   (sublistp x (remove-until v y))
   (sublistp x y)))

(prove-lemma member-nil-periodic-task-requests (rewrite)
  (not (member nil (periodic-task-requests pt n1 n2))))

(prove-lemma member-nil-periodic-tasks-requests (rewrite)
  (not (member nil (periodic-tasks-requests pts n1 n2))))

(prove-lemma member-least-deadline-better (rewrite)
  (implies
   (and
    (sublistp r1 r2)
    (not (member nil r2)))
   (equal
    (member (least-deadline r1) r2)
    (listp r1)))
  ((induct (least-deadline r1))))

(prove-lemma make-schedule-edf-is-edf-simple nil
  (implies
   (and
    (non-overlapping-requests r)
    (cars-non-nil-litatoms r)
    (all-litatoms s)
    (good-schedule s r)
    (equal (plist s) s)
    (all-non-nil-corresponding s r 0)
    (equal r (periodic-tasks-requests pts 0 (length s)))
    (all-nils-or-cars s pts)
    (periodic-tasksp pts)
    (equal (edf n r) (firstn n s))
    (not (lessp (length s) n)))
   (equal
    (nthcdr n (make-schedule-edf s r n))
    (nthcdr n (edf (length s) r))))
  ((induct (make-schedule-edf s r n))
   (disable-theory t)
   (enable-theory ground-zero task-abbr)
   (enable difference-leq-arg1 difference-add1-arg2 equal-sub1-0 periodic-tasksp
           periodic-tasks-requests repeat firstn nthcdr length length-firstn equal-length-0
           plist good-schedule active-task-requests unfulfilled least-deadline
           length-make-element-edf make-schedule-edf append-nil firstn-length-list
           member-car-x-x periodic-tasks-requests-simple non-overlapping-requests
           plist-firstn equal-append nthcdr-1 nlistp-nthcdr firstn-cons firstn-firstn
           firstn-1 nthcdr-repeat firstn-nlistp cars-non-nil-litatoms
           all-non-nil-corresponding all-litatoms member-least-deadline
           unfulfilled-task-later-in-good-schedule sublistp-active-task-requests
```

```
           make-element-edf-preserves-good-schedule make-element-edf-preserves-all-litatoms
           lessp-n-1 all-nils-or-cars all-nils-or-cars-nlistp
           all-non-nil-corresponding-periodic-requests lessp-0-length-means-listp
           all-nils-or-cars-make-element-edf periodic-tasksp-means-cars-non-nil-litatoms
           cars-non-nil-litatoms-periodic-tasks
           non-overlapping-requests-periodic-tasks-requests length-cons edf length-edf-simple
           valid-requests-periodic-tasks listp-active-task-requests-0-multiple unfulfilled-0
           valid-requests-active-task-requests lessp-0-length-means edf-simple-nlistp
           make-schedule-edf-nlistp plist-make-element-edf length-make-schedule-edf
           equal-nthcdr-nthcdr-from-nthcdr-plus1 nth-make-schedule-edf-simple car-append
           listp-edf-simple unfulfilled-schedule-first-part-only nth-edf-simple
           equal-cdr-cdr-means first-instance-same-as-member nth-make-element-edf
           no-unfulfilled-active-task-if-nil make-element-nnumberp
           car-make-schedule-edf-simple car-edf-simple car-make-element-edf
           firstn-sub1-cdr-make-element nthcdr-n-cons-firstn-n nth-make-element-edf-sub1
           member-car-schedule member-least-deadline-better
           member-nil-periodic-tasks-requests)))

(prove-lemma make-schedule-edf-is-edf (rewrite)
  (implies
   (and
    (periodic-tasksp pts)
    (cars-non-nil-litatoms (periodic-tasks-requests pts 0 (length s)))
    (all-litatoms s)
    (good-schedule s (periodic-tasks-requests pts 0 (length s)))
    (equal (plist s) s)
    (all-non-nil-corresponding s (periodic-tasks-requests pts 0 (length s)) 0)
    (all-nils-or-cars s pts)
    (periodic-tasksp pts)
    (equal n (length s)))
   (equal
    (make-schedule-edf s (periodic-tasks-requests pts 0 n) 0)
    (edf (length s) (periodic-tasks-requests pts 0 (length s)))))
  ((use (make-schedule-edf-is-edf-simple
         (n 0)
         (r (periodic-tasks-requests pts 0 (length s)))))
   (disable non-overlapping-requests)))

(prove-lemma plist-make-simple-schedule (rewrite)
  (equal (plist (make-simple-schedule pts bigp length))
         (make-simple-schedule pts bigp length)))


(defn cpu-utilization (pts bigp)
  (if (listp pts)
      (plus (quotient (times bigp (tk-duration (car pts)))
                      (tk-period (car pts)))
            (cpu-utilization (cdr pts) bigp))
    0))

(prove-lemma length-substring-schedule (rewrite)
  (implies
   (and
    (expanded-tasksp pts bigp)
    (periodic-tasksp pts))
   (equal (length (substring-schedule pts bigp))
          (cpu-utilization pts bigp))))

(prove-lemma good-edf-for-expanded (rewrite)
  (implies
   (and
    (expanded-tasksp pts bigp)
```

```
    (not (lessp bigp (cpu-utilization pts bigp)))
    (periodic-tasksp pts)
    (equal (remainder n bigp) 0)
    (equal (remainder n (big-period pts)) 0))
   (good-schedule
    (edf n (periodic-tasks-requests pts 0 n))
    (periodic-tasks-requests pts 0 n)))
  ((use (edf-schedule-good-for-expanded))
   (disable make-simple-schedule)))


(defn expand-list (n list)
  (if (listp list)
      (append (repeat n (car list))
              (expand-list n (cdr list)))
    nil))

(prove-lemma expand-list-append (rewrite)
  (equal
   (expand-list n (append l1 l2))
   (append (expand-list n l1) (expand-list n l2))))

(prove-lemma listp-expand-list (rewrite)
  (equal
   (listp (expand-list n list))
   (and
    (not (zerop n))
    (listp list))))

(prove-lemma length-expand-list (rewrite)
  (equal
   (length (expand-list n list))
   (times n (length list))))

(prove-lemma plist-expand-list (rewrite)
  (equal (plist (expand-list n list)) (expand-list n list)))

(prove-lemma nthcdr-is-nil (rewrite)
  (equal
   (equal (nthcdr n l) nil)
   (and
    (equal (length l) (fix n))
    (equal (plist l) l))))

(prove-lemma active-task-requests-expand-tasks-requests (rewrite)
  (implies
   (not (zerop bigp))
  (equal
   (active-task-requests n (expand-tasks-requests r bigp))
   (expand-tasks-requests (active-task-requests (quotient n bigp) r) bigp)))
  ((induct (expand-tasks-requests r bigp))))

(prove-lemma repeat-1 (rewrite)
  (equal (repeat 1 v) (list v))
  ((expand (repeat 1 v))))

(disable lessp-difference-special)

(prove-lemma quotient-add1-plus-special (rewrite)
  (equal (quotient (add1 (plus z (times bigp v))) bigp)
         (if (zerop bigp) 0
           (plus v (quotient (add1 z) bigp))))
```

```
  ((use (quotient-plus-proof (a (add1 z)) (c bigp) (b (times bigp v))))))

(prove-lemma remainder-add1-plus-special (rewrite)
  (equal (remainder (add1 (plus z (times bigp v))) bigp)
         (if (zerop bigp) (add1 z)
           (remainder (add1 z) bigp)))
  ((use (remainder-plus-proof (a (add1 z)) (c bigp) (b (times bigp v))))))

(prove-lemma plist-nthcdr (rewrite)
  (implies
   (and
    (not (lessp (length l) n))
    (equal (plist l) l))
   (equal (plist (nthcdr n l)) (nthcdr n l))))

(prove-lemma remainder-plus-add1-hack (rewrite)
  (and
   (equal (remainder (plus z (times v z)) (add1 v)) 0)
   (equal (remainder (plus z (times z v)) (add1 v)) 0))
  ((use (remainder-times1-instance-proof (y (add1 v)) (x z)))
   (disable remainder-times1-instance)))

(prove-lemma quotient-plus-add1-hack (rewrite)
  (and
   (equal (quotient (plus z (times v z)) (add1 v)) (fix z))
   (equal (quotient (plus z (times z v)) (add1 v)) (fix z)))
  ((use (quotient-times-instance-temp-proof (y (add1 v)) (x z)))))

(prove-lemma equal-remainder-sub1-0 (rewrite)
  (implies
   (equal (remainder x y) 0)
   (equal (remainder (sub1 x) y)
          (if (zerop x) 0
          (if (zerop y) (sub1 x)
            (if (equal y 1) 0
              (sub1 y)))))))

(prove-lemma lessp-round-means (rewrite)
  (equal
   (lessp (times bigp (quotient n bigp)) n)
   (or
    (not (equal (remainder n bigp) 0))
    (and
     (zerop bigp)
     (not (zerop n))))))


(prove-lemma equal-repeat-nil (rewrite)
  (equal
   (equal (repeat n v) nil)
   (zerop n)))

(prove-lemma times-1-arg2 (rewrite)
  (equal (times x 1) (fix x)))

(prove-lemma firstn-0 (rewrite)
  (equal (firstn 0 l) nil))

(prove-lemma lessp-times-sub1-sub1 (rewrite)
  (equal
   (lessp (times bigp (sub1 (quotient n bigp))) (sub1 n))
   (and
```

```
    (lessp 1 n)
    (not (equal bigp 1)))))

(prove-lemma equal-cons-repeat (rewrite)
  (equal
   (equal (cons a b) (repeat n v))
   (and
    (equal a v)
    (not (zerop n))
    (equal b (repeat (sub1 n) v)))))

(prove-lemma lessp-1-means (rewrite)
  (equal
   (lessp 1 x)
   (and (not (zerop x)) (not (equal x 1)))))

(prove-lemma lessp-sub1-plus-hack (rewrite)
  (equal
   (lessp n1 (sub1 (plus n2 n1)))
   (lessp 1 n2)))

(prove-lemma firstn-nthcdr (rewrite)
   (equal (firstn x (nthcdr n s))
          (nthcdr n (firstn (plus n x) s))))

(disable firstn-nthcdr)

(prove-lemma nthcdr-x-firstn-x (rewrite)
  (equal (nthcdr n (firstn n l)) nil))

(prove-lemma nthcdr-x-edf-x (rewrite)
  (equal (nthcdr n (edf n r)) nil))

(prove-lemma nth-cons (rewrite)
  (equal (nth n (cons a b))
         (if (zerop n) a (nth (sub1 n) b))))

(prove-lemma firstn-nthcdr-too-big (rewrite)
  (implies
   (lessp n (length l))
   (equal
    (firstn (difference n z) (nthcdr z l))
    (if (lessp n z) nil (nthcdr z (firstn n l)))))
  ((induct (difference n z))))

(prove-lemma firstn-nthcdr-edf-plus (rewrite)
  (equal
   (firstn (difference n1 n2) (nthcdr n2 (edf (plus z n1) r)))
   (firstn (difference n1 n2) (nthcdr n2 (edf n1 r))))
  ((disable-theory t)
   (use (firstn-nthcdr-too-big (l (edf (plus z n1) r)) (n n1) (n2 z)))
   (enable firstn firstn-nthcdr edf nthcdr-x-firstn-x firstn-edf-simple repeat nthcdr-x-edf-x
           nthcdr length-append nthcdr-repeat nthcdr-x-firstn-x nthcdr-append
           length-edf-simple)
   (enable-theory ground-zero task-abbr naturals)))

(defn nthcdr-expand-induct (n b l)
  (if (listp l)
      (if (lessp b n)
          (nthcdr-expand-induct (difference n b) b (cdr l))
        t)
    t))
```

```
(prove-lemma nthcdr-expand-list (rewrite)
  (implies
   (equal (remainder n b) 0)
   (equal
    (nthcdr n (expand-list b l))
    (if (lessp (times b (length l)) n)
        0
      (expand-list b (nthcdr (quotient n b) l)))))
  ((induct (nthcdr-expand-induct n b l))
   (disable quotient-difference quotient-difference1)))

(prove-lemma expand-list-repeat (rewrite)
  (equal (expand-list b (repeat n v)) (repeat (times n b) v))
  ((induct (times n b))))

(prove-lemma firstn-expand-list (rewrite)
  (implies
   (equal (remainder n b) 0)
  (equal
   (firstn n (expand-list b l))
   (expand-list b (firstn (quotient n b) l))))
  ((induct (nthcdr-expand-induct n b l))
   (disable quotient-difference quotient-difference1)))

(prove-lemma occurrences-expand-list (rewrite)
  (equal
   (occurrences v (expand-list bigp l))
   (times bigp (occurrences v l))))

(prove-lemma expand-list-0 (rewrite)
  (implies
   (zerop n)
   (equal (expand-list n l) nil)))

(prove-lemma listp-firstn (rewrite)
  (equal (listp (firstn n l)) (not (zerop n))))

(prove-lemma equal-nil-firstn (rewrite)
  (equal (equal nil (firstn n l)) (zerop n)))

(prove-lemma firstn-difference-plus-nthcdr (rewrite)
  (implies
   (and
    (not (lessp (length l) (plus a b)))
    (not (lessp b c)))
  (equal
   (firstn (difference (plus a b) c) (nthcdr c l))
   (append
    (firstn (difference b c) (nthcdr c l))
    (firstn a (nthcdr b l))))))

(prove-lemma lessp-occurrences-firstn (rewrite)
  (not (lessp n (occurrences v (firstn n l)))))

(disable lessp-occurrences-firstn)

(prove-lemma lessp-occurrences-edf (rewrite)
  (not (lessp n (occurrences v (edf n r))))
  ((use (lessp-occurrences-firstn (l (edf n r))))))

(prove-lemma equal-times-hack (rewrite)
```

```
  (implies
   (lessp a b)
   (equal
    (equal a (times b n))
    (and
     (equal a 0)
     (not (zerop b))
     (zerop n)))))

(prove-lemma equal-occurrences-firstn-times (rewrite)
  (implies
   (lessp z b)
  (equal
   (equal (occurrences v (firstn z l)) (times b n))
   (and
    (zerop n)
    (zerop (occurrences v (firstn z l))))))
  ((use (lessp-occurrences-firstn (n z)))))


(prove-lemma firstn-plus (rewrite)
  (equal (firstn (plus a b) l)
         (append (firstn b l) (firstn a (nthcdr b l)))))

(prove-lemma equal-plus-times (rewrite)
  (implies
   (lessp z b)
   (equal
    (equal (plus (times b n) z) (times b v))
    (and
     (zerop z)
     (equal (fix n) (fix v)))))
  ((induct (double-sub1-induction n v))))

(prove-lemma firstn-noop (rewrite)
  (implies
   (equal (length l) (fix n))
   (equal (firstn n l) (plist l))))

(prove-lemma lessp-x-x (rewrite)
  (equal (lessp x x) f))

(prove-lemma overlapping-non-overlapping3-means (rewrite)
  (implies
   (and
    (non-overlapping-requests3 r r2)
    (not (lessp x (cadr r)))
    (equal (car req) (car r))
    (not (lessp x (cadr req)))
    (lessp x (caddr r))
    (member req r2))
   (equal (lessp x (caddr req)) (equal r req)))
  ((induct (non-overlapping-requests3 r r2))
   (disable-theory t)
   (enable non-overlapping-requests3)
   (enable-theory ground-zero task-abbr naturals)))

(prove-lemma assoc-unfulfilled-expanded-non-overlapping (rewrite)
  (implies
   (and
    (not (member r r2))
    (non-overlapping-requests3 r r2)
```

```
    (lessp z b)
    (lessp x (caddr r))
    (not (lessp x (cadr r))))
   (not (assoc (car r)
               (unfulfilled (plus z (times b x)) s
                            (expand-tasks-requests (active-task-requests x r2)
                                                   b)))))
  ((induct (non-overlapping-requests3 r r2))
   (disable-theory t)
   (enable-theory ground-zero task-abbr naturals)
   (enable listp-bagint-with-singleton-implies-member expand-tasks-requests unfulfilled
           non-overlapping-requests3 active-task-requests)))

(prove-lemma not-assoc-car-req (rewrite)
  (implies
   (and
    (valid-requests r2)
    (member req r2)
    (lessp x (caddr req))
    (not (lessp x (cadr req)))
    (non-overlapping-requests r2)
    (lessp z b)
    (equal (occurrences (car req)
                        (firstn (difference (plus z (times b x))
                                            (times b (cadr req)))
                                (nthcdr (times b (cadr req)) s)))
           (times b (cadddr req))))
   (not (assoc (car req)
               (unfulfilled (plus z (times b x))
                            s
                            (expand-tasks-requests (active-task-requests x r2)
                                                   b)))))
  ((induct (active-task-requests x r2))
   (disable-theory t)
   (enable-theory ground-zero task-abbr naturals)
   (enable overlapping-non-overlapping3-means valid-requests non-overlapping-requests2
           non-overlapping-requests3 non-overlapping-requests2
           assoc-unfulfilled-expanded-non-overlapping non-overlapping-requests
           active-task-requests non-overlapping-requests-cdr equal-occurrences-zero
           expand-tasks-requests unfulfilled)))

(prove-lemma equal-car-car-hack (rewrite)
  (implies
   (and
    (member e1 l)
    (not (assoc (car e2) l))
    (valid-requests l)
    (listp e1)
    (listp e2))
   (not (equal (car e1) (car e2)))))

(prove-lemma not-member-nthcdr-add1 (rewrite)
  (implies
   (not (member v (nthcdr (add1 x) s)))
   (equal (member v (nthcdr x s))
          (and
           (lessp x (length s))
           (equal v (nth x s))))))

(prove-lemma occurrences-hack (rewrite)
  (implies
   (and
```

```
   (not (member v (nthcdr n s)))
   (not (lessp n b))
   (litatom v))
   (equal
    (occurrences v (firstn (difference (plus a n) b) (nthcdr b s)))
    (occurrences v (firstn (difference n b) (nthcdr b s)))))
  ((induct (plus n a))))

(prove-lemma occurrences-hack2 (rewrite)
  (implies
   (and
    (not (member v (nthcdr n l)))
    (lessp n (length l)))
    (equal (occurrences v (nthcdr n1 l))
           (occurrences v (nthcdr n1 (firstn n l))))))

(prove-lemma valid-requests-unfulfilled (rewrite)
  (implies
   (valid-requests r)
   (valid-requests (unfulfilled n s r))))


(prove-lemma valid-requests-expand-tasks-requests (rewrite)
  (implies
   (valid-requests r)
   (equal
    (valid-requests (expand-tasks-requests r b))
    (or
     (not (zerop b))
     (not (listp r))))))

(prove-lemma litatom-caar (rewrite)
  (implies
   (valid-requests r)
   (equal (litatom (caar r)) (listp r))))

(disable litatom-caar)

(prove-lemma numberp-cadar (rewrite)
  (implies
   (valid-requests r)
   (numberp (cadar r))))

(disable numberp-cadar)

(prove-lemma not-equal-car-least-deadline (rewrite)
  (implies
   (and
    (valid-requests r2)
    (member req r2)
    (lessp x (caddr req))
    (not (lessp x (cadr req)))
    (non-overlapping-requests r2)
    (lessp z b)
    (not (zerop b))
    (litatom (car req))
    (equal (occurrences (car req)
                        (firstn (difference (plus z (times b x))
                                            (times b (cadr req)))
                                (nthcdr (times b (cadr req)) s)))
           (times b (cadddr req))))
   (not (equal
```

```
        (car req)
        (car (least-deadline
              (unfulfilled (plus z (times b x))
                           s
                           (expand-tasks-requests (active-task-requests x r2)
                                                  b)))))))
  ((use (equal-car-car-hack
        (e1 (least-deadline
             (unfulfilled (plus z (times b x))
                          s
                          (expand-tasks-requests (active-task-requests x r2)
                                                 b))))
        (l  (unfulfilled (plus z (times b x))
                          s
                          (expand-tasks-requests (active-task-requests x r2)
                                                 b)))
        (e2 req)))
   (disable-theory t)
   (enable-theory ground-zero task-abbr naturals)
   (enable member-least-deadline not-assoc-car-req least-deadline valid-requests-unfulfilled
           valid-requests-active-task-requests valid-requests-expand-tasks-requests)))

(prove-lemma not-equal-car-least-deadline-special (rewrite)
  (implies
   (and
    (valid-requests r2)
    (member req r2)
    (lessp x (caddr req))
    (not (lessp x (cadr req)))
    (non-overlapping-requests r2)
    (not (zerop b))
    (litatom (car req))
    (equal (occurrences (car req)
                        (firstn (difference (times b x)
                                            (times b (cadr req)))
                                (nthcdr (times b (cadr req)) s)))
           (times b (cadddr req))))
   (not (equal
        (car req)
        (car (least-deadline
              (unfulfilled (times b x)
                           s
                           (expand-tasks-requests (active-task-requests x r2)
                                                  b)))))))
  ((use (not-equal-car-least-deadline (z 0)))))

(prove-lemma occurrences-edf-satisfied (rewrite)
            (implies
             (and
              (listp r)
              (lessp x (caddar r))
              (not (lessp x (cadar r)))
              (lessp z b)
              (equal (expand-list b (edf x r2))
                     (edf (times b x)
                          (expand-tasks-requests r2 b)))
              (valid-requests r2)
              (valid-requests r)
              (non-overlapping-requests r2)
              (sublistp r r2)
              (not (equal b 0))
              (numberp b)
```

```
              (numberp x)
              (equal
                  (occurrences (caar r)
                               (firstn (difference x (cadar r))
                                       (nthcdr (cadar r) (edf x r2))))
                  (cadddar r)))
             (equal
              (occurrences
                 (caar r)
                 (firstn z
                         (nthcdr (times b x)
                                 (edf (plus z (times b x))
                                      (expand-tasks-requests r2 b)))))
              0))
           ((induct (plus z b))
            (expand (edf (add1 (plus (sub1 z) (times b x)))
                         (expand-tasks-requests r2 b)))
            (disable-theory t)
            (enable-theory ground-zero task-abbr naturals)
            (enable equal-occurrences-zero member-append occurrences-hack member-sublistp
                    member-car-x-x numberp-cadar not-equal-car-least-deadline-special
                    litatom-caar occurrences-hack2 firstn-edf-simple nthcdr-x-edf-x
                    member-non-list firstn-nlistp firstn-expand-list expand-list firstn
                    repeat zerop lessp-x-x firstn-append valid-requests
                    active-task-requests-expand-tasks-requests length-edf-simple
                    equal-sub1-0 nthcdr-expand-list occurrences-expand-list
                    not-equal-car-least-deadline occurrences-hack nthcdr-append firstn-noop
                    plist-nthcdr plist-edf-simple nthcdr length-nthcdr firstn-nlistp
                    firstn-cons firstn-too-big)))


(prove-lemma unfulfilled-expanded-on-line (rewrite)
  (implies
   (and
    (lessp z b)
    (equal (expand-list b (edf x r2))
           (edf (times x b) (expand-tasks-requests r2 b)))
    (valid-requests r2)
    (valid-requests r)
    (non-overlapping-requests r2)
    (sublistp r r2))
  (equal
   (unfulfilled (plus z (times x b))
                (edf (plus z (times x b)) (expand-tasks-requests r2 b))
                (expand-tasks-requests (active-task-requests x r) b))
   (unfulfilled (times x b)
                (edf (plus z (times x b)) (expand-tasks-requests r2 b))
                (expand-tasks-requests (active-task-requests x r) b))))
  ((induct (active-task-requests x r))
   (enable sublistp-cdr1 sublistp-cdr2 edf firstn-nthcdr-edf-plus length-edf-simple
           occurrences firstn-edf-simple lessp-occurrences-edf occurrences-expand-list
           valid-requests firstn nthcdr equal-times-hack firstn-plus equal-plus-times
           firstn-difference-plus-nthcdr occurrences-append occurrences-edf-satisfied
           lessp-occurrences-firstn nthcdr-expand-list expand-list-0 firstn-expand-list
           active-task-requests expand-tasks-requests unfulfilled)
   (enable-theory ground-zero task-abbr naturals)
   (disable-theory t)))

(prove-lemma unfulfilled-expand-list (rewrite)
  (implies
   (and
    (valid-requests r)
```

```
    (equal (remainder n b) 0)
    (not (zerop b)))
   (equal
    (unfulfilled n (expand-list b s) (expand-tasks-requests r b))
    (expand-tasks-requests (unfulfilled (quotient n b) s r) b)))
  ((induct (expand-tasks-requests r b))))

(prove-lemma equal-nil-expand-list (rewrite)
  (equal
   (equal nil (expand-list b l))
   (or
    (zerop b)
    (nlistp l))))

(prove-lemma valid-requests-sublistp (rewrite)
  (implies
   (and
    (sublistp x y)
    (equal (plist x) x)
    (valid-requests y))
   (valid-requests x)))

(prove-lemma unfulfilled-sub1-expanded-on-line (rewrite)
  (implies
   (and
    (not (zerop a))
    (lessp a b)
    (equal (expand-list b (edf x r2))
           (edf (times x b) (expand-tasks-requests r2 b)))
    (non-overlapping-requests r2)
    (valid-requests r2)
    (valid-requests r)
    (sublistp r r2))
  (equal
   (unfulfilled (sub1 (plus a (times b x)))
                (edf (sub1 (plus a (times b x))) (expand-tasks-requests r2 b))
                (expand-tasks-requests (active-task-requests x r) b))
   (unfulfilled (times x b)
                (edf (sub1 (plus a (times x b))) (expand-tasks-requests r2 b))
                (expand-tasks-requests (active-task-requests x r) b))))
  ((use (unfulfilled-expanded-on-line (z (sub1 a))))
   (disable-theory t)
   (enable-theory naturals ground-zero task-abbr)))

(prove-lemma firstn-only-helper nil
  (implies
   (equal (firstn n a) (firstn n b))
   (equal (firstn (difference n c) (nthcdr c a))
          (firstn (difference n c) (nthcdr c b)))))

(prove-lemma firstn-nthcdr-edf-plus-simple (rewrite)
  (equal (firstn (difference n a) (nthcdr a (edf (plus b n) r)))
         (firstn (difference n a) (nthcdr a (edf n r))))
  ((use (firstn-only-helper (n n) (a (edf n r)) (b (edf (plus b n) r))
                            (c a)))))

(prove-lemma unfulfilled-too-big (rewrite)
  (equal
   (unfulfilled n (edf (plus a n) r1) r2)
   (unfulfilled n (edf n r1) r2)))

(prove-lemma unfulfilled-too-big-sub1 (rewrite)
```

```
  (implies
   (not (zerop a))
   (equal
    (unfulfilled n (edf (sub1 (plus a n)) r1) r2)
    (unfulfilled n (edf n r1) r2)))
  ((use (unfulfilled-too-big (a (sub1 a))))
   (disable unfulfilled-too-big)))

(prove-lemma car-least-deadline-expand-tasks-requests (rewrite)
  (implies
   (not (zerop b))
  (equal
   (car (least-deadline (expand-tasks-requests r b)))
   (car (least-deadline r))))
  ((induct (least-deadline r))))

(prove-lemma remainder-hack (rewrite)
  (implies
   (equal (remainder n p) 0)
   (equal
    (remainder (sub1 (plus x n)) p)
    (if (zerop x)
        (remainder (sub1 n) p)
      (remainder (sub1 x) p))))
  ((use (remainder-plus-proof (b n) (c p) (a (sub1 x))))))

(prove-lemma equal-car-least-deadline-nil (rewrite)
  (implies
   (valid-requests r)
   (not (equal (car (least-deadline r)) nil))))

(prove-lemma unfulfilled-0-expand-tasks-requests (rewrite)
  (implies
   (and
    (valid-requests r)
    (not (zerop b)))
  (equal
   (unfulfilled 0 s (expand-tasks-requests r b))
   (expand-tasks-requests (unfulfilled 0 s r) b))))


(prove-lemma unfulfilled-0-edf (rewrite)
  (equal
   (unfulfilled 0 (edf n r) r2)
   (unfulfilled 0 nil r2)))

(prove-lemma unfulfilled-expanded-on-line-beginning (rewrite)
  (implies
   (and
    (lessp z b)
    (non-overlapping-requests r2)
    (not (equal z 0))
    (sublistp r r2)
    (valid-requests r)
    (valid-requests r2))
  (equal
   (unfulfilled z
             (edf z (expand-tasks-requests r2 b))
            (expand-tasks-requests (active-task-requests 0 r) b))
   (unfulfilled 0
          (edf z (expand-tasks-requests r2 b))
            (expand-tasks-requests (active-task-requests 0 r) b))))
```

```
  ((use (unfulfilled-expanded-on-line (x 0)))))

(prove-lemma edf-simple-expand-tasks-requests (rewrite)
  (implies
   (and
    (lessp 1 bigp)
    (non-overlapping-requests r)
    (valid-requests r))
  (equal
   (edf n (expand-tasks-requests r bigp))
   (append
    (expand-list bigp (edf (quotient n bigp) r))
    (let ((undone (unfulfilled
                        (quotient n bigp)
                        (edf (quotient n bigp) r)
                        (active-task-requests (quotient n bigp) r))))
      (repeat (remainder n bigp)
              (if (listp undone)
                (car (least-deadline
                       undone))
              nil))))))
  ((induct (plus n bigp))
   (disable-theory t)
   (enable-theory naturals ground-zero task-abbr)
   (disable times-add1)
   (enable edf expand-list listp-edf-simple listp-expand-list valid-requests-unfulfilled
           unfulfilled-expanded-on-line-beginning equal-car-least-deadline-nil
           length-edf-simple length-expand-list plist-expand-list firstn-edf-simple
           equal-append equal-repeat-repeat equal-cons-repeat lessp-1-means
           remainder-add1-plus-special equal-repeat-nil firstn-0 lessp-times-sub1-sub1 firstn
           unfulfilled-sub1-expanded-on-line unfulfilled-expanded-on-line sublistp-x-x
           unfulfilled-too-big car-least-deadline-expand-tasks-requests nthcdr-expand-list
           listp-expand-tasks-requests valid-requests-active-task-requests expand-list-repeat
           unfulfilled-too-big-sub1 firstn-expand-list unfulfilled-0-edf remainder-hack
           unfulfilled-0-expand-tasks-requests occurrences-expand-list
           unfulfilled-expand-list equal-remainder-sub1-0 lessp-sub1-plus-hack
           active-task-requests-expand-tasks-requests append-nil remainder-plus-add1-hack
           quotient-plus-add1-hack firstn-append firstn-edf-simple length-nthcdr plist-repeat
           plist-edf-simple listp-edf-simple lessp-round-means firstn-repeat length-repeat
           nthcdr-repeat length plist-expand-list quotient-add1-plus-special repeat-1
           plist-nthcdr nthcdr-is-nil length-append expand-list-append nlistp-nthcdr
           listp-nthcdr nthcdr-append nthcdr listp-repeat equal-nil-expand-list times-1-arg2
           repeat expand-list)))

(prove-lemma quotient-difference-special (rewrite)
  (equal
   (quotient (difference (times a b) (times a c)) a)
   (if (zerop a) 0 (difference b c))))

(prove-lemma good-schedule-expand-tasks-reduces (rewrite)
  (implies
   (and
    (lessp 1 bigp)
    (non-overlapping-requests r1)
    (sublistp r2 r1)
    (valid-requests r1)
    (valid-requests r2)
    (equal (remainder n bigp) 0)
    (good-schedule (edf n (expand-tasks-requests r1 bigp))
                   (expand-tasks-requests r2 bigp)))
   (good-schedule (edf (quotient n bigp) r1) r2))
  ((induct (expand-tasks-requests r2 bigp))
```

```
   (enable-theory ground-zero task-abbr)
   (disable-theory t)
   (enable equal-times-0 commutativity-of-times times-quotient
           correctness-of-cancel-equal-times remainder-noop remainder-of-non-number
           remainder-times1-instance firstn remainder-difference1 quotient-times-instance
           quotient-lessp-arg1 repeat nthcdr good-schedule append-nil sublistp sublistp-cdr1
           sublistp-cdr2 occurrences-repeat nlistp-nthcdr firstn-nlistp expand-tasks-requests
           edf length-edf-simple valid-requests expand-list plist-expand-list
           edf-simple-expand-tasks-requests nthcdr-expand-list firstn-expand-list
           occurrences-expand-list quotient-difference-special)))

(prove-lemma expand-tasks-1 (rewrite)
  (implies
   (periodic-tasksp pts)
   (equal (expand-tasks pts 1) pts)))

(prove-lemma cpu-utilization-expand-tasks (rewrite)
  (implies
   (periodic-tasksp pts)
  (equal
   (cpu-utilization (expand-tasks pts n) n2)
   (if (zerop n) 0 (cpu-utilization pts n2)))))

(prove-lemma big-period-expand-tasks (rewrite)
  (equal
   (big-period (expand-tasks pts n))
   (times (exp n (length pts)) (big-period pts))))

(prove-lemma remainder-from-exp (rewrite)
  (implies
   (and
    (equal (remainder n (exp x y)) 0)
    (not (zerop y)))
   (equal (remainder n x) 0)))

(prove-lemma good-edf-help nil
  (implies
   (and
    (not (lessp (big-period pts) (cpu-utilization pts (big-period pts))))
    (periodic-tasksp pts)
    (listp pts)
    (lessp 1 (big-period pts))
    (equal (remainder n (exp (big-period pts) (length pts))) 0))
    (good-schedule
     (edf n (periodic-tasks-requests pts 0 n))
     (periodic-tasks-requests pts 0 n)))
  ((use (good-edf-for-expanded
         (pts (expand-tasks pts (big-period pts)))
         (bigp (big-period pts)) (n (times n (big-period pts))))
        (good-schedule-expand-tasks-reduces
         (bigp (big-period pts))
         (n (times n (big-period pts)))
         (r1 (periodic-tasks-requests pts 0 n))
         (r2 (periodic-tasks-requests pts 0 n))))
   (disable-theory t)
   (enable expanded-tasksp-expand-task expand-tasks-1 sublistp-x-x
           valid-requests-periodic-tasks cpu-utilization-expand-tasks
           periodic-tasks-requests-simple good-schedule equal-length-0
           periodic-tasks-requests remainder-from-exp periodic-tasks-requests-expand
           periodic-tasksp-expand-tasks big-period-expand-tasks
           non-overlapping-requests-periodic-tasks-requests)
   (enable-theory ground-zero task-abbr naturals)))
```

```
(prove-lemma good-edf-with-remainder nil
  (implies
   (and
    (not (lessp (big-period pts) (cpu-utilization pts (big-period pts))))
    (periodic-tasksp pts)
    (lessp 1 (big-period pts))
    (equal (remainder n (exp (big-period pts) (length pts))) 0))
    (good-schedule
     (edf n (periodic-tasks-requests pts 0 n))
     (periodic-tasks-requests pts 0 n)))
  ((use (good-edf-help))))

(defn requests-deadlines-not-greater (r time)
  (if (listp r)
      (if (lessp time (caddar r))
          (requests-deadlines-not-greater (cdr r) time)
        (cons (car r) (requests-deadlines-not-greater (cdr r) time)))
    nil))

(defn firstn-nthcdr-firstn-induct (a b n l)
  (if (zerop a) t
    (firstn-nthcdr-firstn-induct (sub1 a) (sub1 b) (sub1 n) l)))

(prove-lemma firstn-nthcdr-firstn-simple (rewrite)
  (implies
   (not (lessp n (plus a b)))
   (equal (firstn a (nthcdr b (firstn n l))) (firstn a (nthcdr b l))))
  ((enable firstn-nthcdr firstn-firstn)
   (disable-theory t)
   (enable-theory ground-zero task-abbr naturals)))

(prove-lemma good-schedule-requests-not-greater (rewrite)
  (implies
   (and
    (good-schedule s r)
    (valid-requests r))
   (good-schedule (firstn n s) (requests-deadlines-not-greater r n))))

(prove-lemma requests-deadlines-append (rewrite)
  (equal
   (requests-deadlines-not-greater (append a b) time)
   (append
    (requests-deadlines-not-greater a time)
    (requests-deadlines-not-greater b time))))

(prove-lemma plist-requests-deadlines (rewrite)
  (equal (plist (requests-deadlines-not-greater a time))
         (requests-deadlines-not-greater a time)))

(prove-lemma requests-deadlines-periodic-task-simple (rewrite)
  (implies
   (not (lessp n1 n2))
   (equal (requests-deadlines-not-greater (periodic-task-requests pt n1 n) n2)
          nil)))

(prove-lemma equal-nil-periodic-task-requests (rewrite)
  (equal
   (equal (periodic-task-requests pt n1 n2) nil)
   (or
    (not (periodic-taskp pt))
    (not (lessp n1 n2)))))
```

```
(prove-lemma requests-deadlines-not-greater-periodic-task (rewrite)
  (implies
   (and
    (equal (remainder n (cadr pt)) 0)
    (equal (remainder n2 (cadr pt)) 0)
    (equal (remainder n1 (cadr pt)) 0))
  (equal
   (requests-deadlines-not-greater (periodic-task-requests pt n1 n2) n)
   (if (lessp n n2)
       (periodic-task-requests pt n1 n)
     (periodic-task-requests pt n1 n2)))))

(prove-lemma equal-remainder-big-period (rewrite)
  (implies
   (member e l)
   (equal (remainder (big-period l) (cadr e)) 0)))

(prove-lemma remainder-period-0-if-big-period (rewrite)
  (implies
   (and
    (periodic-tasksp pts)
    (equal (remainder n (big-period pts)) 0)
    (member pt pts))
   (equal (remainder n (cadr pt)) 0))
  ((induct (member pt pts))))

(prove-lemma equal-nil-periodic-tasks-requests (rewrite)
  (equal
   (equal nil (periodic-tasks-requests pts n1 n2))
   (or (nlistp pts) (not (periodic-tasksp pts)) (not (lessp n1 n2)))))

(prove-lemma requests-deadlines-not-greater-periodic-help nil
  (implies
   (and
    (periodic-tasksp pts2)
    (sublistp pts1 pts2)
    (equal n1 0)
    (not (lessp n2 n3))
    (equal (remainder n2 (big-period pts2)) 0)
    (equal (remainder n3 (big-period pts2)) 0))
   (equal
    (requests-deadlines-not-greater (periodic-tasks-requests pts1 n1 n2) n3)
    (periodic-tasks-requests pts1 n1 n3)))
  ((induct (periodic-tasks-requests pts1 n1 n2))))

(prove-lemma requests-deadlines-not-greater-periodic-rewrite (rewrite)
  (implies
   (and
    (periodic-tasksp pts)
    (equal n1 0)
    (not (lessp n2 n3))
    (equal (remainder n2 (big-period pts)) 0)
    (equal (remainder n3 (big-period pts)) 0))
   (equal
    (requests-deadlines-not-greater (periodic-tasks-requests pts n1 n2) n3)
    (periodic-tasks-requests pts n1 n3)))
  ((use (requests-deadlines-not-greater-periodic-help
         (pts1 pts) (pts2 pts)))))

(prove-lemma plist-active-task-requests (rewrite)
  (equal (plist (active-task-requests n r)) (active-task-requests n r)))
```

```
(prove-lemma lessp-plus-times-hack (rewrite)
  (equal
   (lessp (plus x (times x y)) (times x z))
   (and
    (not (zerop x))
    (lessp (add1 y) z))))

(prove-lemma periodic-task-requests-simple (rewrite)
  (implies
   (not (lessp n1 n2))
   (equal (periodic-task-requests pt n1 n2) nil)))

(prove-lemma active-task-requests-periodic-task-requests-simple (rewrite)
  (implies
   (lessp n n1)
   (equal (active-task-requests n (periodic-task-requests pt n1 n2)) nil)))

(prove-lemma active-task-requests-periodic-task-requests (rewrite)
  (implies
   (and
    (equal (remainder n1 (cadr pt)) 0)
    (equal (remainder n2 (cadr pt)) 0)
    (equal (remainder n (cadr pt)) 0)
    (numberp n1))
   (equal
    (active-task-requests n (periodic-task-requests pt n1 n2))
    (if (or (not (periodic-taskp pt)) (lessp n n1) (not (lessp n n2))) nil
      (list
       (list (car pt)
             (times (cadr pt) (quotient n (cadr pt)))
             (plus (cadr pt) (times (cadr pt) (quotient n (cadr pt))))
             (caddr pt))))))
  ((induct (periodic-task-requests pt n1 n2))))

(prove-lemma active-task-requests-not-greater (rewrite)
  (equal (active-task-requests n1 (requests-deadlines-not-greater r n2))
         (requests-deadlines-not-greater (active-task-requests n1 r) n2)))

(defn requests-starts-earlier (r time)
  (if (listp r)
      (if (lessp (cadar r) time)
          (cons (car r) (requests-starts-earlier (cdr r) time))
        (requests-starts-earlier (cdr r) time))
    nil))

(prove-lemma requests-starts-earlier-append (rewrite)
  (equal
   (requests-starts-earlier (append a b) n)
   (append
   (requests-starts-earlier a n)
   (requests-starts-earlier b n))))

(prove-lemma active-task-requests-requests-starts-earlier (rewrite)
  (implies
   (lessp n n1)
   (equal (active-task-requests n (requests-starts-earlier r n1))
          (active-task-requests n r))))

(prove-lemma edf-requests-starts-earlier (rewrite)
  (implies
```

```
   (not (lessp n1 n))
   (equal
    (edf n (requests-starts-earlier r n1))
    (edf n r)))
  ((induct (edf n r))))

(prove-lemma plist-requests-starts-earlier (rewrite)
  (equal (plist (requests-starts-earlier r n))
         (requests-starts-earlier r n)))

(prove-lemma requests-starts-earlier-periodic-task (rewrite)
  (implies
   (and
    (equal (remainder n (cadr pt)) 0)
    (equal (remainder n1 (cadr pt)) 0)
    (equal (remainder n2 (cadr pt)) 0))
   (equal (requests-starts-earlier (periodic-task-requests pt n1 n2) n)
          (periodic-task-requests pt n1 (if (lessp n n2) n n2))))
  ((induct (periodic-task-requests pt n1 n2))))

(prove-lemma requests-starts-earlier-periodic-tasks (rewrite)
  (implies
   (and
    (equal (remainder n (big-period pts2)) 0)
    (equal (remainder n1 (big-period pts2)) 0)
    (equal (remainder n2 (big-period pts2)) 0)
    (sublistp pts1 pts2)
    (periodic-tasksp pts2))
   (equal (requests-starts-earlier (periodic-tasks-requests pts1 n1 n2) n)
          (periodic-tasks-requests pts1 n1 (if (lessp n n2) n n2))))
  ((induct (periodic-tasks-requests pts1 n1 n2))))

(prove-lemma good-edf-almost nil
  (implies
   (and
    (not (lessp (big-period pts) (cpu-utilization pts (big-period pts))))
    (periodic-tasksp pts)
    (lessp 1 (big-period pts))
    (equal (remainder n (big-period pts)) 0))
    (good-schedule
     (edf n (periodic-tasks-requests pts 0 n))
     (periodic-tasks-requests pts 0 n)))
  ((use (good-edf-with-remainder
         (n (times n (exp (big-period pts) (length pts)))))
        (good-schedule-requests-not-greater
         (s (edf (times n (exp (big-period pts) (length pts)))
                 (periodic-tasks-requests
                  pts 0 (times n (exp (big-period pts) (length pts))))))
         (r (periodic-tasks-requests
             pts 0 (times n (exp (big-period pts) (length pts)))))
         (n n))
        (edf-requests-starts-earlier
         (n1 n)
         (r (periodic-tasks-requests pts 0
                         (times n
                                (exp (big-period pts) (length pts))))))))


(prove-lemma equal-plus-1 (rewrite)
  (equal
   (equal (plus a b) 1)
   (or
```

```
    (and (zerop a) (equal b 1))
    (and (zerop b) (equal a 1)))))

(prove-lemma equal-cpu-utilization-0 (rewrite)
  (implies
   (and
    (equal (remainder n (big-period x)) 0)
    (not (zerop n))
    (periodic-tasksp x))
   (equal
    (equal (cpu-utilization x n) 0)
    (equal x nil)))
  ((expand (cpu-utilization x n))))

(prove-lemma equal-1-big-period (rewrite)
  (implies
   (and
    (not (lessp (big-period pts) (cpu-utilization pts (big-period pts))))
    (equal (big-period pts) 1))
   (equal
    (periodic-tasksp pts)
    (or (equal pts nil)
        (and
         (equal (plist pts) pts)
         (equal (length pts) 1)
         (periodic-taskp (car pts))
         (equal (cadar pts) 1)
         (equal (caddar pts) 1)))))
  ((expand (big-period pts))))

(defn simple-requests (v n1 n2)
  (if (lessp n1 n2)
      (cons (list v n1 (add1 n1) 1) (simple-requests v (add1 n1) n2))
    nil)
  ((lessp (difference n2 n1))))

(prove-lemma periodic-task-requests-as-simple-requests (rewrite)
  (equal (periodic-task-requests (list v 1 1) n1 n2)
         (if (or (not (litatom v)) (equal v nil)) nil
           (simple-requests v n1 n2))))


(prove-lemma active-task-requests-simple-requests (rewrite)
  (implies
   (and
    (numberp n1)
    (numberp n))
  (equal
   (active-task-requests n (simple-requests v n1 n2))
   (if (lessp n n1) nil
     (if (lessp n n2)
         (list (list v n (add1 n) 1))
       nil)))))

(prove-lemma nthcdr-cons-cons-repeat (rewrite)
  (equal
   (nthcdr n (cons a (cons a (repeat m a))))
   (if (lessp (plus 2 m) n) 0
     (repeat (difference (plus 2 m) n) a)))
  ((induct (repeat m a))
   (expand (nthcdr n (list a a)))))
```

```
(prove-lemma requests-starts-earlier-simple-requests (rewrite)
  (equal
   (requests-starts-earlier (simple-requests v n1 n2) n)
   (if (lessp n n2)
       (simple-requests v n1 n)
     (simple-requests v n1 n2)))
  ((expand (simple-requests v (add1 n1) n))))

(prove-lemma edf-simple-requests-too-big (rewrite)
  (implies
   (lessp z n2)
   (equal
    (edf z (simple-requests v n1 n2))
    (edf z (simple-requests v n1 z))))
  ((use (edf-requests-starts-earlier (n z) (n1 z)
                                     (r (simple-requests v n1 n2))))
   (disable edf-requests-starts-earlier)))

(prove-lemma edf-n-simple-requests (rewrite)
  (equal
   (edf n (simple-requests v 0 n))
   (repeat n v)))

(prove-lemma good-schedule-repeat-simple (rewrite)
  (good-schedule (repeat n v) (simple-requests v n1 n)))

(prove-lemma plist-simple-requests (rewrite)
  (equal (plist (simple-requests v n1 n2)) (simple-requests v n1 n2)))

(prove-lemma good-edf-bigp-1 nil
  (implies
   (and
    (not (lessp (big-period pts) (cpu-utilization pts (big-period pts))))
    (periodic-tasksp pts)
    (equal (big-period pts) 1)
    (equal (remainder n (big-period pts)) 0))
    (good-schedule
     (edf n (periodic-tasks-requests pts 0 n))
     (periodic-tasks-requests pts 0 n))))

(prove-lemma good-edf-periodic nil
  (implies
   (and
    (not (lessp (big-period pts) (cpu-utilization pts (big-period pts))))
    (periodic-tasksp pts)
    (equal (remainder n (big-period pts)) 0))
    (good-schedule
     (edf n (periodic-tasks-requests pts 0 n))
     (periodic-tasks-requests pts 0 n)))
  ((use (good-edf-bigp-1) (good-edf-almost))
   (disable-theory t) (enable-theory ground-zero task-abbr naturals)
   (enable zerop-big-period)))


))
```

Appendix D

"nim.events"

```
(proveall "nim"
'(

#|
 Copyright (C) 1993 by Matt Wilding.

 This script is hereby placed in the public domain, and therefore unlimited
 editing and redistribution is permitted.

 NO WARRANTY

 Matt Wilding PROVIDES ABSOLUTELY NO WARRANTY.  THE EVENT SCRIPT IS
 PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED,
 INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND
 FITNESS FOR A PARTICULAR PURPOSE.  THE ENTIRE RISK AS TO THE QUALITY AND
 PERFORMANCE OF THE SCRIPT IS WITH YOU.  SHOULD THE SCRIPT PROVE DEFECTIVE, YOU
 ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

 IN NO EVENT WILL Matt Wilding BE LIABLE TO YOU FOR ANY DAMAGES,
 ANY LOST PROFITS, LOST MONIES, OR OTHER SPECIAL, INCIDENTAL OR CONSEQUENTIAL
 DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THIS SCRIPT (INCLUDING BUT
 NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES
 SUSTAINED BY THIRD PARTIES), EVEN IF YOU HAVE ADVISED US OF THE POSSIBILITY OF
 SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

NIM Piton proof
Matt Wilding 4-15-92

modified 7-92 to work on Piton library

modified 11-95 to use Nqthm rather than PC-Nqthm

This script takes 10 hours to run on a 64 meg Sparc2.

Nim is a game played with piles of matches.  Two opponents alternate
taking at least one match from exactly one pile until there are no
matches left.  The player who takes the last match loses.

Piton is an assembly-level language with a formal semantics and a
verified compiler.  Piton is described in CLI tech report #22.  One of
the machines to which Piton is targeted is FM9001, a microprocessor
that has a formal semantics and that has been fabricated.

This proof script leads NQTHM to a proof that a Piton program that
"plays" Nim does so optimally.  Informally, this means

  a) A Piton program (see function CM-PROG) when run on the Piton
  interpreter and given as input a reasonable Nim state yields a new
  Nim state equal to what is calculated by function COMPUTER-MOVE.
  (See event COMPUTER-MOVE-IMPLEMENTED.)
```

```
  b) COMPUTER-MOVE generates valid moves.  That is, it removes at
  least one match from exactly one pile.  (VALID-MOVEP-COMPUTER-MOVE)

  c) Depth-first search of the state space of all possible moves is
  used to define what is meant by optimal Nim play.  Exhaustive search
  is used to find if there is a strategy for a Nim player to ensure
  eventual victory from the current Nim state.  An optimal strategy
  transforms any state for which there exists such a winning strategy
  into a state from which exhaustive search can find no winning
  strategy.
     Exhaustive search of all possible moves from a NIM state is
  formalized in the function WSP.  The optimalilty of the strategy
  COMPUTER-MOVE is proved in the event COMPUTER-MOVE-WORKS.

  d) The FM9001 Piton compiler correctness theorem assumes that the
  Piton state that is to be run contains valid Piton code, fit into
  an FM9001's memory, and use constants of word size 32.  A Piton
  state (used in the Indiana test described below) was constructed
  that contains the Nim program, and is proved to meet the compiler
  correctness constraints (CM-PROG-FM9001-LOADABLE)

The algorithm used by the program is non-obvious and very efficient,
avoiding the need to search.  (I invented the programming trick only
to discover subsequently that it has been known since the beginning
of the century.)

This script was developed using only those events from the Piton
library necessary to define the interpreter and the naturals library.
In July 92 it was modified somewhat to "fit" onto the Piton library
that sits on top of the FM9001 library.  The immediate motivation was
to make it easier to include in the then-upcoming NQTHM-1992 proveall
release.  The Piton library contains the events of the FM9001 library.
The FM9001 library contains an older version of the naturals library
(though not explicitly with a note-lib).  Thus, this script requires
only the Piton library.

In May 92, in consultation with researchers at Indiana University, I
compiled the Nim Piton program for the FM9001 and sent the image to
Indiana.  They ran the image and generated an optimal NIM move on a
fabricated FM9001 they had wired up.  Subsequently, the CLI single
board computer was used to run this program.

A non-trivial program compiled using a reasonably-complex compiler for
a small microprocessor worked without any conventional testing done on
any of the components, and everyone was pleased but not surprised.

|#

;; We use the definitions and lemmas of the Piton proof, distributed
;; by Computational Logic, Inc. with their Nqthm-1992 release.

(note-lib "/slocal/src/nqthm-1992/examples/fm9001-piton/piton" t)

(set-status addition-on addition ((otherwise enable)))
(set-status multiplication-on multiplication ((otherwise enable)))
(set-status remainders-on remainders ((otherwise enable)))
(set-status quotients-on quotients ((otherwise enable)))
(set-status exponentiation-on exponentiation ((otherwise enable)))
(set-status logs-on logs ((otherwise enable)))
(set-status gcds-on gcds ((otherwise enable)))

(defn clock-plus (x y)
```

```
  (plus x y))

(prove-lemma p-add1 (rewrite)
             (equal
              (p p0 (add1 n))
              (p (p-step p0) n))
             ((disable p-step)))

(prove-lemma p-0 (rewrite)
             (implies
              (zerop n)
              (equal (p p0 n) p0)))

(prove-lemma clock-plus-function (rewrite)
             (equal
              (p p0 (clock-plus x y))
              (p (p p0 x) y))
             ((induct (p p0 x))
              (disable p p-step)))

(disable p-add1)

(prove-lemma clock-plus-add1 (rewrite)
             (equal
              (p p0 (clock-plus (add1 x) y))
              (p p0 (add1 (clock-plus x y)))))

(disable clock-plus)

(prove-lemma clock-plus-0 (rewrite)
             (implies
              (zerop x)
             (equal
              (clock-plus x y)
              (fix y)))
             ((enable clock-plus)))

(prove-lemma fix-clock-plus (rewrite)
             (equal
              (fix (clock-plus x y))
              (clock-plus x y))
             ((enable clock-plus)))

(prove-lemma p-step1-opener (rewrite)
  (equal (p-step1 (cons opcode operands) p)
         (if (p-ins-okp (cons opcode operands) p)
             (p-ins-step (cons opcode operands) p)
             (p-halt p (x-y-error-msg 'p opcode))))

  ((disable p-ins-okp p-ins-step)))

(disable p-step1)

(prove-lemma p-opener (rewrite)
  (and (equal (p s 0) s)
       (equal (p (p-state pc ctrl temp prog data max-ctrl max-temp word-size
                          psw)
                 (add1 n))
              (p (p-step (p-state pc ctrl temp prog data max-ctrl max-temp
                                  word-size psw))
                 n)))
  ((disable p-step)))
```

```
(disable p)

(defn at-least-morep (base delta value)
  (not (lessp value (plus base delta))))

(prove-lemma at-least-morep-normalize (rewrite)
             (and
              (equal
               (at-least-morep (add1 base) delta value)
               (at-least-morep base (add1 delta) value))
              (equal
               (at-least-morep base (add1 delta) (add1 value))
               (at-least-morep base delta value))))

(prove-lemma at-least-morep-linear (rewrite)
             (implies
              (and
               (at-least-morep base d1 value)
               (not (lessp d1 d2)))
              (at-least-morep base d2 value)))

(prove-lemma lessp-as-at-least-morep (rewrite)
             (implies
              (at-least-morep base delta value)
              (and
               (equal
                (lessp value x)
                (not (at-least-morep x 0 value)))
               (equal
                (lessp x value)
                (at-least-morep x 1 value)))))

(disable at-least-morep)

(defn nat-to-bv (nat size)
  (if (zerop size)
      nil
    (if (lessp nat (exp 2 (sub1 size)))
        (cons 0 (nat-to-bv nat (sub1 size)))
      (cons 1 (nat-to-bv (difference nat (exp 2 (sub1 size)))
                         (sub1 size))))))

(defn nat-to-bv-state (state size)
  (if (listp state)
      (cons (nat-to-bv (car state) size)
            (nat-to-bv-state (cdr state) size))
    nil))

;; a more elegant way to write this program would be to use the
;; bit-vector of all 0's initially, then xor all the elements of
;; the array.  But we don't want a pointer to memory to
;; ever have an improper value, so we write things this way

(defn xor-bvs-program nil
  '(xor-bvs (vecs-addr numvecs)
             nil
             (push-local vecs-addr)
             (fetch)
             (push-local numvecs)
             (sub1-nat)
             (pop-local numvecs)
```

```
      (dl loop ()
              (push-local numvecs))
              (test-nat-and-jump zero done)
              (push-local numvecs)
              (sub1-nat)
              (pop-local numvecs)
              (push-local vecs-addr)
              (push-constant (nat 1))
              (add-addr)
              (set-local vecs-addr)
              (fetch)
              (xor-bitv)
              (jump loop)
     (dl done ()
              (ret))))

(defn bit-vectors-piton (array size)
  (if (listp array)
      (and
       (listp (car array))
       (equal (caar array) 'bitv)
       (bit-vectorp (cadar array) size)
       (equal (cddar array) nil)
       (bit-vectors-piton (cdr array) size))
    (equal array nil)))

(defn array (name segment)
  (cdr (assoc name segment)))

;; vecs is name of state
(defn xor-bvs-input-conditionp (p0)
  (and
   (equal (car (top (p-temp-stk p0))) 'nat)
   (equal (car (top (cdr (p-temp-stk p0)))) 'addr)
   (equal (cdadr (top (cdr (p-temp-stk p0)))) 0)
   (listp (cadr (top (cdr (p-temp-stk p0)))))
   (equal (cddr (top (p-temp-stk p0))) nil)
   (equal (cddr (top (cdr (p-temp-stk p0)))) nil)
   (definedp (caadr (top (cdr (p-temp-stk p0)))) (p-data-segment p0))
   (bit-vectors-piton (array (caadr (top (cdr (p-temp-stk p0))))
                             (p-data-segment p0))
                      (p-word-size p0))
   (equal (cadr (top (p-temp-stk p0)))
          (length (array (caadr (top (cdr (p-temp-stk p0))))
                         (p-data-segment p0))))
   (at-least-morep (p-ctrl-stk-size (p-ctrl-stk p0))
                   4 (p-max-ctrl-stk-size p0))
   (at-least-morep (length (p-temp-stk p0))
                   2 (p-max-temp-stk-size p0))
   (not (zerop (untag (top (p-temp-stk p0)))))
   (lessp (untag (top (p-temp-stk p0))) (exp 2 (p-word-size p0)))
   (listp (p-ctrl-stk p0))))

; time to run loop
(defn xor-bvs-clock-loop (numvecs)
  (if (zerop numvecs)
      3
    (plus 12 (xor-bvs-clock-loop (sub1 numvecs)))))

; time to run xor-bvs, including call and ret

(defn xor-bvs-clock (numvecs)
```

```
  (plus 6 (xor-bvs-clock-loop (sub1 numvecs))))

(defn xor-bvs-array (current array n array-size)
  (if (zerop n)
      current
    (xor-bvs-array
     (xor-bitv current (untag (get (difference array-size n) array)))
     array (sub1 n) array-size)))

(prove-lemma lessp-1-exp (rewrite)
             (equal
              (lessp 1 (exp a b))
              (and (lessp 1 a) (not (zerop b))))
             ((enable exp)))

(prove-lemma bit-vectors-piton-means (rewrite)
             (implies
              (and
               (bit-vectors-piton state size)
               (lessp p (length state)))
              (and
               (equal (car (get p state)) 'bitv)
               (listp (get p state))
               (bit-vectorp (cadr (get p state)) size)
               (equal (cddr (get p state)) nil))))

(defn xor-bvs-loop-correctness-general-induct (i current n s data-segment)
  (if (zerop i) t
    (xor-bvs-loop-correctness-general-induct
     (sub1 i)
     (xor-bitv
      current
      (cadr (get (difference n i) (array s data-segment))))
     n s data-segment)))


(enable bit-vectorp-xor-bitv)

(prove-lemma xor-bvs-loop-correctness-general nil
             (implies
              (and
               (lessp (length (array s data-segment))
                      (exp 2 word-size))
               (not (zerop word-size))
               (listp ctrl-stk)
               (bit-vectors-piton (array s data-segment) word-size)
               (at-least-morep (length temp-stk) 3 max-temp-stk-size)
               (equal (definition 'xor-bvs prog-segment)
                      (xor-bvs-program))
               (definedp s data-segment)
               (numberp i)
               (lessp i n)
               (bit-vectorp current word-size)
               (equal n (length (array s data-segment))))
              (equal
               (p (p-state '(pc (xor-bvs . 5))
                           (cons (list
                                  (list
                                   (cons 'vecs-addr
                                         (list 'addr
                                               (cons
                                                s
```

```
                                        (sub1
                                         (difference n i)))))
                              (cons 'numvecs (list 'nat i)))
                             ret-pc)
                            ctrl-stk)
                      (cons (list 'bitv current)
                            temp-stk)
                      prog-segment
                      data-segment
                      max-ctrl-stk-size
                      max-temp-stk-size
                      word-size
                      'run)
               (xor-bvs-clock-loop i))
            (p-state ret-pc
                     ctrl-stk
                     (cons (list 'bitv (xor-bvs-array
                                        current
                                        (array s data-segment)
                                        i n))
                                  temp-stk)
                     prog-segment
                     data-segment
                     max-ctrl-stk-size
                     max-temp-stk-size
                     word-size
                     'run)))
             ((induct (xor-bvs-loop-correctness-general-induct
                      i current n s data-segment))))

(prove-lemma difference-x-sub1-x-better (rewrite)
             (equal (difference x (sub1 x))
                    (if (lessp 0 x) 1 0)))

(prove-lemma xor-bvs-loop-correctness nil
             (implies
              (and
               (lessp (length (array s data-segment))
                      (exp 2 word-size))
               (not (zerop word-size))
               (listp ctrl-stk)
               (bit-vectors-piton (array s data-segment) word-size)
               (at-least-morep (length temp-stk) 3 max-temp-stk-size)
               (equal (definition 'xor-bvs prog-segment)
                      (xor-bvs-program))
               (definedp s data-segment)
               (lessp 0 n)
               (bit-vectorp current word-size)
               (equal n (length (array s data-segment))))
              (equal
               (p (p-state '(pc (xor-bvs . 5))
                           (cons (list
                                  (list
                                   (cons 'vecs-addr
                                         (list 'addr (cons s 0)))
                                   (cons 'numvecs
                                         (list 'nat (sub1 n))))
                                  ret-pc)
                                 ctrl-stk)
                           (cons (list 'bitv current) temp-stk)
                           prog-segment
                           data-segment
```

```
                          max-ctrl-stk-size
                          max-temp-stk-size
                          word-size
                          'run)
                 (xor-bvs-clock-loop (sub1 n)))
              (p-state ret-pc
                       ctrl-stk
                       (cons (list 'bitv (xor-bvs-array
                                           current
                                           (array s data-segment)
                                           (sub1 n) n))
                                    temp-stk)
                       prog-segment
                       data-segment
                       max-ctrl-stk-size
                       max-temp-stk-size
                       word-size
                       'run)))
             ((use (xor-bvs-loop-correctness-general (i (sub1 n))))))

(prove-lemma exp-0 (rewrite)
             (implies
              (zerop x)
              (and (equal (exp x y) (if (zerop y) 1 0))
                   (equal (exp y x) 1)))
             ((enable exp)))

(prove-lemma bit-vectors-piton-means-more (rewrite)
             (implies
              (and
               (listp x)
               (bit-vectors-piton x size))
              (equal
               (list 'bitv (cadar x))
               (car x))))

;; xor-bvs of an array of at least one bit vector
(defn xor-bvs (array)
  (if (listp array)
      (xor-bitv (car array) (xor-bvs (cdr array)))
    nil))


(defn untag-array (array)
  (if (listp array)
      (cons (untag (car array))
            (untag-array (cdr array)))
    nil))

(prove-lemma bit-vectorp-get (rewrite)
             (implies
              (bit-vectors-piton array size)
              (equal
               (bit-vectorp (untag (get n array)) size)
               (lessp n (length array)))))

(enable difference-sub1-arg2)

(prove-lemma xor-bitv-commutative (rewrite)
             (implies
              (equal (length a) (length b))
              (equal (xor-bitv a b) (xor-bitv b a))))
```

```
(prove-lemma xor-bitv-commutative2 (rewrite)
             (implies
              (equal (length a) (length b))
              (equal (xor-bitv a (xor-bitv b c))
                     (xor-bitv b (xor-bitv a c)))))

(prove-lemma xor-bitv-associative (rewrite)
             (implies
              (equal (length a) (length b))
              (equal (xor-bitv (xor-bitv a b) c)
                     (xor-bitv a (xor-bitv b c)))))


(prove-lemma length-from-bit-vectorp (rewrite)
             (implies
              (bit-vectorp x s)
              (equal (length x) (fix s))))

(prove-lemma length-xor-bitv (rewrite)
             (equal
              (length (xor-bitv a b))
              (length a)))

(prove-lemma length-cadr-get-bit-vectors-piton (rewrite)
             (implies
              (and
               (bit-vectors-piton x l)
               (lessp i (length x)))
              (equal (length (cadr (get i x))) (fix l))))

(prove-lemma equal-xor-bitv-x-x (rewrite)
             (implies
              (and
               (bit-vectorp b (length a))
               (bit-vectorp c (length a)))
              (equal
               (equal (xor-bitv a b) (xor-bitv a c))
               (equal b c))))

(defn bit-vectorp-induct (size a b)
  (if (zerop size) t
    (bit-vectorp-induct (sub1 size) (cdr a) (cdr b))))

(prove-lemma bit-vectorp-xor-bitv2 (rewrite)
             (equal
              (bit-vectorp (xor-bitv a b) size)
              (equal (length a) (fix size)))
             ((induct (bit-vectorp-induct size a b))))


(defn bit-vectorsp (bvs size)
  (if (listp bvs)
      (and
       (bit-vectorp (car bvs) size)
       (bit-vectorsp (cdr bvs) size))
    (equal bvs nil)))

(prove-lemma length-xor-bvs (rewrite)
             (implies
              (bit-vectorsp bvs (length (car bvs)))
              (equal (length (xor-bvs bvs)) (length (car bvs)))))
```

```
(prove-lemma bit-vectorsp-untag (rewrite)
             (implies
              (bit-vectors-piton x s)
              (bit-vectorsp (untag-array x) s)))

(prove-lemma bit-vectorsp-cdr-untag (rewrite)
             (implies
              (bit-vectors-piton (cdr x) s)
              (bit-vectorsp (cdr (untag-array x)) s)))

(enable nthcdr)

(prove-lemma bit-vectorsp-nthcdr (rewrite)
             (implies
              (and
               (bit-vectorsp x s)
               (lessp n (length x)))
              (bit-vectorsp (nthcdr n x) s))
             ((enable nthcdr)))

(prove-lemma bit-vectorp-xor-bvs (rewrite)
             (implies
              (and
               (bit-vectorsp x size)
               (listp x))
              (bit-vectorp (xor-bvs x) size)))

(prove-lemma length-untag-array (rewrite)
             (equal (length (untag-array x)) (length x)))

(enable listp-nthcdr)

(prove-lemma nthcdr-open (rewrite)
             (implies
              (lessp n (length x))
              (equal
               (nthcdr n x)
               (cons (get n x) (nthcdr (add1 n) x))))
             ((enable nthcdr)))

(prove-lemma get-untag-array (rewrite)
             (implies
              (lessp n (length x))
              (equal
               (get n (untag-array x))
               (cadr (get n x)))))


(prove-lemma equal-xor-bitv-x-x-special (rewrite)
             (implies
              (and
               (bit-vectorp b (length a))
               (bit-vectorp (xor-bitv z c) (length a)))
              (equal
               (equal (xor-bitv a b) (xor-bitv z (xor-bitv a c)))
               (equal b (xor-bitv z c)))))

(defn fix-bit (b)
  (if (equal b 0) 0 1))

(prove-lemma xor-bitv-0 (rewrite)
```

```
             (and
              (equal (xor-bit x 0) (fix-bit x))
              (equal (xor-bit 0 x) (fix-bit x))))

(prove-lemma xor-bitv-nlistp (rewrite)
             (implies
              (not (listp c))
              (equal (xor-bitv a (xor-bitv b c))
                     (xor-bitv a b))))

(prove-lemma xor-bitv-nlistp2 (rewrite)
             (implies
              (and
               (bit-vectorp a b)
               (not (listp c)))
              (equal (xor-bitv a c) a)))

(prove-lemma xor-bvs-array-rewrite (rewrite)
             (implies
              (and
               (bit-vectors-piton array (length current))
               (bit-vectorp current (length current))
               (lessp n (length array))
               (equal (length array) as))
             (equal
              (xor-bvs-array current array n as)
              (xor-bitv current
                        (xor-bvs
                         (nthcdr (difference as n)
                                 (untag-array array))))))
             ((induct (xor-bvs-array current array n as))
              (enable nthcdr)))

(enable equal-length-0)

(prove-lemma correctness-of-xor-bvs-helper
 (rewrite)
 (implies
  (and (equal p0
              (p-state pc ctrl-stk
                       (append (list (tag 'nat numvecs)
                                     (tag 'addr (cons state 0)))
                               temp-stk)
                       prog-segment data-segment max-ctrl-stk-size
                       max-temp-stk-size word-size 'run))
       (equal (p-current-instruction p0)
              '(call xor-bvs))
       (equal (definition 'xor-bvs prog-segment)
              (xor-bvs-program))
       (xor-bvs-input-conditionp p0))
  (equal
   (p p0 (xor-bvs-clock numvecs))
   (p-state (add1-addr pc)
            ctrl-stk
            (cons (list 'bitv
                        (xor-bvs-array (untag (car (array state data-segment)))
                                       (array state data-segment)
                                       (sub1 numvecs)
                                       numvecs))
                  temp-stk)
            prog-segment data-segment max-ctrl-stk-size max-temp-stk-size
            word-size 'run)))
```

```
 ((use
   (xor-bvs-loop-correctness (current (untag (car (array state data-segment))))
                             (s state)
                             (n numvecs)
                             (ret-pc (add-addr pc 1))))))

(prove-lemma length-cadar-bvs (rewrite)
             (implies
              (and
               (bit-vectors-piton x s)
               (listp x))
              (equal (length (cadar x)) (fix s))))

(prove-lemma bit-vectorp-from-bit-vectors-piton (rewrite)
             (implies
              (bit-vectors-piton x s)
              (and
               (equal
                (bit-vectorp (cadar x) s)
                (listp x))
               (equal
                (bit-vectorp (cadr (get n x)) s)
                (lessp n (length x))))))

(prove-lemma nthcdr-1 (rewrite)
             (equal
              (nthcdr 1 a)
              (cdr a))
             ((enable nthcdr)))

(prove-lemma listp-untag-array (rewrite)
             (equal
              (listp (untag-array x))
              (listp x)))

(prove-lemma xor-bvs-input-conditionp-means-xor-bvs-hack (rewrite)
             (implies
              (and
               (xor-bvs-input-conditionp
                          (p-state
                          pc
                          ctrl-stk
                          (cons (list 'nat numvecs)
                                (cons (list 'addr (cons state 0))
                                      temp-stk))
                          prog-segment
                          data-segment
                          max-ctrl-stk-size
                          max-temp-stk-size
                          word-size
                          'run))
               (lessp 0 word-size))
              (equal
               (xor-bvs-array
                (untag
                 (car (array state data-segment)))
                (array state data-segment)
                (sub1 numvecs) numvecs)
               (xor-bvs (untag-array (array state data-segment)))))
             ((enable nthcdr)))

(prove-lemma correctness-of-xor-bvs (rewrite)
```

```
          (implies
           (and
            (equal p0 (p-state
                       pc
                       ctrl-stk
                       (cons n (cons s temp-stk))
                       prog-segment
                       data-segment
                       max-ctrl-stk-size
                       max-temp-stk-size
                       word-size
                       'run))
            (lessp 0 word-size)
            (equal (p-current-instruction p0) '(call xor-bvs))
            (equal (definition 'xor-bvs prog-segment)
                   (xor-bvs-program))
            (xor-bvs-input-conditionp p0))
           (equal
            (p (p-state
                       pc
                       ctrl-stk
                       (cons n (cons s temp-stk))
                       prog-segment
                       data-segment
                       max-ctrl-stk-size
                       max-temp-stk-size
                       word-size
                       'run)
               (xor-bvs-clock (cadr n)))
            (p-state (add1-addr pc)
                     ctrl-stk
                     (cons (list 'bitv
                                 (xor-bvs (untag-array
                                           (array (caadr s) data-segment))))
                           temp-stk)
                     prog-segment
                     data-segment
                     max-ctrl-stk-size
                     max-temp-stk-size
                     word-size
                     'run)))
          ((disable xor-bvs-clock)
           (use (correctness-of-xor-bvs-helper
                 (state (caadr s)) (numvecs (cadr n)))
                (xor-bvs-input-conditionp-means-xor-bvs-hack
                 (state (caadr s)) (numvecs (cadr n))))))

(defn example-xor-bvs-p-state ()
  (p-state '(pc (main . 0))
           '((nil (pc (main . 0))))
           nil
           (list '(main nil nil
                        (push-constant (addr (arr . 0)))
                        (push-constant (nat 3))
                        (call xor-bvs)
                        (ret))
                 (xor-bvs-program))
           '((arr (bitv (0 1 0 1 1 0 0 1))
                  (bitv (0 0 0 0 0 0 0 1))
                  (bitv (0 1 1 0 1 0 0 1))))
           10
           8
```

```
          8
          'run))

;;;; push-1-vector
;; Piton currently does not provide any mechanism for creating a
;; bit vector except as an operation on other bit vectors.  Until
;; this apparent flaw is fixed, we'll write our program as a
;; function of the word size.

(defn one-bit-vector (wordsize)
  (if (lessp wordsize 2)
      (list 1)
    (cons 0 (one-bit-vector (sub1 wordsize)))))

(defn push-1-vector-program (wordsize)
  (list 'push-1-vector nil nil
        (list 'push-constant (list 'bitv (one-bit-vector wordsize)))
        (list 'ret)))


(defn example-push-1-vector-state ()
  (p-state '(pc (main . 0))
           '((nil (pc (main . 0))))
           nil
           (list '(main nil nil
                        (call push-1-vector)
                        (ret))
                 (push-1-vector-program 8))
           nil
           10
           8
           8
           'run))

(defn push-1-vector-input-conditionp (p0)
  (and
   (not (lessp (p-max-ctrl-stk-size p0)
               (plus 2 (p-ctrl-stk-size (p-ctrl-stk p0)))))
   (not (lessp (p-max-temp-stk-size p0)
               (plus 1 (length (p-temp-stk p0)))))
   (listp (p-ctrl-stk p0))))

(enable length-append)

(prove-lemma equal-assoc-cons (rewrite)
             (implies
              (equal (assoc k a) (cons x y))
              (and
               (equal (car (assoc k a)) x)
               (equal (cdr (assoc k a)) y))))

(prove-lemma correctness-of-push-1-vector (rewrite)
             (implies
              (and
               (equal p0 (p-state
                          pc
                          ctrl-stk
                          temp-stk
                          prog-segment
                          data-segment
                          max-ctrl-stk-size
                          max-temp-stk-size
```

```
                          word-size
                          'run))
               (equal (p-current-instruction p0)
                      '(call push-1-vector))
               (equal (definition 'push-1-vector prog-segment)
                      (push-1-vector-program word-size))
               (push-1-vector-input-conditionp p0))
              (equal
               (p p0 3)
               (p-state (add1-addr pc)
                        ctrl-stk
                        (cons (list 'bitv (one-bit-vector word-size))
                              temp-stk)
                        prog-segment
                        data-segment
                        max-ctrl-stk-size
                        max-temp-stk-size
                        word-size
                        'run))))

;;;; nat-to-bv

(defn nat-to-bv-program nil
  '(nat-to-bv (value)
              ((current-bit (nat 0)) (temp (nat 0)))
              (call push-1-vector)
              (pop-local current-bit)
              (call push-1-vector)
              (rsh-bitv)
         (dl loop ()
              (push-local value))
              (test-nat-and-jump zero done)
              (push-local value)
              (div2-nat)
              (pop-local temp)
              (pop-local value)
              (push-local temp)
              (test-nat-and-jump zero lab)
              (push-local current-bit)
              (xor-bitv)
         (dl lab ()
              (push-local current-bit))
              (lsh-bitv)
              (pop-local current-bit)
              (jump loop)
         (dl done ()
              (ret))))

(defn example-nat-to-bv-state ()
  (p-state '(pc (main . 0))
           '((nil (pc (main . 0))))
           nil
           (list '(main nil nil
                        (push-constant (nat 86))
                        (call nat-to-bv)
                        (ret))
                 (push-1-vector-program 8)
                 (nat-to-bv-program))
           nil
           10
           8
           8
```

```
           'run))

(defn zero-bit-vector (size)
  (if (zerop size) nil
    (cons 0 (zero-bit-vector (sub1 size)))))


(defn nat-to-bv2-helper (value current-bit bit-vector)
  (if (zerop value)
      bit-vector
    (nat-to-bv2-helper (quotient value 2)
                       (append (cdr current-bit) (list 0))
                       (if (equal (remainder value 2) 1)
                           (xor-bitv current-bit bit-vector)
                         bit-vector))))

(defn nat-to-bv2 (value size)
  (nat-to-bv2-helper value (one-bit-vector size)
                     (zero-bit-vector size)))

(defn nat-to-bv-loop-clock (value)
  (if (zerop value)
      3
    (plus
     (if (equal (remainder value 2) 0) 12 14)
     (nat-to-bv-loop-clock (quotient value 2)))))

(defn correctness-of-nat-to-bv-general-induct (value cb temp bv)
  (if (zerop value) t
    (correctness-of-nat-to-bv-general-induct
     (quotient value 2)
     (append (cdr cb) (list 0))
     (list 'nat (remainder value 2))
     (if (equal (remainder value 2) 0) bv (xor-bitv bv cb)))))

(prove-lemma lessp-remainder-simple (rewrite)
             (implies
              (not (zerop y))
              (lessp (remainder x y) y)))

(prove-lemma lessp-exp-simple (rewrite)
             (implies
              (and
               (lessp x y)
               (not (zerop z)))
              (lessp x (exp y z)))
             ((enable exp)))

(prove-lemma lessp-1 (rewrite)
             (equal
              (lessp x 1)
              (zerop x)))

(prove-lemma lessp-remainder-x-exp-x (rewrite)
             (equal
              (lessp (remainder x y) (exp y z))
              (or
               (and
                (zerop z)
                (equal (remainder x y) 0))
               (and
                (not (zerop z))
```

```
               (not (zerop y)))))
            ((enable exp)))

(prove-lemma bit-vectorp-append (rewrite)
             (implies
              (bit-vectorp x x-size)
              (equal
               (bit-vectorp (append x y) size)
               (and
                (bit-vectorp y (difference size x-size))
                (not (lessp size x-size))))))

(prove-lemma correctness-of-nat-to-bv-general (rewrite)
             (implies
              (and
               (listp ctrl-stk)
               (at-least-morep (length temp-stk)
                               3 max-temp-stk-size)
               (equal (definition 'nat-to-bv prog-segment)
                      (nat-to-bv-program))
               (numberp value)
               (lessp 0 word-size)
               (lessp value (exp 2 word-size))
               (bit-vectorp bv word-size)
               (bit-vectorp cb word-size))
              (equal
               (p (p-state '(pc (nat-to-bv . 4))
                           (cons (list
                                  (list
                                   (cons 'value (list 'nat value))
                                   (cons 'current-bit (list 'bitv cb))
                                   (cons 'temp temp))
                                  ret-pc)
                                 ctrl-stk)
                           (cons (list 'bitv bv)
                                 temp-stk)
                           prog-segment
                           data-segment
                           max-ctrl-stk-size
                           max-temp-stk-size
                           word-size
                           'run)
                  (nat-to-bv-loop-clock value))
               (p-state ret-pc
                        ctrl-stk
                        (cons (list 'bitv (nat-to-bv2-helper
                                           value cb bv))
                              temp-stk)
                        prog-segment
                        data-segment
                        max-ctrl-stk-size
                        max-temp-stk-size
                        word-size
                        'run)))
              ((induct (correctness-of-nat-to-bv-general-induct
                        value cb temp bv))))

(defn nat-to-bv-clock (value)
  (plus 9 (nat-to-bv-loop-clock value)))

(defn nat-to-bv-input-conditionp (p0)
  (and
```

```
   (lessp (cadr (top (p-temp-stk p0))) (exp 2 (p-word-size p0)))
   (numberp (cadr (top (p-temp-stk p0))))
   (at-least-morep (p-ctrl-stk-size (p-ctrl-stk p0))
                   7 (p-max-ctrl-stk-size p0))
   (lessp 0 (p-word-size p0))
   (at-least-morep (length (p-temp-stk p0))
                   2 (p-max-temp-stk-size p0))
   (listp (p-ctrl-stk p0))))

(prove-lemma bit-vectorp-one-bit-vector (rewrite)
             (equal
              (bit-vectorp (one-bit-vector s) s)
              (lessp 0 s)))

(prove-lemma bit-vectorp-zero-bit-vector (rewrite)
             (bit-vectorp (zero-bit-vector s) s))

(prove-lemma all-but-last-one-bit-vector (rewrite)
             (equal
              (all-but-last (one-bit-vector s))
              (zero-bit-vector (sub1 s))))

(prove-lemma correctness-of-nat-to-bv-helper nil
             (implies
              (and
               (equal p0 (p-state
                          pc
                          ctrl-stk
                          (cons (list 'nat value) temp-stk)
                          prog-segment
                          data-segment
                          max-ctrl-stk-size
                          max-temp-stk-size
                          word-size
                          'run))
               (equal (p-current-instruction p0) '(call nat-to-bv))
               (equal (definition 'nat-to-bv prog-segment)
                      (nat-to-bv-program))
               (equal (definition 'push-1-vector prog-segment)
                      (push-1-vector-program word-size))
               (nat-to-bv-input-conditionp p0))
              (equal
               (p p0 (nat-to-bv-clock value))
               (p-state (add1-addr pc)
                        ctrl-stk
                        (cons (list 'bitv
                                    (nat-to-bv2 value word-size))
                              temp-stk)
                        prog-segment
                        data-segment
                        max-ctrl-stk-size
                        max-temp-stk-size
                        word-size
                        'run))))

(defn bv-to-nat (bv)
  (if (listp bv)
      (plus (times (fix-bit (car bv)) (exp 2 (length (cdr bv))))
            (bv-to-nat (cdr bv)))
    0))

(prove-lemma length-nat-to-bv (rewrite)
```

```
             (equal (length (nat-to-bv nat size)) (fix size)))


(prove-lemma bv-to-nat-nat-to-bv (rewrite)
              (equal (bv-to-nat (nat-to-bv nat size))
                     (if (lessp nat (exp 2 size))
                         (fix nat)
                       (sub1 (exp 2 size))))
              ((enable exp)))

(enable exp)

(prove-lemma nat-to-bv-simple (rewrite)
             (implies
              (zerop x)
              (and
               (equal (nat-to-bv x y) (zero-bit-vector y))
               (equal (nat-to-bv y x) nil))))

(prove-lemma nat-to-bv-bv-to-nat (rewrite)
             (implies
              (bit-vectorp bv size)
              (equal (nat-to-bv (bv-to-nat bv) size) bv)))

(prove-lemma bv-to-nat-append (rewrite)
             (equal
              (bv-to-nat (append x y))
              (plus
               (bv-to-nat y)
               (times (exp 2 (length y)) (bv-to-nat x)))))

(prove-lemma lessp-plus-hacks (rewrite)
             (and
             (equal
              (lessp
               (plus a (plus b (plus c (plus (times d e) (times d e)))))
               (plus e e))
              (and
               (zerop d)
               (lessp (plus a (plus b c)) (plus e e))))
             (equal
              (lessp
               (plus a (plus b (plus c (plus e (plus (times d g) h)))))
               g)
              (and
               (zerop d)
               (lessp (plus a (plus b (plus c (plus e h)))) g)))))

(prove-lemma bv-to-nat-xor-bitv (rewrite)
             (implies
              (and
               (bit-vectorp x size)
               (bit-vectorp y size)
               (equal (and-bitv x y) (zero-bit-vector size)))
              (equal
               (bv-to-nat (xor-bitv x y))
               (plus (bv-to-nat x) (bv-to-nat y)))))

(prove-lemma lessp-bv-to-nat-exp-2 (rewrite)
             (implies
              (bit-vectorp x size)
              (equal (lessp (bv-to-nat x) (exp 2 size)) t)))
```

```
(prove-lemma equal-nat-to-bv (rewrite)
             (implies
              (and
               (bit-vectorp bv size)
               (lessp y (exp 2 size)))
              (and
               (equal
                (equal (nat-to-bv y size) bv)
                (equal (bv-to-nat bv) (fix y)))
               (equal
                (equal bv (nat-to-bv y size))
                (equal (fix y) (bv-to-nat bv))))))

(defn least-bit-higher-than-high-bit (x y)
  (if (listp x)
      (and
       (equal (length x) (length y))
       (if (not (equal (car y) 0))
           (all-zero-bitvp x)
         (least-bit-higher-than-high-bit (cdr x) (cdr y))))
    (nlistp y)))

(prove-lemma least-bit-higher-means-and-0 (rewrite)
             (implies
              (and
               (bit-vectorp x size)
               (bit-vectorp y size)
               (least-bit-higher-than-high-bit x y))
              (and
               (equal (and-bitv x y) (zero-bit-vector size))
               (equal (and-bitv y x) (zero-bit-vector size)))))

(prove-lemma all-zero-bitvp-zero-bit-vector (rewrite)
             (all-zero-bitvp (zero-bit-vector size)))

(prove-lemma bv-to-nat-one-bit-vector (rewrite)
             (equal (bv-to-nat (one-bit-vector size)) 1))

;; renamed from firstn to avoid conflict with similar but different
;; definition in piton library
(defn make-list-from (n list)
  (if (zerop n)
      nil
    (cons (car list) (make-list-from (sub1 n) (cdr list)))))

(prove-lemma length-make-list-from (rewrite)
             (equal (length (make-list-from n list)) (fix n)))

(prove-lemma make-list-from-1 (rewrite)
             (equal
              (make-list-from 1 x)
              (list (car x))))

(prove-lemma listp-make-list-from (rewrite)
             (equal
              (listp (make-list-from n x))
              (not (zerop n))))

(defn double-cdr-induct (x z)
  (if (listp x)
      (double-cdr-induct (cdr x) (cdr z))
```

```
   t))


(prove-lemma all-zero-bitvp-append (rewrite)
             (equal
              (all-zero-bitvp (append x y))
              (and
               (all-zero-bitvp x)
               (all-zero-bitvp y))))


(prove-lemma least-bit-higher-than-high-bit-append-0s (rewrite)
             (implies
              (and
               (all-zero-bitvp y)
               (equal (length z) (length (append x y))))
             (equal
              (least-bit-higher-than-high-bit (append x y) z)
              (least-bit-higher-than-high-bit
               x (make-list-from (length x) z))))
             ((induct (double-cdr-induct x z))))

(prove-lemma equal-xor-bitv-x-y-1 (rewrite)
             (equal
              (equal (xor-bitv a b) b)
              (and
               (bit-vectorp b (length a))
               (all-zero-bitvp a))))

(prove-lemma equal-xor-bitv-x-y-2 (rewrite)
             (equal
              (equal (xor-bitv a b) a)
              (and
               (all-zero-bitvp (make-list-from (length a) b))
               (bit-vectorp a (length a)))))

(prove-lemma least-bit-higher-than-high-bit-simple (rewrite)
             (implies
              (all-zero-bitvp x)
              (equal
               (least-bit-higher-than-high-bit x y)
               (equal (length x) (length y)))))

(prove-lemma make-list-from-is-all-but-last (rewrite)
             (implies
              (equal (length x) (add1 n))
              (equal
               (make-list-from n x)
               (all-but-last x))))

(prove-lemma least-bit-higher-all-but-last (rewrite)
             (implies
              (least-bit-higher-than-high-bit a b)
              (equal
               (least-bit-higher-than-high-bit
                a (cons 0 (all-but-last b)))
               (listp a))))

(defn at-most-one-bit-on (bv)
  (if (listp bv)
      (if (equal (car bv) 0)
          (at-most-one-bit-on (cdr bv))
```

```
        (all-zero-bitvp (cdr bv)))
    t))

(enable length-all-but-last)

(prove-lemma listp-xor-bitv (rewrite)
             (equal
              (listp (xor-bitv a b))
              (listp a)))

(prove-lemma all-zero-bitvp-means-at-most-one-bit-on (rewrite)
             (implies
              (all-zero-bitvp x)
              (at-most-one-bit-on x)))

(prove-lemma at-most-one-bit-on-append (rewrite)
             (equal
              (at-most-one-bit-on (append a b))
              (or
               (and
                (all-zero-bitvp a)
                (at-most-one-bit-on b))
               (and
                (at-most-one-bit-on a)
                (all-zero-bitvp b)))))

(defn fix-bitv (list)
  (if (listp list)
      (cons (if (equal (car list) 0) 0 1) (fix-bitv (cdr list)))
    nil))

(prove-lemma xor-bitv-nlistp3 (rewrite)
             (implies
              (not (listp x))
              (and
               (equal (xor-bitv x y) nil)
               (equal (xor-bitv y x) (fix-bitv y)))))

(disable xor-bitv-nlistp)
(disable xor-bitv-nlistp2)

(prove-lemma fix-bitv-all-but-last (rewrite)
             (equal
              (fix-bitv (all-but-last x))
              (all-but-last (fix-bitv x))))

(prove-lemma all-but-last-xor-bitv (rewrite)
             (equal
              (all-but-last (xor-bitv a b))
              (if (lessp (length b) (length a))
                  (xor-bitv (all-but-last a) b)
                (xor-bitv (all-but-last a) (all-but-last b)))))

(prove-lemma least-bit-higher-cons-xor-bitv-hack (rewrite)
             (implies
              (and
               (least-bit-higher-than-high-bit (cdr x) a)
               (least-bit-higher-than-high-bit (cdr x) b))
              (equal
               (least-bit-higher-than-high-bit
                x (cons 0 (xor-bitv a b)))
               (listp x))))
```

```
(prove-lemma least-bit-higher-cdr-all-but-last (rewrite)
             (implies
              (least-bit-higher-than-high-bit a b)
              (least-bit-higher-than-high-bit (cdr a) (all-but-last b))))

(prove-lemma least-bit-higher-x-all-but-last-x (rewrite)
             (implies
              (at-most-one-bit-on x)
              (least-bit-higher-than-high-bit
               (cdr x) (all-but-last x))))

(prove-lemma nat-to-bv-equiv-helper nil
             (implies
              (and
               (bit-vectorp cb size)
               (bit-vectorp bv size)
               (lessp 0 size)
               (lessp (plus (bv-to-nat bv)
                            (times (bv-to-nat cb) value))
                      (exp 2 size))
               (at-most-one-bit-on cb)
               (least-bit-higher-than-high-bit cb bv))
             (equal
              (nat-to-bv2-helper value cb bv)
              (nat-to-bv (plus (times (bv-to-nat cb) value)
                               (bv-to-nat bv))
                         size)))
             ((induct (nat-to-bv2-helper value cb bv))))

(prove-lemma at-most-one-bit-on-one-bit-vector (rewrite)
             (at-most-one-bit-on (one-bit-vector size)))

(prove-lemma bv-to-nat-all-zero-bitvp (rewrite)
             (implies
              (all-zero-bitvp x)
              (equal (bv-to-nat x) 0)))


(prove-lemma least-bit-higher-than-high-bit-simple2 (rewrite)
             (implies
              (all-zero-bitvp x)
              (equal
               (least-bit-higher-than-high-bit y x)
               (equal (length x) (length y)))))

(prove-lemma length-zero-bit-vector (rewrite)
             (equal (length (zero-bit-vector size)) (fix size)))

(prove-lemma length-one-bit-vector (rewrite)
             (equal (length (one-bit-vector size))
                    (if (zerop size) 1 size)))

(prove-lemma nat-to-bv-equivalence (rewrite)
             (implies
              (and
               (lessp value (exp 2 size))
               (lessp 0 size))
             (equal
              (nat-to-bv2 value size)
              (nat-to-bv value size)))
             ((use (nat-to-bv-equiv-helper
```

```
                    (cb (one-bit-vector size))
                    (bv (zero-bit-vector size))))))

(prove-lemma correctness-of-nat-to-bv (rewrite)
             (implies
              (and
               (equal p0 (p-state
                          pc
                          ctrl-stk
                          (cons v temp-stk)
                          prog-segment
                          data-segment
                          max-ctrl-stk-size
                          max-temp-stk-size
                          word-size
                          'run))
               (equal (car v) 'nat)
               (equal (cddr v) nil)
               (equal (p-current-instruction p0) '(call nat-to-bv))
               (equal (definition 'nat-to-bv prog-segment)
                      (nat-to-bv-program))
               (equal (definition 'push-1-vector prog-segment)
                      (push-1-vector-program word-size))
               (equal num (cadr v))
               (nat-to-bv-input-conditionp p0))
              (equal
               (p (p-state
                          pc
                          ctrl-stk
                          (cons v temp-stk)
                          prog-segment
                          data-segment
                          max-ctrl-stk-size
                          max-temp-stk-size
                          word-size
                          'run)
                  (nat-to-bv-clock num))
               (p-state (add1-addr pc)
                        ctrl-stk
                        (cons (list 'bitv
                                    (nat-to-bv (cadr v) word-size))
                              temp-stk)
                        prog-segment
                        data-segment
                        max-ctrl-stk-size
                        max-temp-stk-size
                        word-size
                        'run)))
             ((disable-theory t)
              (enable nat-to-bv-input-conditionp p-state p-word-size-p-state
                      p-temp-stk-p-state top)
              (enable-theory ground-zero)
              (use (correctness-of-nat-to-bv-helper (value (cadr v)))
                   (nat-to-bv-equivalence (size word-size)
                                          (value (cadr v))))))

;;;; bv-to-nat

(defn bv-to-nat-program nil
  '(bv-to-nat (bv)
             ((current-bit (nat 0)) (current-2power (nat 0)))
              (push-constant (nat 1))
```

```
              (pop-local current-2power)
              (call push-1-vector)
              (pop-local current-bit)
              (push-constant (nat 0))
           (dl loop ()
              (push-local bv))
              (push-local current-bit)
              (and-bitv)
              (test-bitv-and-jump all-zero lab)
              (push-local current-2power)
              (add-nat)
           (dl lab ()
              (push-local current-bit))
              (lsh-bitv)
              (set-local current-bit)
              (test-bitv-and-jump all-zero done)
              (push-local current-2power)
              (mult2-nat)
              (pop-local current-2power)
              (jump loop)
           (dl done ()
              (ret))))

(defn example-bv-to-nat-state ()
  (p-state '(pc (main . 0))
           '((nil (pc (main . 0))))
           nil
           (list '(main nil nil
                        (push-constant (bitv (1 0 1 1 0 0 0 1)))
                        (call bv-to-nat)
                        (ret))
                 (push-1-vector-program 8)
                 (bv-to-nat-program))
           nil
           10
           8
           8
           'run))

(defn trailing-zeros-helper (list acc)
  (if (listp list)
      (trailing-zeros-helper
       (cdr list) (if (equal (car list) 0) (add1 acc) 0))
    (fix acc)))

(defn trailing-zeros (list)
  (trailing-zeros-helper list 0))

(prove-lemma trailing-zeros-of-all-zero-bitvp (rewrite)
             (implies
              (all-zero-bitvp bv)
              (equal (trailing-zeros-helper bv n)
                     (plus (length bv) n))))

(prove-lemma non-zero-means-acc-irrelevant-spec (rewrite)
             (implies
              (not (all-zero-bitvp bv))
              (equal (trailing-zeros-helper bv (add1 x))
                     (trailing-zeros-helper bv x))))

(prove-lemma non-zero-means-acc-irrelevant (rewrite)
             (implies
```

```
               (and
                (not (all-zero-bitvp bv))
                (not (equal x 0)))
               (equal (trailing-zeros-helper bv x)
                      (trailing-zeros-helper bv 0))))

(prove-lemma trailing-zeros-helper-append (rewrite)
             (equal
              (trailing-zeros-helper (append x y) acc)
              (if (all-zero-bitvp y)
                  (plus (trailing-zeros-helper x acc) (length y))
                (trailing-zeros-helper y acc))))

(prove-lemma trailing-zeros-append (rewrite)
             (equal
              (trailing-zeros (append x y))
              (if (all-zero-bitvp y)
                  (plus (trailing-zeros x) (length y))
                (trailing-zeros y))))

(prove-lemma lessp-trailing-zeros-helper (rewrite)
             (implies
              (not (lessp n (plus acc (length x))))
              (equal (lessp n (trailing-zeros-helper x acc)) f)))

(defn last (list)
  (if (listp list)
      (if (listp (cdr list))
          (last (cdr list))
        (car list))
    0))

(prove-lemma equal-trailing-zeros-helper-0 (rewrite)
             (equal
              (equal (trailing-zeros-helper x acc) 0)
              (or (and (zerop acc) (nlistp x))
                  (not (equal (last x) 0)))))

(prove-lemma lessp-length-cdr-trailing (rewrite)
             (implies
              (and
               (lessp (length (cdr x)) (trailing-zeros-helper x acc))
               (listp x))
              (and
               (all-zero-bitvp x)
               (all-zero-bitvp (cdr x)))))

(prove-lemma not-all-zero-bitvp-cdr-means (rewrite)
             (implies
              (not (all-zero-bitvp (cdr x)))
              (equal
               (trailing-zeros-helper x acc)
               (trailing-zeros-helper (cdr x) 0))))

(defn bv-to-nat2-helper (bv cb current-2power)
  (if (all-zero-bitvp cb)
      0
    (plus
     (if (all-zero-bitvp (and-bitv bv cb)) 0 current-2power)
     (bv-to-nat2-helper
      bv (append (cdr cb) (list 0)) (times 2 current-2power))))
  ((lessp (difference (length cb) (trailing-zeros cb)))))
```

```
(defn bv-to-nat2 (bv)
  (bv-to-nat2-helper bv (one-bit-vector (length bv)) 1))

(prove-lemma lessp-length-trailing-zeros-hack (rewrite)
             (implies
              (zerop acc)
              (not (lessp (length x) (trailing-zeros-helper x acc)))))

(defn bv-to-nat-loop-clock (cb bv)
  (if (all-zero-bitvp (cdr cb))
      (if (or (equal (car cb) 0) (equal (car bv) 0)) 9 11)
    (plus (if (all-zero-bitvp (and-bitv cb bv)) 12 14)
          (bv-to-nat-loop-clock (append (cdr cb) '(0)) bv)))
  ((lessp (difference (length cb) (trailing-zeros cb)))))

(defn bv-to-nat-induct (value bv cb current-2power)
  (if (all-zero-bitvp cb)
      0
     (bv-to-nat-induct
      (plus (if (all-zero-bitvp (and-bitv bv cb)) 0 current-2power)
            value)
      bv (append (cdr cb) (list 0)) (times 2 current-2power)))
  ((lessp (difference (length cb) (trailing-zeros cb)))))

(defn double-cdr-with-sub1-induct (x y n)
  (if (listp x)
      (double-cdr-with-sub1-induct (cdr x) (cdr y) (sub1 n))
    t))

(prove-lemma bit-vectorp-and-bitv-better (rewrite)
             (equal
              (bit-vectorp (and-bitv x y) size)
              (equal (length x) (fix size)))
             ((induct (double-cdr-with-sub1-induct x y size))))

(prove-lemma lessp-bv-to-nat-exp (rewrite)
             (implies
              (bit-vectorp x size)
              (lessp (bv-to-nat x) (exp 2 size))))

(prove-lemma equal-bv-to-nat-0 (rewrite)
             (implies
              (bit-vectorp x size)
             (equal
              (equal (bv-to-nat x) 0)
              (all-zero-bitvp x))))

(enable commutativity-of-and-bitv)

(prove-lemma commutativity2-of-and-bitv (rewrite)
             (implies
              (equal (length x) (length y))
              (equal (and-bitv x (and-bitv y z))
                     (and-bitv y (and-bitv x z)))))

(prove-lemma associativity-of-and-bitv (rewrite)
             (implies
              (equal (length x) (length y))
             (equal (and-bitv (and-bitv x y) z)
                    (and-bitv x (and-bitv y z)))))
```

```
(prove-lemma all-zero-bitvp-and-bitv (rewrite)
             (implies
              (all-zero-bitvp x)
              (and
               (all-zero-bitvp (and-bitv x y))
               (all-zero-bitvp (and-bitv y x)))))

(prove-lemma bv-to-nat-loop-clock-open (rewrite)
             (implies
              (all-zero-bitvp x)
              (equal (bv-to-nat-loop-clock x y) 9)))

(prove-lemma bv-to-nat2-helper-hack (rewrite)
             (implies
              (and
               (all-zero-bitvp z)
               (equal (length d) (length z)))
              (equal (bv-to-nat2-helper (cons 1 d) (cons 1 z) v)
                     (fix v)))
             ((expand (bv-to-nat2-helper (cons 1 d) (cons 1 z) v))))

(prove-lemma bv-to-nat2-helper-hack2 (rewrite)
             (implies
              (and
               (all-zero-bitvp z)
               (equal (length d) (length z)))
              (equal (bv-to-nat2-helper (cons 0 d) (cons 1 z) v) 0))
             ((expand (bv-to-nat2-helper (cons 0 d) (cons 1 z) v))))

(prove-lemma correctness-of-bv-to-nat-general (rewrite)
             (implies
              (and
               (listp ctrl-stk)
               (at-least-morep (length temp-stk)
                               3 max-temp-stk-size)
               (equal (definition 'bv-to-nat prog-segment)
                      (bv-to-nat-program))
               (numberp c2p)
               (lessp 0 word-size)
               (at-most-one-bit-on cb)
               (equal c2p (bv-to-nat cb))
               (bit-vectorp bv word-size)
               (bit-vectorp cb word-size)
               (numberp value)
               (lessp value c2p))
              (equal
               (p (p-state '(pc (bv-to-nat . 5))
                           (cons (list
                                  (list
                                   (cons 'bv (list 'bitv bv))
                                   (cons 'current-bit (list 'bitv cb))
                                   (cons 'current-2power
                                         (list 'nat c2p)))
                                  ret-pc)
                                 ctrl-stk)
                           (cons (list 'nat value)
                                 temp-stk)
                           prog-segment
                           data-segment
                           max-ctrl-stk-size
                           max-temp-stk-size
                           word-size
```

```
                          'run)
                   (bv-to-nat-loop-clock cb bv))
                (p-state ret-pc
                         ctrl-stk
                         (cons
                          (list 'nat
                                (plus (bv-to-nat2-helper bv cb c2p)
                                      value))
                               temp-stk)
                         prog-segment
                         data-segment
                         max-ctrl-stk-size
                         max-temp-stk-size
                         word-size
                         'run)))
               ((induct (bv-to-nat-induct value bv cb c2p))))


(defn bv-to-nat-clock (word-size bv)
  (plus 8 (bv-to-nat-loop-clock (one-bit-vector word-size) bv)))

(defn bv-to-nat-input-conditionp (p0)
  (and
   (equal (car (top (p-temp-stk p0))) 'bitv)
   (equal (cddr (top (p-temp-stk p0))) nil)
   (bit-vectorp (cadr (top (p-temp-stk p0))) (p-word-size p0))
   (at-least-morep (p-ctrl-stk-size (p-ctrl-stk p0))
                   7 (p-max-ctrl-stk-size p0))
   (lessp 0 (p-word-size p0))
   (at-least-morep (length (p-temp-stk p0))
                   2 (p-max-temp-stk-size p0))
   (listp (p-ctrl-stk p0))))

(prove-lemma correctness-of-bv-to-nat-helper nil
             (implies
              (and
               (equal p0 (p-state
                           pc
                           ctrl-stk
                           (cons (list 'bitv bv) temp-stk)
                           prog-segment
                           data-segment
                           max-ctrl-stk-size
                           max-temp-stk-size
                           word-size
                           'run))
               (equal (p-current-instruction p0) '(call bv-to-nat))
               (equal (definition 'bv-to-nat prog-segment)
                      (bv-to-nat-program))
               (equal (definition 'push-1-vector prog-segment)
                      (push-1-vector-program word-size))
               (bv-to-nat-input-conditionp p0))
              (equal
               (p p0 (bv-to-nat-clock word-size bv))
               (p-state (add1-addr pc)
                         ctrl-stk
                         (cons (list 'nat (bv-to-nat2 bv)) temp-stk)
                         prog-segment
                         data-segment
                         max-ctrl-stk-size
                         max-temp-stk-size
                         word-size
```

```
                        'run))))

(prove-lemma bit-vectorp-append-better (rewrite)
             (implies
              (bit-vectorp x (length x))
              (equal (bit-vectorp (append x y) size)
                     (and
                      (bit-vectorp y (length y))
                      (equal (fix size)
                             (plus (length x) (length y))))))
              ((induct (make-list-from size x))))

(prove-lemma bit-vectorp-hack (rewrite)
             (equal
              (bit-vectorp x (plus a (length (cdr x))))
              (and
               (if (listp x) (equal a 1) (zerop a))
               (bit-vectorp x (length x))))
             ((use (length-from-bit-vectorp
                    (x x) (s (plus a (length (cdr x))))))
              (disable length-from-bit-vectorp)))

(prove-lemma bv-to-nat-one-bit (rewrite)
             (implies
              (and
               (not (all-zero-bitvp x))
               (at-most-one-bit-on x)
               (bit-vectorp x size))
              (equal (bv-to-nat x) (exp 2 (trailing-zeros x)))))

(prove-lemma lessp-sub1-plus-hack (rewrite)
             (equal
              (lessp (sub1 x) (plus y x))
              (or (not (zerop x)) (not (zerop y)))))

(prove-lemma quotient-plus-hack (rewrite)
             (equal
              (quotient (plus a b b) 2)
              (plus (quotient a 2) b)))

(prove-lemma bv-to-nat-all-but-last (rewrite)
             (implies
              (bit-vectorp x size)
             (equal
              (bv-to-nat (all-but-last x))
              (quotient (bv-to-nat x) 2))))

(prove-lemma make-list-from-cons (rewrite)
             (equal
              (make-list-from n (cons a b))
              (if (zerop n) nil (cons a (make-list-from (sub1 n) b)))))

(prove-lemma equal-plus-times-hack (rewrite)
             (equal
              (equal (plus a (times a b) (times a c)) (times a d))
              (or (zerop a) (equal (plus 1 b c) d)))
             ((use (equal-times-arg1 (a a) (x (plus 1 b c)) (y d)))))

(enable nth)

(prove-lemma last-make-list-from (rewrite)
             (equal (last (make-list-from n x))
```

```
                 (if (zerop n) 0 (nth (sub1 n) x))))

(prove-lemma make-list-from-nlistp (rewrite)
             (implies
              (nlistp x)
              (equal (make-list-from n x) (zero-bit-vector n))))

(prove-lemma nth-nlistp (rewrite)
             (implies
              (nlistp x)
              (equal (nth n x) 0)))

(prove-lemma bit-vectorp-make-list-from (rewrite)
             (implies
              (bit-vectorp x size)
              (bit-vectorp (make-list-from n x) n)))

(prove-lemma equal-bv-to-nat-0-2 (rewrite)
             (implies
              (bit-vectorp x (length x))
              (equal
               (equal (bv-to-nat x) 0)
               (all-zero-bitvp x))))

(prove-lemma bv-to-nat-make-list-from-from-sub1-make-list-from (rewrite)
             (implies
              (equal (bv-to-nat (make-list-from (sub1 n) v)) 0)
              (equal (bv-to-nat (make-list-from n v))
                     (if (zerop n) 0 (fix-bit (nth (sub1 n) v))))))

(prove-lemma plus-bv-to-nat-make-list-from (rewrite)
             (equal
              (plus (bv-to-nat (make-list-from (sub1 z) v))
                    (bv-to-nat (make-list-from (sub1 z) v)))
              (difference (bv-to-nat (make-list-from z v))
                          (fix-bit (last (make-list-from z v)))))
             ((induct (make-list-from z v))))

(prove-lemma equal-bv-to-nat-1 (rewrite)
             (implies
              (bit-vectorp x (length x))
             (equal
              (equal (bv-to-nat x) 1)
              (and
               (all-zero-bitvp (all-but-last x))
               (equal (last x) 1))))
             ((induct (all-but-last x))))

(prove-lemma lessp-1-hack (rewrite)
             (equal
              (lessp 1 a)
              (and
               (not (zerop a))
               (not (equal a 1)))))

(prove-lemma not-equal-nth-0-means (rewrite)
             (implies
              (and
               (not (equal (nth n v) 0))
               (lessp n s))
              (not (all-zero-bitvp (make-list-from s v)))))
```

```
(prove-lemma all-zero-bitvp-all-but-last-means nil
             (implies
              (and
               (all-zero-bitvp (all-but-last x))
               (equal (length x) (length y))
               (lessp 0 (trailing-zeros-helper y acc)))
              (all-zero-bitvp (and-bitv x y))))

(prove-lemma all-zero-bitvp-all-but-last-means-spec (rewrite)
             (implies
              (and
               (all-zero-bitvp (all-but-last x))
               (equal (length x) (length y))
               (lessp 0 (trailing-zeros-helper y 0)))
              (all-zero-bitvp (and-bitv x y)))
             ((use (all-zero-bitvp-all-but-last-means (acc 0)))))

(prove-lemma all-zero-means-to-and-bitv (rewrite)
             (implies
              (all-zero-bitvp x)
              (and
               (equal (and-bitv x y) (zero-bit-vector (length x)))
               (equal (and-bitv y x) (zero-bit-vector (length y))))))

(prove-lemma trailing-zeros-nth-proof nil
             (implies
              (and
               (equal n
                      (sub1 (difference (length x)
                                        (trailing-zeros-helper x acc))))
               (not (all-zero-bitvp x)))
              (not (equal (nth n x) 0))))


(prove-lemma trailing-zeros-nth-spec (rewrite)
             (implies
              (and
               (equal n
                      (sub1 (difference (length x)
                                        (trailing-zeros-helper x 0))))
               (not (all-zero-bitvp x)))
              (not (equal (nth n x) 0)))
             ((use (trailing-zeros-nth-proof (acc 0)))))

(prove-lemma and-bitv-special (rewrite)
             (implies
              (and
               (equal (nth n w) 0)
               (not (equal (nth n x) 0))
               (equal (length x) (length x))
               (at-most-one-bit-on x))
              (equal (and-bitv w x) (zero-bit-vector (length w)))))


(prove-lemma equal-trailing-zeros-length-spec nil
             (equal
              (equal (trailing-zeros-helper x acc) (plus acc (length x)))
              (all-zero-bitvp x)))

(prove-lemma equal-trailing-zeros-length (rewrite)
             (implies
              (zerop acc)
```

```
             (equal
              (equal (trailing-zeros-helper x acc) (length x))
              (all-zero-bitvp x)))
             ((use (equal-trailing-zeros-length-spec))))

(prove-lemma bit-vectorp-trailing-zeros (rewrite)
             (equal
              (bit-vectorp x (trailing-zeros-helper x acc))
              (and
               (all-zero-bitvp x)
               (bit-vectorp x (length x))
               (zerop acc)))
             ((use (equal-trailing-zeros-length-spec)
                   (length-from-bit-vectorp
                    (s (trailing-zeros-helper x acc))))
              (disable length-from-bit-vectorp)))


(prove-lemma quotient-exp-hack (rewrite)
             (implies
              (not (zerop b))
             (equal (quotient (plus x (exp b y)) b)
                    (if (zerop y)
                        (quotient (add1 x) b)
                      (plus (exp b (sub1 y)) (quotient x b))))))

(prove-lemma bit-vectorp-means-properp (rewrite)
             (implies
              (bit-vectorp x (length x))
              (properp x)))

(prove-lemma make-list-from-simplify (rewrite)
             (implies
              (and
               (equal n (length x))
               (properp x))
              (equal (make-list-from n x) x)))

(prove-lemma equal-add1-plus-hack (rewrite)
             (and
              (equal
               (equal (add1 (plus a b)) (plus c (plus d a)))
               (equal (add1 b) (plus c d)))
              (equal
               (equal (add1 (plus a b)) (plus c a))
               (equal (add1 b) (fix c)))
              (equal
               (equal (add1 a) (plus b a))
               (equal 1 (fix b)))))

(prove-lemma plus-quotient-bv-to-nat (rewrite)
             (equal
              (plus (quotient (bv-to-nat x) 2)
                    (quotient (bv-to-nat x) 2))
              (difference (bv-to-nat x) (fix-bit (last x)))))

(prove-lemma equal-plus-times-hack2 (rewrite)
             (equal
              (equal (plus (times a b) (times a c)) (times a d))
              (or (zerop a)
                  (equal (plus b c) (fix d))))
             ((use (equal-times-arg1 (a a) (x (plus b c)) (y d)))))
```

```
(prove-lemma and-bitv-special-special (rewrite)
             (implies
              (and
               (equal (last w) 0)
               (not (equal (last x) 0))
               (equal (length x) (length w))
               (at-most-one-bit-on x))
              (equal (and-bitv w x) (zero-bit-vector (length w)))))

(prove-lemma and-bitv-special-3 (rewrite)
             (implies
              (and
               (equal (length x) (length y))
               (not (equal (last x) 0))
               (not (equal (last y) 0)))
              (not (all-zero-bitvp (and-bitv x y)))))

(prove-lemma and-bitv-special-4 (rewrite)
             (implies
              (and
               (not (equal (nth n w) 0))
               (not (equal (nth n x) 0))
               (equal (length x) (length x)))
               (not (all-zero-bitvp (and-bitv w x)))))

(prove-lemma and-bitv-special-5 (rewrite)
             (implies
              (and
               (equal (last w) 0)
               (not (equal (last x) 0))
               (equal (add1 (length x)) (length w))
               (at-most-one-bit-on x))
              (equal (and-bitv w (cons 0 x))
                     (zero-bit-vector (length w)))))

(prove-lemma and-bitv-special-6 (rewrite)
             (implies
              (and
               (not (equal (last w) 0))
               (not (equal (last x) 0))
               (equal (add1 (length x)) (length w)))
              (not (all-zero-bitvp (and-bitv w (cons 0 x))))))

(prove-lemma not-last-0-means-not-all-0 (rewrite)
             (implies
              (not (equal (last x) 0))
              (not (all-zero-bitvp x))))

(prove-lemma bit-vectorp-plus-length-hack (rewrite)
             (and
             (equal
              (bit-vectorp x (plus z (length x)))
              (and
               (bit-vectorp x (length x))
               (zerop z)))
             (equal
              (bit-vectorp x (plus (add1 z) (length (cdr x))))
              (and
               (bit-vectorp x (length x))
               (listp x)
               (zerop z)))))
```

```
(prove-lemma last-append (rewrite)
             (equal
              (last (append x y))
              (if (listp y) (last y) (last x))))

(prove-lemma bv-to-nat2-helper-bv-to-nat (rewrite)
             (implies
              (and
               (at-most-one-bit-on cb)
               (bit-vectorp cb size)
               (bit-vectorp bv size)
               (equal c2 (bv-to-nat cb)))
              (equal
               (bv-to-nat2-helper bv cb c2)
               (bv-to-nat (append
                           (make-list-from (difference (length bv)
                                                       (trailing-zeros cb))
                                   bv)
                           (zero-bit-vector (trailing-zeros cb)))))))

(prove-lemma bv-to-nat2-helper-bv-to-nat-better (rewrite)
             (implies
              (and
               (at-most-one-bit-on cb)
               (bit-vectorp cb (length cb))
               (bit-vectorp bv (length cb))
               (equal c2 (bv-to-nat cb)))
              (equal
               (bv-to-nat2-helper bv cb c2)
               (bv-to-nat (append
                           (make-list-from (difference (length bv)
                                                       (trailing-zeros cb))
                                   bv)
                           (zero-bit-vector (trailing-zeros cb)))))))

(prove-lemma nlistp-bv-to-nat2 (rewrite)
             (implies
              (not (listp x))
              (equal (bv-to-nat2-helper x a b) 0)))

(prove-lemma bv-to-nat2-bv-to-nat-helper nil
             (implies (nlistp x)
                      (equal (bv-to-nat2 x) (bv-to-nat x))))

(prove-lemma bit-vectorp-one-bit-vector-rewrite (rewrite)
             (equal
              (bit-vectorp (one-bit-vector s) n)
              (if (zerop s) (equal n 1) (equal (fix s) (fix n))))
             ((induct (lessp s n))))

(prove-lemma trailing-zeros-helper-one-bit-vector (rewrite)
             (equal
              (trailing-zeros-helper (one-bit-vector n) acc)
              0))

(prove-lemma bv-to-nat2-bv-to-nat (rewrite)
             (implies
              (bit-vectorp x (length x))
              (equal (bv-to-nat2 x) (bv-to-nat x)))
             ((use (bv-to-nat2-bv-to-nat-helper))))
```

```
(prove-lemma bv-length-weaker (rewrite)
             (implies
              (bit-vectorp x s)
              (bit-vectorp x (length x))))

(prove-lemma correctness-of-bv-to-nat (rewrite)
             (implies
              (and
               (equal p0 (p-state
                          pc
                          ctrl-stk
                          (cons b temp-stk)
                          prog-segment
                          data-segment
                          max-ctrl-stk-size
                          max-temp-stk-size
                          word-size
                          'run))
               (equal (p-current-instruction p0) '(call bv-to-nat))
               (equal (definition 'bv-to-nat prog-segment)
                      (bv-to-nat-program))
               (equal (definition 'push-1-vector prog-segment)
                      (push-1-vector-program word-size))
               (equal bv (cadr b))
               (bv-to-nat-input-conditionp p0))
              (equal
               (p (p-state
                          pc
                          ctrl-stk
                          (cons b temp-stk)
                          prog-segment
                          data-segment
                          max-ctrl-stk-size
                          max-temp-stk-size
                          word-size
                          'run)
                  (bv-to-nat-clock word-size bv))
               (p-state (add1-addr pc)
                        ctrl-stk
                        (cons (list 'nat (bv-to-nat bv)) temp-stk)
                        prog-segment
                        data-segment
                        max-ctrl-stk-size
                        max-temp-stk-size
                        word-size
                        'run)))
             ((use (correctness-of-bv-to-nat-helper)
                   (bv-to-nat2-bv-to-nat (x bv)))
              (disable-theory t)
              (enable bv-to-nat-input-conditionp p-state p-word-size-p-state bv-length-weaker
                      top p-temp-stk-p-state)
              (enable-theory ground-zero)))

;;;; number-with-at-least


(defn number-with-at-least-program nil
  '(number-with-at-least (nums-addr numnums min) ((i (nat 0)))
            (push-constant (nat 0))
            (set-local i)
     (dl loop ()
            (push-local nums-addr))
```

```
           (fetch)
           (push-local min)
           (lt-nat)
           (test-bool-and-jump t lab)
           (add1-nat)
     (dl lab ()
           (push-local numnums))
           (push-local i)
           (add1-nat)
           (set-local i)
           (sub-nat)
           (test-nat-and-jump zero done)
           (push-local nums-addr)
           (push-constant (nat 1))
           (add-addr)
           (pop-local nums-addr)
           (jump loop)
      (dl done ()
          (ret))))

(defn example-number-with-at-least-state ()
  (p-state '(pc (main . 0))
          '((nil (pc (main . 0))))
          nil
          (list '(main nil nil
                       (push-constant (addr (nums . 0)))
                       (push-constant (nat 5))
                       (push-constant (nat 3))
                       (call number-with-at-least)
                       (ret))
                (number-with-at-least-program))
          '((nums (nat 3) (nat 8) (nat 9) (nat 2) (nat 100)))
          10
          8
          8
          'run))

(defn number-with-at-least (numlist min)
  (if (listp numlist)
      (if (lessp (car numlist) min)
          (number-with-at-least (cdr numlist) min)
        (add1 (number-with-at-least (cdr numlist) min)))
    0))

(defn nat-list-piton (array word-size)
  (if (listp array)
      (and
       (listp (car array))
       (equal (caar array) 'nat)
       (numberp (cadar array))
       (equal (cddar array) nil)
       (lessp (cadar array) (exp 2 word-size))
       (nat-list-piton (cdr array) word-size))
    (equal array nil)))

(defn number-with-at-least-general-induct (i current n s min data-segment)
  (if (lessp i n)
      (number-with-at-least-general-induct
       (add1 i)
       (if (lessp (cadr (get i (array s data-segment))) min)
           current (add1 current))
       n s min data-segment)
```

```
    t)
  ((lessp (difference n i))))

(defn number-with-at-least-clock-loop (i min array)
  (if (not (lessp i (length array))) 0
    (plus
     (if (lessp (cadr (get i array)) min) 0 1)
     (if (equal (add1 i) (length array))
         12
       (plus 16 (number-with-at-least-clock-loop (add1 i) min array)))))
  ((lessp (difference (length array) i))))

(prove-lemma equal-difference-1 (rewrite)
             (equal
              (equal (difference x y) 1)
              (equal x (add1 y))))


(prove-lemma nat-list-piton-means (rewrite)
             (implies
              (and
               (nat-list-piton state size)
               (lessp p (length state)))
              (and
               (equal (car (get p state)) 'nat)
               (listp (get p state))
               (numberp (cadr (get p state)))
               (lessp (cadr (get p state)) (exp 2 size))
               (equal (cddr (get p state)) nil))))

(prove-lemma number-with-at-least-nlistp (rewrite)
             (implies
              (not (listp x))
              (equal (number-with-at-least x min) 0)))

(prove-lemma equal-add1-length (rewrite)
             (equal
              (equal (add1 x) (length y))
              (and
               (listp y)
               (equal (fix x) (length (cdr y))))))

(disable number-with-at-least-clock-loop)


(prove-lemma length-cdr-untag-array (rewrite)
             (equal
              (length (cdr (untag-array x)))
              (length (cdr x))))

(prove-lemma number-with-at-least-correctness-general nil
             (implies
              (and
               (lessp (length (array s data-segment))
                      (exp 2 word-size))
               (not (zerop word-size))
               (listp ctrl-stk)
               (nat-list-piton (array s data-segment) word-size)
               (at-least-morep (length temp-stk)
                               3 max-temp-stk-size)
               (equal (definition 'number-with-at-least prog-segment)
                      (number-with-at-least-program))
```

```
               (definedp s data-segment)
               (lessp min (exp 2 word-size))
               (numberp min)
               (lessp i n)
               (numberp i)
               (lessp n (exp 2 word-size))
               (numberp current)
               (not (lessp i current))
               (equal n (length (array s data-segment))))
              (equal
               (p (p-state '(pc (number-with-at-least . 2))
                           (cons (list
                                  (list
                                   (cons 'nums-addr
                                         (list 'addr (cons s i)))
                                   (cons 'numnums (list 'nat n))
                                   (cons 'min (list 'nat min))
                                   (cons 'i (list 'nat i)))
                                  ret-pc)
                                 ctrl-stk)
                           (cons (list 'nat current)
                                 temp-stk)
                           prog-segment
                           data-segment
                           max-ctrl-stk-size
                           max-temp-stk-size
                           word-size
                           'run)
                  (number-with-at-least-clock-loop
                   i min (array s data-segment)))
               (p-state ret-pc
                        ctrl-stk
                        (cons
                         (list 'nat
                               (plus current
                               (number-with-at-least
                                (nthcdr i (untag-array
                                           (array s data-segment)))
                                min)))
                         temp-stk)
                        prog-segment
                        data-segment
                        max-ctrl-stk-size
                        max-temp-stk-size
                        word-size
                        'run)))
             ((induct (number-with-at-least-general-induct
                        i current n s min data-segment))
              (expand (number-with-at-least-clock-loop
                        i min (cdr (assoc s data-segment))))))

(defn number-with-at-least-clock (min array)
  (plus 3 (number-with-at-least-clock-loop 0 min array)))

(defn number-with-at-least-input-conditionp (p0)
  (and
   (equal (car (car (p-temp-stk p0))) 'nat)
   (equal (cddr (car (p-temp-stk p0))) nil)
   (equal (car (cadr (p-temp-stk p0))) 'nat)
   (equal (cddr (cadr (p-temp-stk p0))) nil)
   (equal (car (caddr (p-temp-stk p0))) 'addr)
   (equal (cdadr (caddr (p-temp-stk p0))) 0)
```

```
   (equal (cddr (caddr (p-temp-stk p0))) nil)
   (listp (cadr (caddr (p-temp-stk p0))))
   (lessp (length (array (car (cadr (caddr (p-temp-stk p0))))
                         (p-data-segment p0)))
          (exp 2 (p-word-size p0)))
   (not (zerop (p-word-size p0)))
   (listp (p-ctrl-stk p0))
   (nat-list-piton (array (car (cadr (caddr (p-temp-stk p0))))
                          (p-data-segment p0))
                   (p-word-size p0))
   (at-least-morep (p-ctrl-stk-size (p-ctrl-stk p0))
                   6 (p-max-ctrl-stk-size p0))
   (at-least-morep (length (p-temp-stk p0))
                   0 (p-max-temp-stk-size p0))
   (equal (definition 'number-with-at-least (p-prog-segment p0))
          (number-with-at-least-program))
   (definedp (car (cadr (caddr (p-temp-stk p0)))) (p-data-segment p0))
   (lessp (cadr (car (p-temp-stk p0))) (exp 2 (p-word-size p0)))
   (numberp (cadr (car (p-temp-stk p0))))
   (lessp 0 (cadr (cadr (p-temp-stk p0))))
   (equal (cadr (cadr (p-temp-stk p0)))
          (length (array (car (cadr (caddr (p-temp-stk p0))))
                         (p-data-segment p0))))))

(prove-lemma correctness-of-number-with-at-least (rewrite)
             (implies
              (and
               (equal p0 (p-state
                          pc
                          ctrl-stk
                          (cons m (cons n (cons s temp-stk)))
                          prog-segment
                          data-segment
                          max-ctrl-stk-size
                          max-temp-stk-size
                          word-size
                          'run))
               (equal (p-current-instruction p0)
                      '(call number-with-at-least))
               (number-with-at-least-input-conditionp p0)
               (equal minc (cadr m))
               (equal arrayc (array (car (cadr s)) data-segment)))
              (equal
               (p (p-state
                          pc
                          ctrl-stk
                          (cons m (cons n (cons s temp-stk)))
                          prog-segment
                          data-segment
                          max-ctrl-stk-size
                          max-temp-stk-size
                          word-size
                          'run)
                  (number-with-at-least-clock minc arrayc))
               (p-state (add1-addr pc)
                        ctrl-stk
                        (cons (list 'nat
                                    (number-with-at-least
                                     (untag-array
                                      (array (car (cadr s))
                                             data-segment))
                                     (cadr m)))
```

```
                              temp-stk)
                        prog-segment
                        data-segment
                        max-ctrl-stk-size
                        max-temp-stk-size
                        word-size
                        'run)))
              ((use (number-with-at-least-correctness-general
                     (i 0) (current 0) (s (car (cadr s)))
                    (ret-pc (add1-addr pc))
                    (n (cadr n)) (min (cadr m))))
               (disable number-with-at-least-clock-loop)))

;; highest-bit


(defn highest-bit-program nil
  '(highest-bit (bv) ((cb (nat 0)))
            (call push-1-vector)
            (set-local cb)
            (rsh-bitv)
     (dl loop ()
            (push-local cb))
            (test-bitv-and-jump all-zero done)
            (push-local bv)
            (push-local cb)
            (and-bitv)
            (test-bitv-and-jump all-zero lab)
            (pop)
            (push-local cb)
       (dl lab ()
            (push-local cb))
            (lsh-bitv)
            (pop-local cb)
            (jump loop)
       (dl done ()
            (ret))))

(defn example-highest-bit-state ()
  (p-state '(pc (main . 0))
           '((nil (pc (main . 0))))
           nil
           (list '(main nil nil
                        (push-constant (bitv (0 0 0 0 0 0 0 0)))
                        (call highest-bit)
                        (push-constant (bitv (0 0 1 1 0 1 0 0)))
                        (call highest-bit)
                        (push-constant (bitv (1 0 1 1 0 1 0 0)))
                        (call highest-bit)
                        (ret))
                 (highest-bit-program)
                 (push-1-vector-program 8))
           nil
           10
           8
           8
           'run))

(defn highest-bit (x)
  (if (listp x)
      (if (equal (car x) 0)
          (cons 0 (highest-bit (cdr x)))
```

```
        (cons 1 (zero-bit-vector (length (cdr x)))))
    nil))

(prove-lemma listp-highest-bit (rewrite)
             (equal (listp (highest-bit x)) (listp x)))

(prove-lemma length-highest-bit (rewrite)
             (equal (length (highest-bit x)) (length x)))

(defn highest-bit-loop-clock (cb bv)
  (if (all-zero-bitvp cb)
      3
    (plus
     10 (if (all-zero-bitvp (and-bitv cb bv)) 0 2)
     (highest-bit-loop-clock (append (cdr cb) '(0)) bv)))
  ((lessp (difference (length cb) (trailing-zeros cb)))))

(defn highest-bit-induct (current bv cb)
  (if (all-zero-bitvp cb)
      t
    (highest-bit-induct
     (if (all-zero-bitvp (and-bitv cb bv)) current cb)
     bv
     (append (cdr cb) '(0))))
  ((lessp (difference (length cb) (trailing-zeros cb)))))

(prove-lemma bit-vectorp-simple-not (rewrite)
             (implies
              (not (equal (length x) (fix y)))
              (not (bit-vectorp x y))))

(prove-lemma at-most-one-bit-on-cdr (rewrite)
             (implies
              (at-most-one-bit-on x)
              (at-most-one-bit-on (cdr x))))


(prove-lemma equal-one-bit-vector (rewrite)
             (equal
              (equal x (one-bit-vector z))
              (or
               (and
                (zerop z)
                (equal x '(1)))
               (and
                (bit-vectorp x z)
                (all-zero-bitvp (all-but-last x))
                (not (equal (last x) 0))))))

(prove-lemma at-most-one-bit-is-all-zeros (rewrite)
             (implies
              (not (equal (last x) 0))
              (equal
               (at-most-one-bit-on x)
               (all-zero-bitvp (all-but-last x)))))

(defn highest-bit2-helper (current cb bv)
  (if (all-zero-bitvp cb)
      current
    (highest-bit2-helper
     (if (all-zero-bitvp (and-bitv bv cb)) current cb)
     (append (cdr cb) '(0))
```

```
    bv))
  ((lessp (difference (length cb) (trailing-zeros cb)))))

(prove-lemma equal-x-zero-bit-vector (rewrite)
             (equal
              (equal x (zero-bit-vector y))
              (and
               (all-zero-bitvp x)
               (bit-vectorp x y))))

(prove-lemma all-zero-bitvp-all-but-last-simple (rewrite)
             (implies
              (all-zero-bitvp x)
              (all-zero-bitvp (all-but-last x))))

(prove-lemma equal-trailing-zeros-acc-irrelevant (rewrite)
             (equal
              (equal (trailing-zeros-helper x acc1)
                     (trailing-zeros-helper x acc2))
              (or
               (equal (fix acc1) (fix acc2))
               (not (all-zero-bitvp x)))))

(prove-lemma lessp-trailing-zeros (rewrite)
             (implies
              (not (all-zero-bitvp z))
              (lessp (trailing-zeros-helper z acc)
                     (length z))))

(prove-lemma nthcdr-append (rewrite)
             (equal (nthcdr n (append x y))
                    (if (lessp (length x) n)
                        (nthcdr (difference n (length x)) y)
                      (append (nthcdr n x) y))))

(prove-lemma listp-zero-bit-vector (rewrite)
             (equal
              (listp (zero-bit-vector x))
              (not (zerop x))))

(prove-lemma and-bitv-append (rewrite)
             (equal
              (and-bitv (append x y) z)
              (append
               (and-bitv x (make-list-from (length x) z))
               (and-bitv y (nthcdr (length x) z))))
             ((induct (double-cdr-induct x z))))

(prove-lemma and-bitv-append2 (rewrite)
             (implies
              (equal (length (append x y)) (length z))
             (equal
              (and-bitv z (append x y))
              (append
               (and-bitv x (make-list-from (length x) z))
               (and-bitv y (nthcdr (length x) z)))))
             ((induct (double-cdr-induct x z))))

(prove-lemma all-zero-bitvp-and-bitv-append (rewrite)
             (equal
              (all-zero-bitvp (and-bitv z (append x y)))
              (and
```

```
               (all-zero-bitvp (and-bitv (make-list-from (length x) z) x))
               (all-zero-bitvp (and-bitv (nthcdr (length x) z) y))))
             ((induct (double-cdr-induct x z))))

(prove-lemma length-cdr-zero-bit-vector (rewrite)
             (equal
              (length (cdr (zero-bit-vector x)))
              (sub1 x)))

(prove-lemma length-nthcdr (rewrite)
             (equal
              (length (nthcdr n x))
              (difference (length x) n)))

(disable and-bitv-special)
(disable and-bitv-special-special)
(disable and-bitv-special-3)
(disable and-bitv-special-4)
(disable and-bitv-special-5)
(disable and-bitv-special-6)


(prove-lemma bit-vectorp-zero-bit-vector-better (rewrite)
             (equal
              (bit-vectorp (zero-bit-vector x) size)
              (equal (fix x) (fix size))))

(prove-lemma highest-bit-correctness-general nil
             (implies
              (and
               (not (zerop word-size))
               (at-most-one-bit-on cb)
               (listp ctrl-stk)
               (at-least-morep (length temp-stk)
                               3 max-temp-stk-size)
               (equal (definition 'highest-bit prog-segment)
                      (highest-bit-program))
               (bit-vectorp bv word-size)
               (bit-vectorp cb word-size)
               (bit-vectorp current word-size))
              (equal
               (p (p-state '(pc (highest-bit . 3))
                           (cons (list
                                  (list
                                   (cons 'bv (list 'bitv bv))
                                   (cons 'cb (list 'bitv cb)))
                                  ret-pc)
                                 ctrl-stk)
                           (cons (list 'bitv current) temp-stk)
                           prog-segment
                           data-segment
                           max-ctrl-stk-size
                           max-temp-stk-size
                           word-size
                           'run)
                  (highest-bit-loop-clock cb bv))
               (p-state ret-pc
                        ctrl-stk
                        (cons
                         (list 'bitv
                               (highest-bit2-helper current cb bv))
                         temp-stk)
```

```
                       prog-segment
                       data-segment
                       max-ctrl-stk-size
                       max-temp-stk-size
                       word-size
                       'run)))
             ((induct (highest-bit-induct current bv cb))))

(prove-lemma make-list-from-simple (rewrite)
             (implies
              (zerop n)
              (equal (make-list-from n x) nil)))

(prove-lemma not-all-zero-bitvp-make-list-from (rewrite)
             (implies
              (and
               (all-zero-bitvp (make-list-from n1 x))
               (not (lessp n1 n2)))
              (all-zero-bitvp (make-list-from n2 x))))

(disable nthcdr-open)

(enable listp-append)

(prove-lemma highest-bit2-helper-cons-1 (rewrite)
             (implies
              (and
               (not (all-zero-bitvp cb))
               (at-most-one-bit-on cb)
               (equal (length cb) (length bv))
               (bit-vectorp cb (length cb))
               (equal (car bv) 1))
             (equal
              (highest-bit2-helper current cb bv)
              (cons 1 (zero-bit-vector (length (cdr bv)))))))

(prove-lemma bit-vectorp-append-cdr-hack (rewrite)
             (implies
              (bit-vectorp x n)
              (equal (bit-vectorp (append (cdr x) '(0)) n)
                     (listp x))))

(defn triple-cdr-with-sub1-induct (x y z n)
  (if (zerop n) t
    (triple-cdr-with-sub1-induct (cdr x) (cdr y) (cdr z) (sub1 n))))


(prove-lemma car-append-better (rewrite)
             (equal
              (car (append x y))
              (if (listp x)
                  (car x)
                (car y))))

(prove-lemma open-highest-when-and-not-0 (rewrite)
             (implies
              (and
               (not (all-zero-bitvp (and-bitv x y)))
               (equal (length x) (length y)))
              (equal (highest-bit2-helper c x y)
                     (highest-bit2-helper x (append (cdr x) '(0)) y))))
```

```
(enable cdr-append)

(prove-lemma highest-bit2-helper-cons-0 (rewrite)
             (implies
              (and
               (not (all-zero-bitvp cb))
               (at-most-one-bit-on cb)
               (bit-vectorp cb size)
               (bit-vectorp bv size)
               (bit-vectorp current size)
               (equal (car bv) 0)
               (equal (car current) 0))
             (equal
              (highest-bit2-helper current cb bv)
              (cons 0 (highest-bit2-helper
                       (cdr current) (cdr cb) (cdr bv)))))
             ((expand (highest-bit2-helper z x v))))

(prove-lemma highest-bit2-helper-cons-0-rewrite (rewrite)
             (implies
              (and
               (not (all-zero-bitvp cb))
               (at-most-one-bit-on cb)
               (bit-vectorp cb (length cb))
               (bit-vectorp bv (length cb))
               (bit-vectorp current (length cb))
               (equal (car bv) 0)
               (equal (car current) 0))
             (equal
              (highest-bit2-helper current cb bv)
              (cons 0 (highest-bit2-helper
                       (cdr current) (cdr cb) (cdr bv)))))
             ((use (highest-bit2-helper-cons-0 (size (length cb))))))

(prove-lemma append-zeros-0 (rewrite)
             (implies
              (and
               (all-zero-bitvp x)
               (properp x))
              (equal (append x '(0)) (cons 0 x))))

(prove-lemma highest-bit2-helper-cons-helper nil
             (implies
              (and
               (bit-vectorp bv size)
               (bit-vectorp cb size)
               (at-most-one-bit-on cb)
               (at-most-one-bit-on current)
               (not (all-zero-bitvp cb))
               (or
                (all-zero-bitvp current)
                (lessp (trailing-zeros current) (trailing-zeros cb)))
               (bit-vectorp current size)
               (not (zerop size)))
               (equal
                (highest-bit2-helper current cb bv)
                (if (equal (car bv) 0)
                    (cons 0 (highest-bit2-helper (cdr current)
                                                 (cdr cb) (cdr bv)))
                  (cons 1 (zero-bit-vector (sub1 size))))))
             ((induct (triple-cdr-with-sub1-induct
                       bv cb current size))))
```

```
(prove-lemma highest-bit2-helper-cons-helper-rewrite (rewrite)
             (implies
              (and
               (bit-vectorp bv (length bv))
               (bit-vectorp cb (length bv))
               (at-most-one-bit-on cb)
               (at-most-one-bit-on current)
               (not (all-zero-bitvp cb))
               (or
                (all-zero-bitvp current)
                (lessp (trailing-zeros current) (trailing-zeros cb)))
               (bit-vectorp current (length bv))
               (listp bv))
               (equal
                (highest-bit2-helper current cb bv)
                (if (equal (car bv) 0)
                    (cons 0 (highest-bit2-helper (cdr current)
                                                 (cdr cb) (cdr bv)))
                  (cons 1 (zero-bit-vector (sub1 (length bv)))))))
             ((use (highest-bit2-helper-cons-helper (size (length bv))))
              (disable-theory t)
              (enable equal-length-0)
              (enable-theory ground-zero)))

(prove-lemma bit-vectorp-cdr-from-free (rewrite)
             (implies
              (bit-vectorp x n)
              (equal (bit-vectorp (cdr x) s)
                     (equal (add1 s) n))))

(prove-lemma all-zero-bitvp-one-bit-vector (rewrite)
             (not (all-zero-bitvp (one-bit-vector x))))

(prove-lemma trailing-zeros-one-bit-vector (rewrite)
             (equal (trailing-zeros (one-bit-vector n)) 0))

(prove-lemma cdr-zero-one-bit-vector (rewrite)
             (and
              (equal (cdr (one-bit-vector x))
                     (if (lessp x 2) nil (one-bit-vector (sub1 x))))
              (equal (cdr (zero-bit-vector x))
                     (if (zerop x) 0 (zero-bit-vector (sub1 x))))))

(prove-lemma highest-bit2-helper-highest-bit (rewrite)
             (implies
              (and
               (bit-vectorp bv word-size)
               (not (zerop word-size)))
             (equal
              (highest-bit2-helper (zero-bit-vector word-size)
                                   (one-bit-vector word-size) bv)
              (highest-bit bv)))
             ((induct (bit-vectorp bv word-size))
              (disable-theory t)
              (enable-theory ground-zero naturals)
              (enable highest-bit2-helper-cons-helper-rewrite bv-length-weaker
                      length-from-bit-vectorp at-most-one-bit-on-one-bit-vector
                      *1*zero-bit-vector *1*one-bit-vector *1*highest-bit2-helper bitp
                      bit-vectorp cdr-zero-one-bit-vector all-zero-bitvp-zero-bit-vector
                      bit-vectorp-simple-not length highest-bit
                      trailing-zeros-of-all-zero-bitvp bit-vectorp-zero-bit-vector-better
```

```
                      length-zero-bit-vector trailing-zeros-one-bit-vector
                      all-zero-bitvp-one-bit-vector all-zero-bitvp-means-at-most-one-bit-on
                      bit-vectorp-one-bit-vector-rewrite)))


(defn highest-bit-input-conditionp (p0)
  (and
   (equal (car (top (p-temp-stk p0))) 'bitv)
   (equal (cddr (top (p-temp-stk p0))) nil)
   (not (zerop (p-word-size p0)))
   (listp (p-ctrl-stk p0))
   (at-least-morep (p-ctrl-stk-size (p-ctrl-stk p0))
                   6 (p-max-ctrl-stk-size p0))
   (at-least-morep (length (p-temp-stk p0))
                   2 (p-max-temp-stk-size p0))
   (equal (definition 'highest-bit (p-prog-segment p0))
          (highest-bit-program))
   (equal (definition 'push-1-vector (p-prog-segment p0))
          (push-1-vector-program (p-word-size p0)))
   (bit-vectorp (cadr (top (p-temp-stk p0))) (p-word-size p0))))

(defn highest-bit-clock (bv)
  (plus 6 (highest-bit-loop-clock (one-bit-vector (length bv)) bv)))

(prove-lemma cons-0-zero-bit-vector (rewrite)
             (equal
              (cons 0 (zero-bit-vector x))
              (zero-bit-vector (add1 x))))

(disable cons-0-zero-bit-vector)

(prove-lemma correctness-of-highest-bit (rewrite)
             (implies
              (and
               (equal p0 (p-state
                          pc
                          ctrl-stk
                          (cons b temp-stk)
                          prog-segment
                          data-segment
                          max-ctrl-stk-size
                          max-temp-stk-size
                          word-size
                          'run))
               (equal (p-current-instruction p0)
                      '(call highest-bit))
               (equal bc (cadr b))
               (highest-bit-input-conditionp p0))
              (equal
               (p (p-state
                          pc
                          ctrl-stk
                          (cons b temp-stk)
                          prog-segment
                          data-segment
                          max-ctrl-stk-size
                          max-temp-stk-size
                          word-size
                          'run)
                  (highest-bit-clock bc))
               (p-state (add1-addr pc)
                        ctrl-stk
```

```
                      (cons (list 'bitv (highest-bit (cadr b)))
                            temp-stk)
                      prog-segment
                      data-segment
                      max-ctrl-stk-size
                      max-temp-stk-size
                      word-size
                      'run)))
           ((use (highest-bit-correctness-general
                  (ret-pc (add1-addr pc))
                  (current (zero-bit-vector word-size))
                  (cb (one-bit-vector word-size))
                  (bv (cadr b))))
            (expand (highest-bit-clock (cadr b)))
            (enable cons-0-zero-bit-vector)
            (disable highest-bit-loop-clock zero-bit-vector)))
```

```
;; match-and-xor

(defn match-and-xor-program ()
  '(match-and-xor (vecs numvecs match xor-vector) ((i (nat 0)))
                 (push-constant (nat 0))
                 (pop-local i)
           (dl loop ()
                 (push-local vecs))
                 (fetch)
                 (push-local match)
                 (and-bitv)
                 (test-bitv-and-jump not-all-zeros found)
                 (push-local i)
                 (add1-nat)
                 (set-local i)
                 (push-local numvecs)
                 (lt-nat)
                 (test-bool-and-jump f done)
                 (push-local vecs)
                 (push-constant (nat 1))
                 (add-addr)
                 (pop-local vecs)
                 (jump loop)
           (dl found ()
                 (push-local vecs))
                 (fetch)
                 (push-local xor-vector)
                 (xor-bitv)
                 (push-local vecs)
                 (deposit)
            (dl done ()
                 (ret))))


(defn example-match-and-xor-p-state ()
  (p-state '(pc (main . 0))
          '((nil (pc (main . 0))))
          nil
          (list '(main nil nil
                      (push-constant (addr (arr . 0)))
                      (push-constant (nat 6))
                      (push-constant (bitv (0 0 0 1 0 0 0 0)))
```

```
                      (push-constant (bitv (1 1 1 1 1 1 1 1)))
                      (call match-and-xor)
                      (ret))
                 (match-and-xor-program))
           '((arr (bitv (0 1 0 0 1 0 0 1))
                  (bitv (0 0 0 0 0 0 0 1))
                  (bitv (0 0 0 1 0 0 0 1))
                  (bitv (0 0 0 0 0 0 0 1))
                  (bitv (0 0 0 1 0 0 0 1))
                  (bitv (0 1 1 0 1 0 0 1))))
           10
           8
           8
           'run))

(defn match-and-xor (bvs match xor-vector)
  (if (listp bvs)
      (if (all-zero-bitvp (and-bitv (car bvs) match))
          (cons (car bvs) (match-and-xor (cdr bvs) match xor-vector))
        (cons (xor-bitv (car bvs) xor-vector) (cdr bvs)))
    nil))

(defn match-and-xor-loop-clock (bvs match)
  (if (listp bvs)
      (if (all-zero-bitvp (and-bitv (car bvs) match))
          (if (listp (cdr bvs))
            (plus 16 (match-and-xor-loop-clock (cdr bvs) match))
            12)
        12)
    0))

(defn tag-array (tag array)
  (if (listp array)
      (cons (tag tag (car array))
            (tag-array tag (cdr array)))
    nil))


(defn match-and-xor-general-induct (numvecs i)
  (if (lessp i numvecs)
      (match-and-xor-general-induct numvecs (add1 i))
    t)
  ((lessp (difference numvecs i))))

(prove-lemma tag-array-untag-array (rewrite)
  (implies
   (bit-vectors-piton x (length (untag (car x))))
   (equal (tag-array 'bitv (untag-array x)) x)))

(prove-lemma bit-vectors-piton-free-means (rewrite)
             (implies
              (bit-vectors-piton x size)
              (and
               (bit-vectors-piton x (length (cadr (car x))))
               (equal
                (bit-vectors-piton (cdr x) (length (cadr (cadr x))))
                (listp x)))))

(prove-lemma listp-cdr-nthcdr (rewrite)
             (equal
              (listp (cdr (nthcdr i x)))
              (lessp i (length (cdr x)))))
```

```
(prove-lemma nthcdr-untag-array (rewrite)
              (equal
               (nthcdr i (untag-array x))
               (if (lessp i (length x))
                   (untag-array (nthcdr i x))
                 (if (equal (fix i) (length x)) nil 0))))

(prove-lemma put-length-cdr (rewrite)
             (implies
              (properp x)
             (equal
              (put val (length (cdr x)) x)
              (append (all-but-last x) (list val)))))

(prove-lemma nthcdr-length-cdr (rewrite)
             (implies
              (properp x)
             (equal
              (nthcdr (length (cdr x)) x)
              (if (listp x) (list (last x)) nil))))

(prove-lemma get-length-cdr (rewrite)
             (equal
              (get (length (cdr x)) x)
              (if (listp x) (last x) 0)))

(prove-lemma append-all-but-last-last (rewrite)
             (implies
              (properp x)
              (equal
               (append (all-but-last x) (list (last x)))
               (if (nlistp x) (list (last x)) x))))

(prove-lemma listp-cdr-untag-array (rewrite)
             (equal
              (listp (cdr (untag-array x)))
              (listp (cdr x))))

(prove-lemma bit-vectorp-last (rewrite)
             (implies
              (bit-vectors-piton bvs s)
              (equal (bit-vectorp (cadr (last bvs)) s) (listp bvs))))

(prove-lemma bit-vectors-piton-means-properp (rewrite)
             (implies
              (bit-vectors-piton x s)
              (properp x)))

(prove-lemma bit-vectors-piton-means-last (rewrite)
             (implies
              (and
               (bit-vectors-piton state size)
               (listp state))
              (and
               (equal (car (last state)) 'bitv)
               (listp (last state))
               (bit-vectorp (cadr (last state)) size)
               (equal (cddr (last state)) nil)
               (equal (length (cadr (last state))) (fix size))))
             ((disable bit-vectorp-last)))
```

```
(prove-lemma list-bitv-cadr-bitvp (rewrite)
             (implies
              (and
               (equal (car x) 'bitv)
               (equal (cddr x) nil))
              (equal (list 'bitv (cadr x)) x)))

(prove-lemma equal-x-put-assoc-x (rewrite)
             (equal
              (equal x (put-assoc val name  x))
              (or
               (not (definedp name x))
               (equal (assoc name x) (cons name val)))))

(prove-lemma cons-n-assoc-n (rewrite)
             (implies
              (listp (assoc n d))
              (equal (cons n (cdr (assoc n d)))
                     (if (definedp n d)
                         (assoc n d)
                       (cons n 0)))))

(prove-lemma cons-n-cadr-list-assoc-n (rewrite)
             (implies
               (equal (cddr (assoc n d)) nil)
              (equal (list n (cadr (assoc n d)))
                     (if (definedp n d)
                         (assoc n d)
                       (cons n 0)))))

(prove-lemma cons-n-assoc-n-hack (rewrite)
         (implies
         (listp (cdr (assoc n d)))
         (equal (cons n (cdr (assoc n d)))
              (if (definedp n d)
                  (assoc n d)
                   (cons n 0)))))

(prove-lemma car-untag-array (rewrite)
             (equal
              (car (untag-array x))
              (untag (car x))))

(prove-lemma car-nthcdr (rewrite)
             (equal
              (car (nthcdr i x))
              (get i x)))


(prove-lemma tag-array-cdr-untag-array-hack (rewrite)
             (implies
              (bit-vectors-piton x (length (untag (car x))))
             (equal
              (tag-array 'bitv (cdr (untag-array x)))
              (if (listp x) (cdr x) nil))))

(prove-lemma bit-vectors-piton-means-car (rewrite)
             (implies
              (and
               (bit-vectors-piton state size)
               (listp state))
              (and
```

```
              (equal (caar state) 'bitv)
              (listp (car state))
              (bit-vectorp (cadr (car state)) size)
              (equal (cddr (car state)) nil)
              (equal (length (cadr (car state))) (fix size)))))

(prove-lemma equal-put-assoc (rewrite)
            (equal
             (equal (put-assoc v1 n s) (put-assoc v2 n s))
             (or
              (not (definedp n s))
              (equal v1 v2))))

(prove-lemma get-nlistp-better (rewrite)
            (implies
             (not (lessp i (length x)))
             (equal (get i x) 0)))

(prove-lemma equal-append-zero-bit-vector-zero-bit-vector (rewrite)
            (equal
             (equal (append (zero-bit-vector n1) x)
                    (append (zero-bit-vector n2) y))
             (if (lessp n1 n2)
                 (equal
                  x (append (zero-bit-vector (difference n2 n1)) y))
               (equal
                y (append (zero-bit-vector (difference n1 n2)) x)))))

(prove-lemma append-make-list-from-cons-get-hack (rewrite)
            (equal
             (append (make-list-from i x) (cons (get i x) y))
             (append (make-list-from (add1 i) x) y)))

(prove-lemma cdr-untag-array-nthcdr (rewrite)
            (implies
             (lessp i (length x))
             (equal
              (cdr (untag-array (nthcdr i x)))
              (untag-array (nthcdr i (cdr x))))))

(prove-lemma equal-nil-nthcdr-length (rewrite)
            (equal
             (equal nil (nthcdr (length x) x))
             (properp x)))

(prove-lemma lessp-0-length-better (rewrite)
            (implies
             (zerop x)
             (equal (lessp x (length y)) (listp y))))

(prove-lemma bit-vectors-piton-means-get-cdr (rewrite)
            (implies
             (bit-vectors-piton state size)
             (and
              (equal (car (get i (cdr state)))
                     (if (lessp i (length (cdr state))) 'bitv 0))
              (equal (listp (get i (cdr state)))
                     (lessp i (length (cdr state))))
              (equal (bit-vectorp (cadr (get i (cdr state))) size)
                     (lessp i (length (cdr state))))
              (equal (cddr (get i (cdr state)))
                     (if (lessp i (length (cdr state))) nil 0))
```

```
               (equal (length (cadr (get i (cdr state))))
                      (if (lessp i (length (cdr state))) (fix size) 0))))
             ((induct (get i state))))

(prove-lemma bit-vectors-piton-nthcdr (rewrite)
             (implies
              (bit-vectors-piton x s1)
              (and
               (equal (bit-vectors-piton (nthcdr i x) s)
                      (or
                       (and
                        (lessp i (length x))
                        (equal (fix s1) (fix s)))
                       (equal (fix i) (length x))))
               (equal (bit-vectors-piton (nthcdr i (cdr x)) s)
                      (and
                       (lessp i (length x))
                       (or (equal (fix s) (fix s1))
                           (equal (fix i) (length (cdr x)))))))))

(prove-lemma tag-array-untag-array-nthcdr-cddr-hack (rewrite)
             (implies
              (and
               (lessp i (length x))
               (bit-vectors-piton x s))
              (equal
               (tag-array 'bitv (untag-array (nthcdr i (cdr x))))
               (nthcdr i (cdr x))))
             ((use (tag-array-untag-array (x (nthcdr i (cdr x)))))))

(prove-lemma append-make-list-from-put-hack (rewrite)
             (implies
              (lessp i (length x))
             (equal
              (append
               (make-list-from i x)
               (cons v (nthcdr i (cdr x))))
              (put v i x))))

(enable lessp-sub1-x-x)
(enable lessp-x-x)

(prove-lemma correctness-of-match-and-xor-general nil
             (implies
              (and
                  (listp ctrl-stk)
                  (at-least-morep (p-ctrl-stk-size ctrl-stk)
                                  7 max-ctrl-stk-size)
                  (at-least-morep (length temp-stk)
                                  4 max-temp-stk-size)
                  (equal (definition 'match-and-xor prog-segment) (match-and-xor-program))
                  (bit-vectorp match word-size)
                  (bit-vectorp xor-vector word-size)
                  (equal numvecs (length (array vecs data-segment)))
                  (bit-vectors-piton (array vecs data-segment) word-size)
                  (definedp vecs data-segment)
                  (numberp i)
                  (lessp i numvecs)
                  (lessp (length (array vecs data-segment)) (exp 2 word-size)))
              (equal
               (p
                (p-state '(pc (match-and-xor . 2))
```

```
                         (cons (list
                                (list
                                 (cons 'vecs (list 'addr (cons vecs i)))
                                 (cons 'numvecs (list 'nat numvecs))
                                 (cons 'match (list 'bitv match))
                                 (cons 'xor-vector (list 'bitv xor-vector))
                                 (cons 'i (list 'nat i)))
                                ret-pc)
                               ctrl-stk)
                         temp-stk
                         prog-segment
                         data-segment
                         max-ctrl-stk-size
                         max-temp-stk-size
                         word-size
                         'run)
                (match-and-xor-loop-clock
                 (nthcdr i (untag-array (array vecs data-segment))) match))
               (p-state
                ret-pc
                ctrl-stk
                temp-stk
                prog-segment
                (put-assoc
                 (append (make-list-from i (array vecs data-segment))
                         (tag-array 'bitv
                                    (match-and-xor
                                     (untag-array (nthcdr i (array vecs data-segment)))
                                     match xor-vector)))
                 vecs data-segment)
                max-ctrl-stk-size
                max-temp-stk-size
                word-size
                'run)))
             ((induct (match-and-xor-general-induct numvecs i))
              (disable bit-vectors-piton)
              (expand
               (match-and-xor-loop-clock
                (untag-array (nthcdr i (cdr (assoc vecs data-segment))))
                match)
               (match-and-xor-loop-clock
                (untag-array (nthcdr (length (cddr (assoc vecs data-segment)))
                                     (cdr (assoc vecs data-segment))))
                match))))

(defn match-and-xor-clock (bvs match)
  (plus 3 (match-and-xor-loop-clock bvs match)))


(defn match-and-xor-input-conditionp (p0)
  (and
   (equal (car (top (p-temp-stk p0))) 'bitv)
   (equal (cddr (top (p-temp-stk p0))) nil)
   (equal (car (top (cdr (p-temp-stk p0)))) 'bitv)
   (equal (cddr (top (cdr (p-temp-stk p0)))) nil)
   (equal (car (top (cddr (p-temp-stk p0)))) 'nat)
   (equal (cddr (top (cddr (p-temp-stk p0)))) nil)
   (equal (car (top (cdddr (p-temp-stk p0)))) 'addr)
   (equal (cdr (cadr (top (cdddr (p-temp-stk p0))))) 0)
   (listp (cadr (top (cdddr (p-temp-stk p0)))))
   (equal (cddr (top (cdddr (p-temp-stk p0)))) nil)
   (not (zerop (cadr (top (cddr (p-temp-stk p0))))))
```

```
   (listp (p-ctrl-stk p0))
   (at-least-morep (p-ctrl-stk-size (p-ctrl-stk p0))
                   7 (p-max-ctrl-stk-size p0))
   (at-least-morep (length (p-temp-stk p0))
                   0 (p-max-temp-stk-size p0))
   (equal (definition 'match-and-xor (p-prog-segment p0))
          (match-and-xor-program))
   (bit-vectorp (cadr (top (cdr (p-temp-stk p0)))) (p-word-size p0))
   (bit-vectorp (cadr (top (p-temp-stk p0))) (p-word-size p0))
   (equal (cadr (top (cddr (p-temp-stk p0))))
          (length (array (car (cadr (top (cdddr (p-temp-stk p0)))))
                         (p-data-segment p0))))
   (definedp (car (cadr (top (cdddr (p-temp-stk p0)))))
     (p-data-segment p0))
   (lessp (length (array (car (cadr (top (cdddr (p-temp-stk p0)))))
                         (p-data-segment p0)))
          (exp 2 (p-word-size p0)))
   (bit-vectors-piton
    (array (car (cadr (top (cdddr (p-temp-stk p0)))))
           (p-data-segment p0))
    (p-word-size p0))))

(prove-lemma correctness-of-match-and-xor (rewrite)
             (implies
              (and
               (equal p0 (p-state
                          pc
                          ctrl-stk
                          (cons x (cons m (cons n (cons v temp-stk))))
                          prog-segment
                          data-segment
                          max-ctrl-stk-size
                          max-temp-stk-size
                          word-size
                          'run))
               (equal (p-current-instruction p0) '(call match-and-xor))
               (match-and-xor-input-conditionp p0)
               (equal match (cadr m))
               (equal vecs-to-match
                      (untag-array (array (caadr v) data-segment))))
              (equal
               (p (p-state
                          pc
                          ctrl-stk
                          (cons x (cons m (cons n (cons v temp-stk))))
                          prog-segment
                          data-segment
                          max-ctrl-stk-size
                          max-temp-stk-size
                          word-size
                          'run)
                  (match-and-xor-clock vecs-to-match match))
               (p-state (add1-addr pc)
                        ctrl-stk
                        temp-stk
                        prog-segment
                        (put-assoc
                         (tag-array 'bitv
                          (match-and-xor
                           (untag-array
                            (array (caadr v) data-segment))
                           (cadr m) (cadr x)))
```

```
                        (caadr v) data-segment)
                      max-ctrl-stk-size
                      max-temp-stk-size
                      word-size
                      'run)))
              ((disable match-and-xor-loop-clock)
               (use (correctness-of-match-and-xor-general
                     (ret-pc (add1-addr pc)) (i 0)
                     (xor-vector (cadr x)) (match (cadr m))
                     (numvecs (cadr n)) (vecs (caadr v))))))

;;; nat-to-bv-list

(defn nat-to-bv-list-program ()
  '(nat-to-bv-list (nat-list bv-list length) ((i (nat 0)))
               (dl loop ()
                     (push-local nat-list))
                     (fetch)
                     (call nat-to-bv)
                     (push-local bv-list)
                     (deposit)
                     (push-local i)
                     (add1-nat)
                     (set-local i)
                     (push-local length)
                     (eq)
                     (test-bool-and-jump t done)
                     (push-local nat-list)
                     (push-constant (nat 1))
                     (add-addr)
                     (pop-local nat-list)
                     (push-local bv-list)
                     (push-constant (nat 1))
                     (add-addr)
                     (pop-local bv-list)
                     (jump loop)
                 (dl done () (ret))))

(defn example-nat-to-bv-list-p-state ()
  (p-state '(pc (main . 0))
           '((nil (pc (main . 0))))
           nil
           (list '(main nil nil
                        (push-constant (addr (arr . 0)))
                        (push-constant (addr (arr2 . 0)))
                        (push-constant (nat 6))
                        (call nat-to-bv-list)
                        (ret))
                 (nat-to-bv-list-program)
                 (nat-to-bv-program)
                 (push-1-vector-program 8))
           '((arr (nat 3)
                  (nat 5)
                  (nat 8)
                  (nat 0)
                  (nat 23)
                  (nat 9))
             (arr2 nil nil nil nil nil nil))
           100
           80
           8
           'run))
```

```
(defn nat-to-bv-list (nat-list size)
  (if (listp nat-list)
      (cons
       (nat-to-bv (car nat-list) size)
       (nat-to-bv-list (cdr nat-list) size))
    nil))

(defn nat-to-bv-list-loop-clock (i nats)
       (if (lessp i (length nats))
           (clock-plus '2
               (clock-plus (nat-to-bv-clock (untag (get i nats)))
                   (if (lessp i (length (cdr nats)))
                       (clock-plus '17
                           (nat-to-bv-list-loop-clock (add1 i) nats))
                       '9)))
           '0)
       ((lessp (difference (length nats) i))))

(defn nat-to-bv-list-loop-induct (i nats bvs-name word-size data-segment)
  (if (and (lessp i (length nats))
           (lessp i (length (cdr nats))))
      (nat-to-bv-list-loop-induct
       (add1 i) nats bvs-name word-size
       (put-assoc (append
                   (make-list-from i (cdr (assoc bvs-name data-segment)))
                   (cons
                    (list 'bitv (nat-to-bv (untag (get i nats)) word-size))
                    (nthcdr (add1 i) (cdr (assoc bvs-name data-segment)))))
                  bvs-name data-segment))
    t)
  ((lessp (difference (length nats) i))))



(prove-lemma nat-list-piton-means-car (rewrite)
             (implies
              (and
               (nat-list-piton state size)
               (listp state))
              (and
               (equal (caar state) 'nat)
               (listp (car state))
               (numberp (cadr (car state)))
               (lessp (cadr (car state)) (exp 2 size))
               (equal (lessp (cadr (car state)) (exp 2 size)) t)
               (equal (cddr (car state)) nil))))

(defn array-pitonp (array length)
  (if (listp array)
      (and
       (not (zerop length))
       (array-pitonp (cdr array) (sub1 length)))
    (and
     (equal array nil)
     (zerop length))))


(prove-lemma nat-list-piton-means-last (rewrite)
             (implies
              (nat-list-piton state size)
              (and
```

```
               (equal (car (last state)) (if (listp state) 'nat 0))
               (equal (listp (last state)) (listp state))
               (numberp (cadr (last state)))
               (equal (lessp (cadr (last state)) (exp 2 size)) t)
               (equal (cddr (last state)) (if (listp state) nil 0)))))

(prove-lemma nat-list-piton-means-last-cdr (rewrite)
             (implies
              (and
               (nat-list-piton state size)
               (listp state))
              (and
               (equal (car (last (cdr state)))
                      (if (listp (cdr state)) 'nat 0))
               (equal (listp (last (cdr state))) (listp (cdr state)))
               (numberp (cadr (last (cdr state))))
               (equal (lessp (cadr (last (cdr state))) (exp 2 size)) t)
               (equal (cddr (last (cdr state)))
                      (if (listp (cdr state)) nil 0)))))

(enable definedp-put-assoc)

(prove-lemma assoc-put-assoc-better (rewrite)
             (equal (assoc n1 (put-assoc v n2 a))
                    (if (definedp n1 a)
                        (if (equal n1 n2)
                            (cons n1 v)
                          (assoc n1 a))
                      f)))

(disable nat-to-bv-list-loop-clock)

(prove-lemma get-add1 (rewrite)
             (implies
              (lessp n 4)
              (equal (get (add1 n) x) (get n (cdr x)))))

(prove-lemma get-0-better (rewrite)
             (implies
              (zerop n)
              (equal (get n x) (car x))))

(prove-lemma length-cdr-tag-array (rewrite)
             (equal
              (length (cdr (tag-array l x)))
              (length (cdr x))))

(prove-lemma length-nat-to-bv-list (rewrite)
             (equal
              (length (nat-to-bv-list x s))
              (length x)))

(prove-lemma length-cdr-nat-to-bv-list (rewrite)
             (equal
              (length (cdr (nat-to-bv-list x s)))
              (length (cdr x))))

(prove-lemma length-tag-array (rewrite)
             (equal
              (length (tag-array l x))
              (length x)))
```

```
(prove-lemma listp-tag-array (rewrite)
             (equal
              (listp (tag-array l x))
              (listp x)))

(prove-lemma listp-nat-to-bv-list (rewrite)
             (equal
              (listp (nat-to-bv-list x s))
              (listp x)))

(prove-lemma array-pitonp-tag-array (rewrite)
             (equal
              (array-pitonp (tag-array l x) length)
              (equal (length x) (fix length)))
             ((induct (array-pitonp x length))))

(prove-lemma array-pitonp-append (rewrite)
             (equal
              (array-pitonp (append x y) size)
              (and
               (not (lessp size (length x)))
               (array-pitonp y (difference size (length x)))))
             ((induct (get size x))))

(prove-lemma nat-list-piton-means-cadr (rewrite)
             (implies
              (and
               (nat-list-piton state size)
               (listp (cdr state)))
              (and
               (equal (caadr state) 'nat)
               (listp (cadr state))
               (numberp (cadr (cadr state)))
               (lessp (cadr (cadr state)) (exp 2 size))
               (equal (lessp (cadr (cadr state)) (exp 2 size)) t)
               (equal (cddr (cadr state)) nil))))

(prove-lemma array-pitonp-add1 (rewrite)
             (equal
              (array-pitonp x (add1 n))
              (and
               (listp x)
               (array-pitonp (cdr x) n))))

(enable put-assoc-put-assoc)

(prove-lemma length-cdr-nlistp (rewrite)
             (implies
              (not (listp (cdr x)))
              (equal (length x) (if (listp x) 1 0))))

(prove-lemma equal-nil-cdr-tag-array-hack (rewrite)
             (equal
              (equal nil (cdr (tag-array l x)))
              (equal (length x) 1)))

(prove-lemma length-from-array-pitonp (rewrite)
             (implies
              (array-pitonp x s)
              (equal (length x) (fix s))))

(disable nat-to-bv-clock)
```

```
(disable nat-to-bv-list-loop-clock)

(prove-lemma nat-list-piton-means-get (rewrite)
             (implies
              (nat-list-piton state size)
              (and
               (equal (car (get p state))
                      (if (lessp p (length state)) 'nat 0))
               (equal (listp (get p state)) (lessp p (length state)))
               (numberp (cadr (get p state)))
               (equal (lessp (cadr (get p state)) (exp 2 size)) t)
               (equal (cddr (get p state))
                      (if (lessp p (length state)) nil 0)))))

(prove-lemma nat-list-piton-means-get-cdr (rewrite)
             (implies
              (and
               (nat-list-piton state size)
               (listp state))
              (and
               (equal (car (get p (cdr state)) )
                      (if (lessp p (length (cdr state))) 'nat 0))
               (equal (listp (get p (cdr state))) (lessp p (length (cdr state))))
               (numberp (cadr (get p (cdr state))))
               (equal (lessp (cadr (get p (cdr state))) (exp 2 size)) t)
               (equal (cddr (get p (cdr state)))
                      (if (lessp p (length (cdr state))) nil 0)))))

(disable nat-list-piton-means)

(prove-lemma length-put-better (rewrite)
             (equal (length (put v n a))
                    (if (lessp n (length a)) (length a) (add1 n))))

(prove-lemma cons-car-x-put-append-make-list-from-hack (rewrite)
             (equal
              (cons (car (put v x b))
                    (append (make-list-from x (cdr (put v x b))) y))
              (append (make-list-from x b) (cons v y))))

(prove-lemma array-pitonp-put (rewrite)
             (implies
              (and
               (array-pitonp a l)
               (equal (fix l) (fix length)))
              (equal
               (array-pitonp (put v n a) length)
               (lessp n (length a)))))


(prove-lemma array-pitonp-means-properp (rewrite)
             (and
              (implies (array-pitonp x s) (properp x))
              (implies (array-pitonp (cdr x) s) (properp x))))

(prove-lemma put-length-cdr-general (rewrite)
             (implies
              (and
               (properp x)
               (equal (length x) (length y)))
             (equal
```

```
              (put val (length (cdr y)) x)
              (append (all-but-last x) (list val)))))

(prove-lemma nthcdr-x-cdr-put-x (rewrite)
             (implies
              (properp a)
             (equal
              (nthcdr x (cdr (put val x a)))
              (if (lessp x (length a))
                  (nthcdr x (cdr a))
                nil))))

(prove-lemma equal-append-a-append-a (rewrite)
             (equal
              (equal (append a b) (append a c))
              (equal b c)))

;The correctness lemma of nat-to-bv-list could be more general -
;the proof assumes the data areas are distinct, though the program
;works when they're not.  I did it this way because I had assumed
;I'd need distinct arrays in NIM, and designed the proof accordingly.
;This is a weakness I should correct in this proof as it will lead
;to sloppy use of memory in the program, but it takes so long
;to do these proofs I'll wait.


(prove-lemma correctness-of-nat-to-bv-list-general nil
             (implies
              (and
                  (listp ctrl-stk)
                  (at-least-morep (p-ctrl-stk-size ctrl-stk)
                                  13 max-ctrl-stk-size)
                  (at-least-morep (length temp-stk)
                                  3 max-temp-stk-size)
                  (equal (definition 'nat-to-bv-list prog-segment)
                         (nat-to-bv-list-program))
                  (equal (definition 'nat-to-bv prog-segment)
                         (nat-to-bv-program))
                  (equal (definition 'push-1-vector prog-segment)
                         (push-1-vector-program word-size))
                  (equal length (length (array nats data-segment)))
                  (array-pitonp (array bvs data-segment) length)
                  (nat-list-piton (array nats data-segment) word-size)
                  (definedp nats data-segment)
                  (definedp bvs data-segment)
                  (not (zerop word-size))
                  (numberp i)
                  (lessp i length)
                  (lessp length (exp 2 word-size))
                  (not (equal nats bvs))
                  (equal natlist (array nats data-segment)))
              (equal
               (p
                (p-state '(pc (nat-to-bv-list . 0))
                         (cons (list
                                (list
                                 (cons 'nat-list
                                       (list 'addr (cons nats i)))
                                 (cons 'bv-list
                                       (list 'addr (cons bvs i)))
                                 (cons 'length (list 'nat length))
                                 (cons 'i (list 'nat i)))
```

```
                         ret-pc)
                        ctrl-stk)
                  temp-stk
                  prog-segment
                  data-segment
                  max-ctrl-stk-size
                  max-temp-stk-size
                  word-size
                  'run)
         (nat-to-bv-list-loop-clock i natlist))
        (p-state
         ret-pc
         ctrl-stk
         temp-stk
         prog-segment
         (put-assoc
          (append (make-list-from i (array bvs data-segment))
                  (tag-array 'bitv
                             (nat-to-bv-list
                              (untag-array
                               (nthcdr i (array nats data-segment)))
                              word-size)))
          bvs data-segment)
         max-ctrl-stk-size
         max-temp-stk-size
         word-size
         'run)))
      ((induct (nat-to-bv-list-loop-induct
                i natlist bvs word-size data-segment))
       (disable nat-list-piton)
       (expand (array-pitonp (cdr (assoc bvs data-segment)) 1)
               (untag-array (cdr (assoc nats data-segment)))
               (untag-array (cdr (assoc bvs data-segment)))
               (nat-to-bv-list-loop-clock 0
                (cdr (assoc nats data-segment)))
               (nat-to-bv-list-loop-clock i natlist)
               (nat-to-bv-list-loop-clock i
                (cdr (assoc nats data-segment))))))

(defn nat-to-bv-list-clock (natlist)
  (clock-plus 1 (nat-to-bv-list-loop-clock 0 natlist)))

(defn nat-to-bv-list-input-conditionp (p0)
              (and
                  (listp (p-ctrl-stk p0))
                  (at-least-morep (p-ctrl-stk-size (p-ctrl-stk p0))
                                  13 (p-max-ctrl-stk-size p0))
                  (at-least-morep (length (p-temp-stk p0))
                                  0 (p-max-temp-stk-size p0))
                  (equal (definition 'nat-to-bv-list (p-prog-segment p0))
                         (nat-to-bv-list-program))
                  (equal (definition 'nat-to-bv (p-prog-segment p0))
                         (nat-to-bv-program))
                  (equal (definition 'push-1-vector (p-prog-segment p0))
                         (push-1-vector-program (p-word-size p0)))
                  (equal
                   (cadr (top (p-temp-stk p0)))
                   (length (array (caadr (top (cddr (p-temp-stk p0)))) (p-data-segment p0))))
                  (array-pitonp
                   (array (caadr (top (cdr (p-temp-stk p0)))) (p-data-segment p0))
                   (cadr (top (p-temp-stk p0))))
                  (nat-list-piton
```

```
                    (array (caadr (top (cddr (p-temp-stk p0)))) (p-data-segment p0))
                    (p-word-size p0))
                   (definedp (caadr (top (cddr (p-temp-stk p0)))) (p-data-segment p0))
                   (definedp (caadr (top (cdr (p-temp-stk p0)))) (p-data-segment p0))
                   (equal (p-current-instruction p0)
                          '(call nat-to-bv-list))
                   (not (zerop (p-word-size p0)))
                   (not (equal (caadr (top (cddr (p-temp-stk p0))))
                               (caadr (top (cdr (p-temp-stk p0))))))
                   (equal (car (top (p-temp-stk p0))) 'nat)
                   (not (zerop (cadr (top (p-temp-stk p0)))))
                   (lessp (cadr (top (p-temp-stk p0))) (exp 2 (p-word-size p0)))
                   (equal (cddr (top (p-temp-stk p0))) nil)
                   (equal (car (top (cdr (p-temp-stk p0)))) 'addr)
                   (listp (cadr (top (cdr (p-temp-stk p0)))))
                   (equal (cdadr (top (cdr (p-temp-stk p0)))) 0)
                   (equal (cddr (top (cdr (p-temp-stk p0)))) nil)
                   (equal (car (top (cddr (p-temp-stk p0)))) 'addr)
                   (listp (cadr (top (cddr (p-temp-stk p0)))))
                   (equal (cdadr (top (cddr (p-temp-stk p0)))) 0)
                   (equal (cddr (top (cddr (p-temp-stk p0)))) nil)))


(prove-lemma correctness-of-nat-to-bv-list (rewrite)
             (implies
              (and
               (equal p0
                      (p-state pc
                         ctrl-stk
                         (cons l (cons b (cons n temp-stk)))
                         prog-segment
                         data-segment
                         max-ctrl-stk-size
                         max-temp-stk-size
                         word-size
                         'run))
               (nat-to-bv-list-input-conditionp p0)
               (equal natlist (array (caadr n) data-segment)))
              (equal
               (p
                (p-state pc
                         ctrl-stk
                         (cons l (cons b (cons n temp-stk)))
                         prog-segment
                         data-segment
                         max-ctrl-stk-size
                         max-temp-stk-size
                         word-size
                         'run)
                (nat-to-bv-list-clock natlist))
               (p-state
                (add1-addr pc)
                ctrl-stk
                temp-stk
                prog-segment
                (put-assoc
                 (tag-array 'bitv
                           (nat-to-bv-list
                            (untag-array
                             (array (caadr n) data-segment))
                            word-size))
                 (caadr b) data-segment)
```

```
                max-ctrl-stk-size
                max-temp-stk-size
                word-size
                'run)))
              ((use (correctness-of-nat-to-bv-list-general
                     (i 0)
                     (ret-pc (list (car pc) (cons (caadr pc) (add1 (cdadr pc)))))
                     (nats (caadr n))
                     (bvs (caadr b))
                     (length (cadr l))))))


(disable nat-to-bv-list-program)
(disable match-and-xor-program)
(disable highest-bit-program)
(disable number-with-at-least-program)
(disable bv-to-nat-program)
(disable nat-to-bv-program)
(disable push-1-vector-program)
(disable xor-bvs-program)


;; bv-to-nat-list

(defn bv-to-nat-list-program ()
  '(bv-to-nat-list (bv-list nat-list length) ((i (nat 0)))
              (dl loop ()
                    (push-local bv-list))
                    (fetch)
                    (call bv-to-nat)
                    (push-local nat-list)
                    (deposit)
                    (push-local i)
                    (add1-nat)
                    (set-local i)
                    (push-local length)
                    (eq)
                    (test-bool-and-jump t done)
                    (push-local nat-list)
                    (push-constant (nat 1))
                    (add-addr)
                    (pop-local nat-list)
                    (push-local bv-list)
                    (push-constant (nat 1))
                    (add-addr)
                    (pop-local bv-list)
                    (jump loop)
                (dl done () (ret))))

(defn example-bv-to-nat-list-p-state ()
  (p-state '(pc (main . 0))
           '((nil (pc (main . 0))))
           nil
           (list '(main nil nil
                        (push-constant (addr (arr . 0)))
                        (push-constant (addr (arr2 . 0)))
                        (push-constant (nat 6))
                        (call bv-to-nat-list)
                        (ret))
                 (bv-to-nat-list-program)
                 (bv-to-nat-program)
                 (push-1-vector-program 8))
```

```
'((arr (bitv (0 1 0 1 0 1 0 0))
       (bitv (1 1 1 1 1 1 1 1))
       (bitv (0 1 0 1 0 1 0 0))
       (bitv (0 1 0 1 0 1 0 0))
       (bitv (0 0 0 0 0 0 0 0))
       (bitv (0 1 0 1 0 1 0 0)))
  (arr2 nil nil nil nil nil nil))
100
80
8
'run))
```

```
(defn bv-to-nat-list-input-conditionp (p0)
      (and
          (listp (p-ctrl-stk p0))
          (at-least-morep (p-ctrl-stk-size (p-ctrl-stk p0))
                          13 (p-max-ctrl-stk-size p0))
          (at-least-morep (length (p-temp-stk p0))
                          0 (p-max-temp-stk-size p0))
          (equal (definition 'bv-to-nat-list (p-prog-segment p0))
                 (bv-to-nat-list-program))
          (equal (definition 'bv-to-nat (p-prog-segment p0))
                 (bv-to-nat-program))
          (equal (definition 'push-1-vector (p-prog-segment p0))
                 (push-1-vector-program (p-word-size p0)))
          (equal
           (cadr (top (p-temp-stk p0)))
           (length (array (caadr (top (cddr (p-temp-stk p0)))) (p-data-segment p0))))
          (array-pitonp
           (array (caadr (top (cdr (p-temp-stk p0)))) (p-data-segment p0))
           (cadr (top (p-temp-stk p0))))
          (bit-vectors-piton
           (array (caadr (top (cddr (p-temp-stk p0)))) (p-data-segment p0))
           (p-word-size p0))
          (definedp (caadr (top (cddr (p-temp-stk p0)))) (p-data-segment p0))
          (definedp (caadr (top (cdr (p-temp-stk p0)))) (p-data-segment p0))
          (equal (p-current-instruction p0)
                 '(call bv-to-nat-list))
          (not (zerop (p-word-size p0)))
          (not (equal (caadr (top (cddr (p-temp-stk p0))))
                      (caadr (top (cdr (p-temp-stk p0))))))
          (equal (car (top (p-temp-stk p0))) 'nat)
          (not (zerop (cadr (top (p-temp-stk p0)))))
          (lessp (cadr (top (p-temp-stk p0))) (exp 2 (p-word-size p0)))
          (equal (cddr (top (p-temp-stk p0))) nil)
          (equal (car (top (cdr (p-temp-stk p0)))) 'addr)
          (listp (cadr (top (cdr (p-temp-stk p0)))))
          (equal (cdadr (top (cdr (p-temp-stk p0)))) 0)
          (equal (cddr (top (cdr (p-temp-stk p0)))) nil)
          (equal (car (top (cddr (p-temp-stk p0)))) 'addr)
          (listp (cadr (top (cddr (p-temp-stk p0)))))
          (equal (cdadr (top (cddr (p-temp-stk p0)))) 0)
          (equal (cddr (top (cddr (p-temp-stk p0)))) nil)))

(defn bv-to-nat-list-loop-clock (wordsize i bvs)
      (if (lessp i (length bvs))
          (clock-plus '2
              (clock-plus (bv-to-nat-clock wordsize (untag (get i bvs)))
                  (if (lessp i (length (cdr bvs)))
                      (clock-plus '17
                          (bv-to-nat-list-loop-clock
```

```
                            wordsize (add1 i) bvs))
                       '9)))
           '0)
       ((lessp (difference (length bvs) i))))

(defn bv-to-nat-list (bv-list)
  (if (listp bv-list)
      (cons
       (bv-to-nat (car bv-list))
       (bv-to-nat-list (cdr bv-list)))
    nil))

(defn bv-to-nat-list-loop-induct (i bvs nats-name data-segment)
  (if (and (lessp i (length bvs))
           (lessp i (length (cdr bvs))))
      (bv-to-nat-list-loop-induct
       (add1 i) bvs nats-name
       (put-assoc (append
                   (make-list-from i (cdr (assoc nats-name data-segment)))
                   (cons
                    (list 'nat (bv-to-nat (untag (get i bvs))))
                    (nthcdr (add1 i) (cdr (assoc nats-name data-segment)))))
                  nats-name data-segment))
    t)
  ((lessp (difference (length bvs) i))))

(disable bv-to-nat-clock)

(prove-lemma correctness-of-bv-to-nat-list-general nil
             (implies
              (and
                  (listp ctrl-stk)
                  (at-least-morep (p-ctrl-stk-size ctrl-stk)
                                  13 max-ctrl-stk-size)
                  (at-least-morep (length temp-stk)
                                  3 max-temp-stk-size)
                  (equal (definition 'bv-to-nat-list prog-segment)
                         (bv-to-nat-list-program))
                  (equal (definition 'bv-to-nat prog-segment)
                         (bv-to-nat-program))
                  (equal (definition 'push-1-vector prog-segment)
                         (push-1-vector-program word-size))
                  (equal length (length (array bvs data-segment)))
                  (bit-vectors-piton (array bvs data-segment)
                                      word-size)
                  (array-pitonp (array nats data-segment) length)
                  (definedp nats data-segment)
                  (definedp bvs data-segment)
                  (not (zerop word-size))
                  (numberp i)
                  (lessp i length)
                  (lessp length (exp 2 word-size))
                  (not (equal nats bvs))
                  (equal bvlist (array bvs data-segment)))
              (equal
               (p
                (p-state '(pc (bv-to-nat-list . 0))
                         (cons (list
                                (list
                                 (cons 'bv-list
                                       (list 'addr (cons bvs i)))
                                 (cons 'nat-list
```

```
                                   (list 'addr (cons nats i)))
                             (cons 'length (list 'nat length))
                             (cons 'i (list 'nat i)))
                            ret-pc)
                           ctrl-stk)
                     temp-stk
                     prog-segment
                     data-segment
                     max-ctrl-stk-size
                     max-temp-stk-size
                     word-size
                     'run)
            (bv-to-nat-list-loop-clock word-size i bvlist))
           (p-state
            ret-pc
            ctrl-stk
            temp-stk
            prog-segment
            (put-assoc
             (append (make-list-from i (array nats data-segment))
                     (tag-array 'nat
                                (bv-to-nat-list
                                 (untag-array
                                  (nthcdr i (array bvs data-segment))))))
             nats data-segment)
            max-ctrl-stk-size
            max-temp-stk-size
            word-size
            'run)))
         ((induct (bv-to-nat-list-loop-induct
                   i bvlist nats data-segment))
          (disable bit-vectors-piton)
          (expand (array-pitonp (cdr (assoc bvs data-segment)) 1)
                  (untag-array (cdr (assoc nats data-segment)))
                  (untag-array (cdr (assoc bvs data-segment)))
                  (bv-to-nat-list-loop-clock wordsize 0
                   (cdr (assoc nats data-segment)))
                  (bv-to-nat-list-loop-clock wordsize i bvlist)
                  (bv-to-nat-list-loop-clock wordsize i
                   (cdr (assoc nats data-segment))))))


(defn bv-to-nat-list-clock (word-size natlist)
  (clock-plus 1 (bv-to-nat-list-loop-clock word-size 0 natlist)))

(disable bv-to-nat-list-loop-clock)

(disable bv-to-nat-list-program)

(prove-lemma correctness-of-bv-to-nat-list (rewrite)
             (implies
              (and
               (equal p0
                      (p-state pc
                         ctrl-stk
                         (cons l (cons n (cons b temp-stk)))
                         prog-segment
                         data-segment
                         max-ctrl-stk-size
                         max-temp-stk-size
                         word-size
                         'run))
```

```
               (bv-to-nat-list-input-conditionp p0)
               (equal bvlist (array (caadr b) data-segment)))
              (equal
               (p
                (p-state pc
                         ctrl-stk
                         (cons l (cons n (cons b temp-stk)))
                         prog-segment
                         data-segment
                         max-ctrl-stk-size
                         max-temp-stk-size
                         word-size
                         'run)
                (bv-to-nat-list-clock word-size bvlist))
               (p-state
                (add1-addr pc)
                ctrl-stk
                temp-stk
                prog-segment
                (put-assoc
                 (tag-array 'nat
                            (bv-to-nat-list
                             (untag-array
                              (array (caadr b) data-segment))))
                 (caadr n) data-segment)
                max-ctrl-stk-size
                max-temp-stk-size
                word-size
                'run)))
              ((use (correctness-of-bv-to-nat-list-general
                     (i 0)
                     (ret-pc (list (car pc) (cons (caadr pc) (add1 (cdadr pc)))))
                     (nats (caadr n))
                     (bvs (caadr b))
                     (length (cadr l))))))

;; max-nat


(defn max-nat-program ()
  '(max-nat (nat-list length)  ((i (nat 0)) (j (nat 0)))
                 (push-constant (nat 0))
              (dl loop ()
                 (set-local j))
                 (push-local j)
                 (push-local nat-list)
                 (fetch)
                 (set-local j)
                 (lt-nat)
                 (test-bool-and-jump f lab)
                 (pop)
                 (push-local j)
               (dl lab ()
                 (push-local i))
                 (add1-nat)
                 (set-local i)
                 (push-local length)
                 (eq)
                 (test-bool-and-jump t done)
                 (push-local nat-list)
                 (push-constant (nat 1))
                 (add-addr)
```

```
                    (pop-local nat-list)
                    (jump loop)
                (dl done () (ret))))


(defn example-max-nat-p-state ()
  (p-state '(pc (main . 0))
           '((nil (pc (main . 0))))
           nil
           (list '(main nil nil
                        (push-constant (addr (arr . 0)))
                        (push-constant (nat 6))
                        (call max-nat)
                        (ret))
                 (max-nat-program))
           '((arr (nat 3) (nat 10) (nat 3) (nat 6) (nat 9) (nat 0))
             (arr2 nil nil nil nil nil nil))
           100
           80
           8
           'run))

(defn max-list-helper (val x)
  (if (listp x)
      (if (lessp val (car x))
          (max-list-helper (car x) (cdr x))
        (max-list-helper val (cdr x)))
    (fix val)))

(defn max-list (x)
  (if (listp x)
      (if (lessp (max-list (cdr x)) (car x))
          (car x)
        (max-list (cdr x)))
    0))

(prove-lemma max-list-helper-max-list (rewrite)
             (equal (max-list-helper val x)
                    (if (lessp (max-list x) val)
                        val
                      (max-list x))))

(prove-lemma max-list-helper-max-list-0 (rewrite)
             (equal (max-list-helper 0 x)
                    (max-list x)))

(disable max-list-helper-max-list)

(defn max-nat-loop-clock (val i nats)
  (if (lessp i (length nats))
       (if (lessp val (untag (get i nats)))
           (if (lessp i (length (cdr nats)))
               (clock-plus 20
                (max-nat-loop-clock
                 (untag (get i nats)) (add1 i) nats))
             16)
           (if (lessp i (length (cdr nats)))
               (clock-plus 18 (max-nat-loop-clock val (add1 i) nats))
             14))
    0)
  ((lessp (difference (length nats) i))))
```

```
(defn max-nat-loop-induct (i j x natlist length)
  (if (lessp i length)
      (max-nat-loop-induct
       (add1 i) (get i natlist)
       (if (lessp (untag (get i natlist)) (untag x))
           (list 'nat (untag x)) (list 'nat (untag (get i natlist))))
       natlist length)
    t)
  ((lessp (difference length i))))


(prove-lemma list-nat-from-assoc-nat-list-piton-hack (rewrite)
             (implies
              (nat-list-piton nl s)
              (and
               (implies
                (lessp z (length (cdr nl)))
                (equal (list 'nat (cadr (get z (cdr nl))))
                       (get z (cdr nl))))
               (implies
                (listp nl)
                (and
                 (equal
                  (list 'nat (cadr (last nl)))
                  (last nl))
                (equal
                 (list 'nat (cadr (car nl)))
                 (car nl))))))
             ((induct (get z nl))))

(prove-lemma correctness-of-max-nat-general nil
             (implies
              (and
                  (listp ctrl-stk)
                  (at-least-morep (p-ctrl-stk-size ctrl-stk)
                                  4 max-ctrl-stk-size)
                  (at-least-morep (length temp-stk)
                                  4 max-temp-stk-size)
                  (equal (definition 'max-nat prog-segment)
                         (max-nat-program))
                  (equal length (length (array nats data-segment)))
                  (nat-list-piton (array nats data-segment) word-size)
                  (definedp nats data-segment)
                  (not (zerop word-size))
                  (numberp i)
                  (lessp i length)
                  (numberp (untag x))
                  (lessp (untag x) (exp 2 word-size))
                  (equal (car x) 'nat)
                  (equal (cddr x) nil)
                  (lessp length (exp 2 word-size))
                  (equal natlist (array nats data-segment)))
              (equal
               (p
                (p-state '(pc (max-nat . 1))
                         (cons (list
                                (list
                                 (cons 'nat-list
                                       (list 'addr (cons nats i)))
                                 (cons 'length (list 'nat length))
                                 (cons 'i (list 'nat i))
                                 (cons 'j j))
```

```
                                ret-pc)
                               ctrl-stk)
                         (cons x
                          temp-stk)
                         prog-segment
                         data-segment
                         max-ctrl-stk-size
                         max-temp-stk-size
                         word-size
                         'run)
                (max-nat-loop-clock (untag x) i (array nats data-segment)))
               (p-state
                ret-pc
                ctrl-stk
                (cons
                 (list 'nat
                       (max-list-helper (untag x)
                         (untag-array
                          (nthcdr i (array nats data-segment)))))
                 temp-stk)
                prog-segment
                data-segment
                max-ctrl-stk-size
                max-temp-stk-size
                word-size
                'run)))
              ((expand (max-nat-loop-clock
                        (cadr x) i (cdr (assoc nats data-segment))))
               (induct (max-nat-loop-induct i j x natlist length))))

(defn max-nat-clock (natlist)
  (clock-plus 2 (max-nat-loop-clock 0 0 natlist)))

(defn max-nat-input-conditionp (p0)
  (and
   (listp (p-ctrl-stk p0))
   (at-least-morep (p-ctrl-stk-size (p-ctrl-stk p0))
                   6 (p-max-ctrl-stk-size p0))
   (at-least-morep (length (p-temp-stk p0))
                   2 (p-max-temp-stk-size p0))
   (equal (definition 'max-nat (p-prog-segment p0))
          (max-nat-program))
   (equal (untag (top (p-temp-stk p0)))
         (length (array (car (untag (top (cdr (p-temp-stk p0)))))
                        (p-data-segment p0))))
  (nat-list-piton (array (car (untag (top (cdr (p-temp-stk p0)))))
                         (p-data-segment p0))
                  (p-word-size p0))
  (definedp (car (untag (top (cdr (p-temp-stk p0)))))
    (p-data-segment p0))
  (not (zerop (p-word-size p0)))
  (not (zerop (untag (top (p-temp-stk p0)))))
  (equal (car (top (p-temp-stk p0))) 'nat)
  (equal (car (top (cdr (p-temp-stk p0)))) 'addr)
  (listp (untag (top (cdr (p-temp-stk p0)))))
  (lessp (untag (top (p-temp-stk p0))) (exp 2 (p-word-size p0)))
  (equal (cddr (top (cdr (p-temp-stk p0)))) nil)
  (equal (cddr (top (p-temp-stk p0))) nil)
  (equal (cdr (untag (top (cdr (p-temp-stk p0))))) 0)))

(prove-lemma correctness-of-max-nat (rewrite)
             (implies
```

```
              (and
               (equal p0 (p-state
                          pc
                          ctrl-stk
                          (cons n (cons s temp-stk))
                          prog-segment
                          data-segment
                          max-ctrl-stk-size
                          max-temp-stk-size
                          word-size
                          'run))
               (equal (p-current-instruction p0)
                      '(call max-nat))
               (max-nat-input-conditionp p0)
               (equal natlist (array (car (untag s)) data-segment)))
              (equal
               (p (p-state
                          pc
                          ctrl-stk
                          (cons n (cons s temp-stk))
                          prog-segment
                          data-segment
                          max-ctrl-stk-size
                          max-temp-stk-size
                          word-size
                          'run)
                  (max-nat-clock natlist))
               (p-state (add1-addr pc)
                        ctrl-stk
                        (cons
                         (list 'nat (max-list (untag-array natlist)))
                         temp-stk)
                        prog-segment
                        data-segment
                        max-ctrl-stk-size
                        max-temp-stk-size
                        word-size
                        'run)))
              ((use (correctness-of-max-nat-general
                      (i 0) (j '(nat 0)) (x (list 'nat 0))
                      (nats (caadr s))
                     (ret-pc (add1-addr pc))
                     (length (cadr n))))))

(disable max-nat-program)

;; replace-value

;; replace the first occurrence of oldval by newval in list
;; of naturals (assumes oldval occurs in list)

(defn replace-value-program ()
  '(replace-value (list oldval newval) ()
              (dl loop ()
                  (push-local list))
                  (fetch)
                  (push-local oldval)
                  (eq)
                  (test-bool-and-jump t done)
                  (push-local list)
                  (push-constant (nat 1))
                  (add-addr)
```

```
                 (pop-local list)
                 (jump loop)
              (dl done ()
                 (push-local newval))
                 (push-local list)
                 (deposit)
                 (ret)))

(defn example-replace-value-p-state ()
  (p-state '(pc (main . 0))
           '((nil (pc (main . 0))))
           nil
           (list '(main nil nil
                        (push-constant (addr (arr . 0)))
                        (push-constant (nat 3))
                        (push-constant (nat 4))
                        (call replace-value)
                        (ret))
                 (replace-value-program))
           '((arr (nat 9) (nat 10) (nat 3) (nat 6) (nat 9) (nat 0)))
           100
           80
           8
           'run))

(defn replace-value-loop-clock (i list oldvalue)
  (if (lessp i (length list))
      (if (equal (get i list) oldvalue)
          9
        (clock-plus
         10 (replace-value-loop-clock (add1 i) list oldvalue)))
    0)
  ((lessp (difference (length list) i))))

(defn replace-value-loop-induct (i list list-addr)
  (if (lessp i (length list))
      (replace-value-loop-induct (add1 i) list (add1-addr list-addr))
    t)
  ((lessp (difference (length list) i))))

(defn replace-value (list oldvalue newvalue)
  (if (listp list)
      (if (equal (car list) oldvalue)
          (cons newvalue (cdr list))
        (cons (car list) (replace-value (cdr list) oldvalue newvalue)))
    nil))

(prove-lemma member-nthcdr-means (rewrite)
             (implies
              (member x (nthcdr i y))
              (member x y)))

(prove-lemma member-of-natlist-means (rewrite)
             (implies
              (and
               (nat-list-piton y s)
               (member x y))
              (and
               (listp x)
               (equal (car x) 'nat)
               (numberp (cadr x))
               (lessp (cadr x) (exp 2 s))
```

```
              (equal (cddr x) nil))))

(prove-lemma replace-value-nthcdr-open (rewrite)
             (implies
              (and
               (not (equal (get x a) old))
               (lessp x (length a)))
              (equal
               (replace-value (nthcdr x a) old new)
               (cons (get x a)
                     (replace-value (nthcdr x (cdr a)) old new)))))

(prove-lemma list-nat-cadr-get-hack (rewrite)
             (implies
              (and
               (nat-list-piton y s)
               (lessp x (length y)))
              (equal
               (list 'nat (cadr (get x y)))
               (get x y))))

(prove-lemma member-nthcdr-simplify (rewrite)
             (implies
              (and
               (lessp i (length x))
               (not (member a (nthcdr i (cdr x)))))
              (equal
               (member a (nthcdr i x))
               (equal a (get i x)))))

(prove-lemma append-make-list-from-cons-cdr
             (rewrite)
             (implies (lessp i (length y))
                      (equal (append (make-list-from i y)
                                     (cons v (cdr (nthcdr i y))))
                             (put v i y))))

(prove-lemma correctness-of-replace-value-general nil
             (implies
              (and
                  (listp ctrl-stk)
                  (at-least-morep (length temp-stk)
                                  2 max-temp-stk-size)
                  (equal (definition 'replace-value prog-segment)
                         (replace-value-program))
                  (nat-list-piton (array list data-segment) word-size)
                  (definedp list data-segment)
                  (not (zerop word-size))
                  (numberp i)
                  (equal vallist (array list data-segment))
                  (equal (car list-addr) 'addr)
                  (equal (cddr list-addr) nil)
                  (listp (untag list-addr))
                  (equal (car (untag list-addr)) list)
                  (equal (cdr (untag list-addr)) i)
                  (equal (car newvalue) 'nat)
                  (equal (cddr newvalue) nil)
                  (equal (car oldvalue) 'nat)
                  (equal (cddr oldvalue) nil)
                  (numberp (cadr oldvalue))
                  (lessp (cadr oldvalue) (exp 2 word-size))
                  (numberp (untag newvalue))
```

```
              (lessp (untag newvalue) (exp 2 word-size))
              (member oldvalue (nthcdr i vallist)))
          (equal
           (p
            (p-state '(pc (replace-value . 0))
                     (cons (list
                            (list
                             (cons 'list list-addr)
                             (cons 'oldval oldvalue)
                             (cons 'newval newvalue))
                            ret-pc)
                           ctrl-stk)
                     temp-stk
                     prog-segment
                     data-segment
                     max-ctrl-stk-size
                     max-temp-stk-size
                     word-size
                     'run)
            (replace-value-loop-clock i vallist oldvalue))
           (p-state
            ret-pc
            ctrl-stk
            temp-stk
            prog-segment
            (put-assoc
             (append
              (make-list-from i (array list data-segment))
              (replace-value (nthcdr i (array list data-segment))
                             oldvalue newvalue))
             list data-segment)
            max-ctrl-stk-size
            max-temp-stk-size
            word-size
            'run)))
         ((expand
           (replace-value-loop-clock
            (cdadr list-addr)
            (cdr (assoc (caadr list-addr)
                        data-segment))
            oldvalue))
           (induct (replace-value-loop-induct i vallist list-addr))
           (disable member-of-natlist-means)))

(defn replace-value-clock (list ov)
  (clock-plus 1 (replace-value-loop-clock 0 list ov)))

(defn replace-value-input-conditionp (p0)
          (and
              (listp (p-ctrl-stk p0))
              (at-least-morep (length (p-temp-stk p0))
                              0 (p-max-temp-stk-size p0))
              (at-least-morep (p-ctrl-stk-size (p-ctrl-stk p0))
                              5 (p-max-ctrl-stk-size p0))
              (equal (definition 'replace-value (p-prog-segment p0))
                     (replace-value-program))
              (nat-list-piton
               (array (car (untag (top (cddr (p-temp-stk p0)))))
                      (p-data-segment p0))
               (p-word-size p0))
              (definedp (car (untag (top (cddr (p-temp-stk p0)))))
                (p-data-segment p0))
```

```
          (not (zerop (p-word-size p0)))
          (equal (car (top (cddr (p-temp-stk p0)))) 'addr)
          (equal (cddr (top (cddr (p-temp-stk p0)))) nil)
          (listp (untag (top (cddr (p-temp-stk p0)))))
          (equal (cdr (untag (top (cddr (p-temp-stk p0))))) 0)
          (equal (car (top (p-temp-stk p0))) 'nat)
          (equal (cddr (top (p-temp-stk p0))) nil)
          (numberp (untag (top (p-temp-stk p0))))
          (lessp (untag (top (p-temp-stk p0))) (exp 2 (p-word-size p0)))
          (member (top (cdr (p-temp-stk p0)))
                  (array (car (untag (top (cddr (p-temp-stk p0)))))
                         (p-data-segment p0)))))

(prove-lemma correctness-of-replace-value (rewrite)
             (implies
              (and
               (equal p0 (p-state
                          pc
                          ctrl-stk
                          (cons nv (cons ov (cons nats temp-stk)))
                          prog-segment
                          data-segment
                          max-ctrl-stk-size
                          max-temp-stk-size
                          word-size
                          'run))
               (equal (p-current-instruction p0)
                      '(call replace-value))
               (replace-value-input-conditionp p0)
               (equal natlist
                      (array (car (untag nats)) data-segment))
               (equal ov ov2))
              (equal
               (p (p-state
                          pc
                          ctrl-stk
                          (cons nv (cons ov (cons nats temp-stk)))
                          prog-segment
                          data-segment
                          max-ctrl-stk-size
                          max-temp-stk-size
                          word-size
                          'run)
                  (replace-value-clock natlist ov2))
               (p-state
                (add1-addr pc)
                ctrl-stk
                temp-stk
                prog-segment
                (put-assoc
                 (replace-value natlist ov nv)
                 (caadr nats) data-segment)
                max-ctrl-stk-size
                max-temp-stk-size
                word-size
                'run)))
             ((disable replace-value-loop-clock nat-list-piton-means-car)
              (use (correctness-of-replace-value-general
                    (newvalue nv) (oldvalue ov) (list-addr nats)
                    (i 0) (vallist natlist) (list (caadr nats))
                    (ret-pc (add1-addr pc))))))
```

```
(disable replace-value-program)

;; smart-move

(defn smart-move-program ()
  '(smart-move (state numpiles work-area) ((i (nat 0)))
               (push-local state)
               (push-local numpiles)
               (push-constant (nat 2))
               (call number-with-at-least)
               (push-constant (nat 2))
               (lt-nat)
               (test-bool-and-jump t lab)
               (push-local state)
               (push-local work-area)
               (push-local numpiles)
               (call nat-to-bv-list)
               (push-local work-area)
               (push-local numpiles)
               (push-local work-area)
               (push-local numpiles)
               (call xor-bvs)
               (set-local i)
               (call highest-bit)
               (push-local i)
               (call match-and-xor)
               (push-local work-area)
               (push-local state)
               (push-local numpiles)
               (call bv-to-nat-list)
               (ret)
           (dl lab ()
               (push-local state))
               (push-local state)
               (push-local numpiles)
               (call max-nat)
               (push-local state)
               (push-local numpiles)
               (push-constant (nat 1))
               (call number-with-at-least)
               (div2-nat)
               (pop-local i)
               (pop)
               (push-local i)
               (call replace-value)
               (ret)))


(defn example-smart-move-p-state ()
  (p-state '(pc (main . 0))
           '((nil (pc (main . 0))))
           nil
           (list '(main nil nil
                        (push-constant (addr (arr . 0)))
                        (push-constant (nat 4))
                        (push-constant (addr (arr5 . 0)))
                        (call smart-move)
                        (push-constant (addr (arr2 . 0)))
                        (push-constant (nat 4))
                        (push-constant (addr (arr5 . 0)))
                        (call smart-move)
                        (push-constant (addr (arr3 . 0)))
```

```
                        (push-constant (nat 4))
                        (push-constant (addr (arr5 . 0)))
                        (call smart-move)
                        (push-constant (addr (arr4 . 0)))
                        (push-constant (nat 4))
                        (push-constant (addr (arr5 . 0)))
                        (call smart-move)
                        (ret))
                 (replace-value-program)
                 (max-nat-program)
                 (bv-to-nat-list-program)
                 (nat-to-bv-list-program)
                 (match-and-xor-program)
                 (highest-bit-program)
                 (number-with-at-least-program)
                 (bv-to-nat-program)
                 (nat-to-bv-program)
                 (push-1-vector-program 8)
                 (xor-bvs-program)
                 (smart-move-program))
           '((arr (nat 3) (nat 4) (nat 2) (nat 1))
             (arr2 (nat 1) (nat 1) (nat 1) (nat 9))
             (arr3 (nat 1) (nat 1) (nat 0) (nat 9))
             (arr4 (nat 0) (nat 0) (nat 0) (nat 0))
             (arr5 (nat 3) (nat 4) (nat 2) (nat 1)))
           100
           80
           8
           'run))

(defn smart-move (state wordsize)
  (if (lessp (number-with-at-least state 2) 2)
      (replace-value
       state (max-list state)
       (remainder (number-with-at-least state 1) 2))
    (bv-to-nat-list
     (match-and-xor
      (nat-to-bv-list state wordsize)
      (highest-bit (xor-bvs (nat-to-bv-list state wordsize)))
      (xor-bvs (nat-to-bv-list state wordsize))))))

(defn smart-move-clock (state wordsize)
  (clock-plus 4
   (clock-plus (number-with-at-least-clock 2 (tag-array 'nat state))
    (clock-plus 3
     (if (lessp (number-with-at-least state 2) 2)
         (clock-plus 3
          (clock-plus (max-nat-clock (tag-array 'nat state))
           (clock-plus 3
            (clock-plus (number-with-at-least-clock
                         1 (tag-array 'nat state))
             (clock-plus 4
              (clock-plus (replace-value-clock (tag-array 'nat state)
                                               (list 'nat (max-list state)))
               1))))))
       (clock-plus 3
        (clock-plus (nat-to-bv-list-clock (tag-array 'nat state))
         (clock-plus 4
          (clock-plus (xor-bvs-clock (length state))
           (clock-plus 1
            (clock-plus (highest-bit-clock
                         (xor-bvs (nat-to-bv-list state wordsize)))
```

```
             (clock-plus 1
              (clock-plus (match-and-xor-clock
                           (nat-to-bv-list state wordsize)
                           (highest-bit
                            (xor-bvs (nat-to-bv-list state wordsize))))
               (clock-plus 3
                (clock-plus (bv-to-nat-list-clock
                             wordsize
                             (tag-array
                              'bitv
                              (match-and-xor
                               (nat-to-bv-list state wordsize)
                               (highest-bit
                                (xor-bvs (nat-to-bv-list state wordsize)))
                               (xor-bvs (nat-to-bv-list state wordsize)))))
                            1)))))))))))))))

(disable replace-value-clock)
(disable max-nat-clock)
(disable bv-to-nat-list-clock)
(disable nat-to-bv-list-clock)
(disable match-and-xor-clock)
(disable highest-bit-clock)
(disable number-with-at-least-clock)
(disable bv-to-nat-clock)
(disable nat-to-bv-clock)
(disable xor-bvs-clock)
(disable replace-value-program)
(disable max-nat-program)
(disable bv-to-nat-list-program)
(disable nat-to-bv-list-program)
(disable match-and-xor-program)
(disable highest-bit-program)
(disable number-with-at-least-program)
(disable bv-to-nat-program)
(disable nat-to-bv-program)
(disable push-1-vector-program)
(disable *1*xor-bvs-program)
(disable *1*replace-value-program)
(disable *1*max-nat-program)
(disable *1*bv-to-nat-list-program)
(disable *1*nat-to-bv-list-program)
(disable *1*match-and-xor-program)
(disable *1*highest-bit-program)
(disable *1*number-with-at-least-program)
(disable *1*bv-to-nat-program)
(disable *1*nat-to-bv-program)
(disable *1*push-1-vector-program)
(disable *1*xor-bvs-program)

(defn nat-listp (list size)
  (if (listp list)
      (and
       (numberp (car list))
       (lessp (car list) (exp 2 size))
       (nat-listp (cdr list) size))
    (equal list nil)))

(prove-lemma bv-to-nat-list-nat-to-bv-list (rewrite)
             (implies
              (nat-listp x size)
              (equal
```

```
               (bv-to-nat-list (nat-to-bv-list x size))
               x)))

(prove-lemma bit-vectorsp-nat-to-bv-list (rewrite)
             (bit-vectorsp (nat-to-bv-list x size) size))

(prove-lemma nat-to-bv-list-bv-to-nat-list (rewrite)
             (implies
              (bit-vectorsp x size)
              (equal
               (nat-to-bv-list (bv-to-nat-list x) size)
               x)))

(prove-lemma bit-vectorsp-match-and-xor (rewrite)
             (implies
              (bit-vectorsp x size)
              (bit-vectorsp (match-and-xor x y z) size)))

(prove-lemma equal-sub1-add1 (rewrite)
             (and
              (equal
               (equal (sub1 x) (add1 y))
               (equal x (add1 (add1 y))))
              (equal
               (equal (sub1 x) 0)
               (or (zerop x) (equal x 1)))))

;; part of more recent naturals library that's missing from Piton library
(prove-lemma equal-times-x-x
             (rewrite)
             (and (equal (equal (times x y) x)
                         (or (and (numberp x) (equal y 1))
                             (equal x 0)))
                  (equal (equal (times y x) x)
                         (or (and (numberp x) (equal y 1))
                             (equal x 0))))
             ((induct (times y x))))

(prove-lemma equal-exp-x-y-x (rewrite)
             (equal
              (equal (exp x y) x)
              (or
               (equal x 1)
               (and
                (equal x 0)
                (not (zerop y)))
               (and
                (numberp x)
                (equal y 1)))))

(prove-lemma lessp-number-with-at-least (rewrite)
             (not (lessp (length x) (number-with-at-least x min))))

(prove-lemma bit-vectors-piton-tag-array (rewrite)
             (implies
              (bit-vectorsp x size)
              (bit-vectors-piton (tag-array 'bitv x) size)))

(prove-lemma tag-array-untag-array-of-nat-list-piton (rewrite)
             (implies
              (nat-list-piton x size)
              (equal (tag-array 'nat (untag-array x)) x)))
```

```
(prove-lemma tag-array-untag-array-of-bit-vectors-piton (rewrite)
             (implies
              (bit-vectors-piton x size)
              (equal (tag-array 'bitv (untag-array x)) x)))

(prove-lemma bit-vectorp-xor-bvs-nat-to-bv-list (rewrite)
             (equal
              (bit-vectorp (xor-bvs (nat-to-bv-list x s)) s)
              (or
               (listp x)
               (zerop s))))

(prove-lemma listp-cdr-assoc-hack-from-free (rewrite)
             (implies
              (and
               (equal (length (cdr (assoc x y))) free)
               (not (zerop free)))
              (listp (cdr (assoc x y)))))

(prove-lemma bit-vectorp-highest-bit (rewrite)
             (implies
              (bit-vectorp x s)
              (bit-vectorp (highest-bit x) s)))

(prove-lemma length-match-and-xor (rewrite)
             (equal
              (length (match-and-xor list m v))
              (length list)))

(prove-lemma array-pitonp-from-nat-list-piton (rewrite)
             (implies
              (nat-list-piton x s)
              (equal
               (array-pitonp x length)
               (equal (fix length) (length x)))))

(prove-lemma untag-array-tag-array-of-bit-vectorsp (rewrite)
             (implies
              (bit-vectorsp x size)
              (equal (untag-array (tag-array l x)) x)))

(prove-lemma untag-array-tag-array-of-nat-to-bv-list (rewrite)
             (equal
              (untag-array (tag-array l (nat-to-bv-list x size)))
              (nat-to-bv-list x size)))

(prove-lemma untag-array-tag-array-of-match-and-xor-hack (rewrite)
             (equal
              (untag-array
               (tag-array l (match-and-xor (nat-to-bv-list x s) y z)))
              (match-and-xor (nat-to-bv-list x s) y z)))

(prove-lemma member-list-max-list (rewrite)
             (implies
              (nat-list-piton x s)
              (equal
               (member (list 'nat (max-list (untag-array x))) x)
               (listp x))))

(prove-lemma tag-array-replace-value-untag-array (rewrite)
             (implies
```

```
             (nat-list-piton x s)
             (equal
              (tag-array 'nat (replace-value (untag-array x) y z))
              (replace-value x (list 'nat y) (list 'nat z)))))


(defn smart-move-input-conditionp (p0)
  (and
   (listp (p-ctrl-stk p0))
   (at-least-morep (length (p-temp-stk p0))
                   3 (p-max-temp-stk-size p0))
   (at-least-morep (p-ctrl-stk-size (p-ctrl-stk p0))
                   19 (p-max-ctrl-stk-size p0))
   (lessp 1 (p-word-size p0))
   (equal (cddr (top (p-temp-stk p0))) nil)
   (equal (cddr (top (cdr (p-temp-stk p0)))) nil)
   (equal (cddr (top (cddr (p-temp-stk p0)))) nil)
   (equal (car (top (p-temp-stk p0))) 'addr)
   (equal (car (top (cdr (p-temp-stk p0)))) 'nat)
   (lessp (untag (top (cdr (p-temp-stk p0)))) (exp 2 (p-word-size p0)))
   (not (zerop (untag (top (cdr (p-temp-stk p0))))))
   (equal (car (top (cddr (p-temp-stk p0)))) 'addr)
   (listp (untag (top (p-temp-stk p0))))
   (listp (untag (top (cddr (p-temp-stk p0)))))
   (equal (cdr (untag (top (p-temp-stk p0)))) 0)
   (equal (cdr (untag (top (cddr (p-temp-stk p0))))) 0)
   (definedp (car (untag (top (p-temp-stk p0)))) (p-data-segment p0))
   (definedp (car (untag (top (cddr (p-temp-stk p0))))) (p-data-segment p0))
   (not (equal (car (untag (top (p-temp-stk p0))))
               (car (untag (top (cddr (p-temp-stk p0)))))))
   (nat-list-piton (array (car (untag (top (cddr (p-temp-stk p0))))) (p-data-segment p0))
                   (p-word-size p0))
   (array-pitonp (array (car (untag (top (p-temp-stk p0)))) (p-data-segment p0))
                 (untag (top (cdr (p-temp-stk p0)))))
   (equal (length (array (car (untag (top (cddr (p-temp-stk p0))))) (p-data-segment p0)))
          (untag (top (cdr (p-temp-stk p0)))))
   (equal (assoc 'smart-move (p-prog-segment p0))
          (smart-move-program))
   (equal (assoc 'replace-value (p-prog-segment p0))
          (replace-value-program))
   (equal (assoc 'max-nat (p-prog-segment p0))
          (max-nat-program))
   (equal (assoc 'bv-to-nat-list (p-prog-segment p0))
          (bv-to-nat-list-program))
   (equal (assoc 'nat-to-bv-list (p-prog-segment p0))
          (nat-to-bv-list-program))
   (equal (assoc 'match-and-xor (p-prog-segment p0))
          (match-and-xor-program))
   (equal (assoc 'highest-bit (p-prog-segment p0))
          (highest-bit-program))
   (equal (assoc 'number-with-at-least (p-prog-segment p0))
          (number-with-at-least-program))
   (equal (assoc 'bv-to-nat (p-prog-segment p0))
          (bv-to-nat-program))
   (equal (assoc 'nat-to-bv (p-prog-segment p0))
          (nat-to-bv-program))
   (equal (assoc 'push-1-vector (p-prog-segment p0))
          (push-1-vector-program (p-word-size p0)))
   (equal (assoc 'xor-bvs (p-prog-segment p0))
          (xor-bvs-program))))

(prove-lemma correctness-of-smart-move (rewrite)
```

```
            (implies
             (and
              (equal p0 (p-state
                         pc
                         ctrl-stk
                         (cons wa (cons np (cons s temp-stk)))
                         prog-segment
                         data-segment
                         max-ctrl-stk-size
                         max-temp-stk-size
                         word-size
                         'run))
              (equal (p-current-instruction p0)
                     '(call smart-move))
              (equal state
                  (untag-array (array (car (untag s)) data-segment)))
              (equal word-size word-size2)
              (smart-move-input-conditionp p0))
                     (equal
              (p (p-state
                         pc
                         ctrl-stk
                         (cons wa (cons np (cons s temp-stk)))
                         prog-segment
                         data-segment
                         max-ctrl-stk-size
                         max-temp-stk-size
                         word-size
                         'run)
                 (smart-move-clock state word-size2))
              (p-state
               (add1-addr pc)
               ctrl-stk
               temp-stk
               prog-segment
               (put-assoc
                (tag-array 'nat (smart-move state word-size))
                (car (untag s))
                (if (lessp (number-with-at-least state 2) 2)
                    data-segment
                  (put-assoc
                   (tag-array 'bitv
                    (nat-to-bv-list (smart-move state word-size)
                                    word-size))
                   (car (untag wa)) data-segment)))
               max-ctrl-stk-size
               max-temp-stk-size
               word-size
               'run))))

(disable smart-move-clock)
(disable smart-move-program)
(disable *1*smart-move-program)

;; delay

(defn delay-program ()
  '(delay (time) ()
          (dl lab ()
               (push-local time))
               (sub1-nat)
               (set-local time)
```

```
                (no-op)
                (no-op)
                (no-op)
                (no-op)
                (test-nat-and-jump zero done)
                (no-op)
                (jump lab)
           (dl done () (ret))))


(defn example-delay-p-state ()
  (p-state '(pc (main . 0))
           '((nil (pc (main . 0))))
           nil
           (list '(main nil nil
                        (push-constant (nat 4))
                        (call delay)
                        (ret))
                 (delay-program))
           nil
           100
           80
           8
           'run))


(defn delay-loop-clock (time)
  (if (lessp time 2) 9 (plus 10 (delay-loop-clock (sub1 time)))))

(prove-lemma correctness-of-delay-general nil
             (implies
              (and
                  (listp ctrl-stk)
                  (equal (definition 'delay prog-segment)
                         (delay-program))
                  (at-least-morep (length temp-stk) 1 max-temp-stk-size)
                  (lessp 0 time)
                  (lessp time (exp 2 word-size)))
              (equal
               (p
                (p-state '(pc (delay . 0))
                         (cons (list
                                (list
                                 (cons 'time (list 'nat time)))
                                ret-pc)
                               ctrl-stk)
                         temp-stk
                         prog-segment
                         data-segment
                         max-ctrl-stk-size
                         max-temp-stk-size
                         word-size
                         'run)
                (delay-loop-clock time))
               (p-state
                ret-pc
                ctrl-stk
                temp-stk
                prog-segment
                data-segment
                max-ctrl-stk-size
                max-temp-stk-size
```

```
               word-size
               'run)))
            ((induct (times time y))))

(defn delay-input-conditionp (p0)
  (and
   (listp (p-ctrl-stk p0))
   (equal (definition 'delay (p-prog-segment p0)) (delay-program))
   (equal (car (top (p-temp-stk p0))) 'nat)
   (at-least-morep (length (p-temp-stk p0)) 0 (p-max-temp-stk-size p0))
   (at-least-morep (p-ctrl-stk-size (p-ctrl-stk p0)) 3
                   (p-max-ctrl-stk-size p0))
   (lessp 0 (cadr (top (p-temp-stk p0))))
   (lessp (cadr (top (p-temp-stk p0))) (exp 2 (p-word-size p0)))
   (equal (cddr (top (p-temp-stk p0))) nil)))

(defn delay-clock (time)
  (add1 (delay-loop-clock time)))

(prove-lemma correctness-of-delay (rewrite)
             (implies
              (and
               (equal p0 (p-state
                          pc
                          ctrl-stk
                          (cons n temp-stk)
                          prog-segment
                          data-segment
                          max-ctrl-stk-size
                          max-temp-stk-size
                          word-size
                          'run))
               (equal (p-current-instruction p0) '(call delay))
               (delay-input-conditionp p0)
               (equal time (cadr n)))
                     (equal
               (p (p-state
                          pc
                          ctrl-stk
                          (cons n temp-stk)
                          prog-segment
                          data-segment
                          max-ctrl-stk-size
                          max-temp-stk-size
                          word-size
                          'run)
                  (delay-clock time))
               (p-state
                (add1-addr pc)
                ctrl-stk
                temp-stk
                prog-segment
                data-segment
                max-ctrl-stk-size
                max-temp-stk-size
                word-size
                'run)))
            ((use (correctness-of-delay-general
                   (time (cadr n))
                   (ret-pc (add1-addr pc))))))

;; computer-move
```

```
(defn computer-move-program ()
  '(computer-move (state numpiles work-area) ((i (nat 0)))
              (push-constant (nat 250))
              (call delay)
              (push-constant (nat 250))
              (call delay)
              (push-constant (nat 250))
              (call delay)
              (push-constant (nat 250))
              (call delay)
              (push-local state)
              (push-local numpiles)
              (push-constant (nat 2))
              (call number-with-at-least)
              (test-nat-and-jump zero lab)
              (push-local state)
              (push-local work-area)
              (push-local numpiles)
              (call nat-to-bv-list)
              (push-local work-area)
              (push-local numpiles)
              (call xor-bvs)
              (test-bitv-and-jump all-zero lab)
              (push-local state)
              (push-local numpiles)
              (push-local work-area)
              (call smart-move)
              (ret)
           (dl lab ()
              (push-local state))
              (push-local state)
              (push-local numpiles)
              (call max-nat)
              (pop-local i)
              (push-local i)
              (push-local i)
              (sub1-nat)
              (call replace-value)
              (ret)))

(defn example-computer-move-p-state ()
  (p-state '(pc (main . 0))
           '((nil (pc (main . 0))))
           nil
           (list '(main nil nil
                        (push-constant (addr (arr . 0)))
                        (push-constant (nat 4))
                        (push-constant (addr (arr5 . 0)))
                        (call computer-move)
                        (push-constant (addr (arr2 . 0)))
                        (push-constant (nat 4))
                        (push-constant (addr (arr5 . 0)))
                        (call computer-move)
                        (push-constant (addr (arr3 . 0)))
                        (push-constant (nat 4))
                        (push-constant (addr (arr5 . 0)))
                        (call computer-move)
                        (push-constant (addr (arr4 . 0)))
                        (push-constant (nat 4))
                        (push-constant (addr (arr5 . 0)))
                        (call computer-move)
```

```
                    (ret))
                (computer-move-program)
                (delay-program)
                (replace-value-program)
                (max-nat-program)
                (bv-to-nat-list-program)
                (nat-to-bv-list-program)
                (match-and-xor-program)
                (highest-bit-program)
                (number-with-at-least-program)
                (bv-to-nat-program)
                (nat-to-bv-program)
                (push-1-vector-program 8)
                (xor-bvs-program)
                (smart-move-program))
          '((arr (nat 3) (nat 4) (nat 2) (nat 1))
            (arr2 (nat 1) (nat 1) (nat 1) (nat 0))
            (arr3 (nat 1) (nat 1) (nat 0) (nat 9))
            (arr4 (nat 7) (nat 4) (nat 2) (nat 2))
            (arr5 (nat 3) (nat 4) (nat 2) (nat 1)))
          100
          80
          8
          'run))

(defn computer-move (state wordsize)
  (if (or (equal (number-with-at-least state 2) 0)
          (all-zero-bitvp (xor-bvs (nat-to-bv-list state wordsize))))
      (replace-value state (max-list state) (sub1 (max-list state)))
    (smart-move state wordsize)))

(disable delay-clock)

(defn computer-move-clock (state wordsize)
  (clock-plus
   2
   (clock-plus (delay-clock 250)
               (clock-plus 1 (clock-plus (delay-clock 250)
               (clock-plus 1 (clock-plus (delay-clock 250)
               (clock-plus 1 (clock-plus (delay-clock 250)
  (clock-plus
   3
   (clock-plus
    (number-with-at-least-clock 2 (tag-array 'nat state))
    (clock-plus
     1
    (if (equal (number-with-at-least state 2) 0)
        (clock-plus
            3
            (clock-plus
             (max-nat-clock (tag-array 'nat state))
             (clock-plus
              4
              (clock-plus
               (replace-value-clock
                (tag-array 'nat state)
                (list 'nat (max-list state)))
               1))))
  (clock-plus
   3
   (clock-plus
    (nat-to-bv-list-clock (tag-array 'nat state))
```

```
 (clock-plus
  2
  (clock-plus
   (xor-bvs-clock (length state))
   (clock-plus
    1
    (if (all-zero-bitvp (xor-bvs (nat-to-bv-list state wordsize)))
        (clock-plus
         3
         (clock-plus
          (max-nat-clock (tag-array 'nat state))
          (clock-plus
           4
           (clock-plus
            (replace-value-clock
             (tag-array 'nat state)
             (list 'nat (max-list state)))
            1))))
      (clock-plus
       3
       (clock-plus
        (smart-move-clock state wordsize)
        1))))))))))))))))))))

(defn computer-move-input-conditionp (p0)
  (and
   (not (all-zero-bitvp
         (untag-array
          (array (car (untag (top (cddr (p-temp-stk p0)))))
                 (p-data-segment p0)))))
   (listp (p-ctrl-stk p0))
   (at-least-morep (length (p-temp-stk p0))
                   3 (p-max-temp-stk-size p0))
   (at-least-morep (p-ctrl-stk-size (p-ctrl-stk p0))
                   25 (p-max-ctrl-stk-size p0))
   (lessp 7 (p-word-size p0))
   (not (zerop (p-word-size p0)))
   (not (equal (p-word-size p0) 1))
   (equal (cddr (top (p-temp-stk p0))) nil)
   (equal (cddr (top (cdr (p-temp-stk p0)))) nil)
   (equal (cddr (top (cddr (p-temp-stk p0)))) nil)
   (equal (car (top (p-temp-stk p0))) 'addr)
   (equal (car (top (cdr (p-temp-stk p0)))) 'nat)
   (lessp (untag (top (cdr (p-temp-stk p0)))) (exp 2 (p-word-size p0)))
   (not (zerop (untag (top (cdr (p-temp-stk p0))))))
   (equal (car (top (cddr (p-temp-stk p0)))) 'addr)
   (listp (untag (top (p-temp-stk p0))))
   (listp (untag (top (cddr (p-temp-stk p0)))))
   (equal (cdr (untag (top (p-temp-stk p0)))) 0)
   (equal (cdr (untag (top (cddr (p-temp-stk p0))))) 0)
   (definedp (car (untag (top (p-temp-stk p0)))) (p-data-segment p0))
   (definedp (car (untag (top (cddr (p-temp-stk p0))))) (p-data-segment p0))
   (not (equal (car (untag (top (p-temp-stk p0))))
               (car (untag (top (cddr (p-temp-stk p0)))))))
   (nat-list-piton (array (car (untag (top (cddr (p-temp-stk p0))))) (p-data-segment p0))
                   (p-word-size p0))
   (array-pitonp (array (car (untag (top (p-temp-stk p0)))) (p-data-segment p0))
                 (untag (top (cdr (p-temp-stk p0)))))
   (equal (length (array (car (untag (top (cddr (p-temp-stk p0))))) (p-data-segment p0)))
          (untag (top (cdr (p-temp-stk p0)))))
   (equal (assoc 'delay (p-prog-segment p0))
          (delay-program))
```

```
   (equal (assoc 'computer-move (p-prog-segment p0))
          (computer-move-program))
   (equal (assoc 'smart-move (p-prog-segment p0))
          (smart-move-program))
   (equal (assoc 'replace-value (p-prog-segment p0))
          (replace-value-program))
   (equal (assoc 'max-nat (p-prog-segment p0))
          (max-nat-program))
   (equal (assoc 'bv-to-nat-list (p-prog-segment p0))
          (bv-to-nat-list-program))
   (equal (assoc 'nat-to-bv-list (p-prog-segment p0))
          (nat-to-bv-list-program))
   (equal (assoc 'match-and-xor (p-prog-segment p0))
          (match-and-xor-program))
   (equal (assoc 'highest-bit (p-prog-segment p0))
          (highest-bit-program))
   (equal (assoc 'number-with-at-least (p-prog-segment p0))
          (number-with-at-least-program))
   (equal (assoc 'bv-to-nat (p-prog-segment p0))
          (bv-to-nat-program))
   (equal (assoc 'nat-to-bv (p-prog-segment p0))
          (nat-to-bv-program))
   (equal (assoc 'push-1-vector (p-prog-segment p0))
          (push-1-vector-program (p-word-size p0)))
   (equal (assoc 'xor-bvs (p-prog-segment p0))
          (xor-bvs-program))))

(prove-lemma numberp-max-list (rewrite)
             (numberp (max-list x)))

(prove-lemma max-0-means (rewrite)
             (implies
              (nat-list-piton x s)
              (equal
               (equal (max-list (untag-array x)) 0)
               (all-zero-bitvp (untag-array x)))))

(prove-lemma max-list-not-too-big (rewrite)
             (implies
              (and
               (nat-list-piton x s)
               (not (zerop s)))
              (and
               (lessp (max-list (untag-array x)) (exp 2 s))
               (equal (lessp (sub1 (max-list (untag-array x))) (exp 2 s)) t))))

(disable delay-clock)
(disable *1*delay-clock)

(prove-lemma lessp-exp-2-8-hack (rewrite)
             (implies
              (and
               (lessp 7 free)
               (equal x free)
               (lessp val 256))
              (equal (lessp val (exp 2 x)) t)))

(prove-lemma correctness-of-computer-move (rewrite)
             (implies
              (and
               (equal p0 (p-state
                          pc
```

```
                     ctrl-stk
                     (cons wa (cons np (cons s temp-stk)))
                     prog-segment
                     data-segment
                     max-ctrl-stk-size
                     max-temp-stk-size
                     word-size
                     'run))
          (equal (p-current-instruction p0)
                 '(call computer-move))
          (equal state
              (untag-array (array (car (untag s)) data-segment)))
          (equal word-size word-size2)
          (computer-move-input-conditionp p0))
                 (equal
                  (p (p-state
                     pc
                     ctrl-stk
                     (cons wa (cons np (cons s temp-stk)))
                     prog-segment
                     data-segment
                     max-ctrl-stk-size
                     max-temp-stk-size
                     word-size
                     'run)
                     (computer-move-clock state word-size2))
          (p-state
           (add1-addr pc)
           ctrl-stk
           temp-stk
           prog-segment
           (put-assoc
            (tag-array 'nat (computer-move state word-size))
            (car (untag s))
            (if (equal (number-with-at-least state 2) 0)
                data-segment
              (put-assoc
               (if (or (all-zero-bitvp
                        (xor-bvs (nat-to-bv-list state word-size)))
                       (lessp (number-with-at-least state 2) 2))
                   (tag-array 'bitv
                              (nat-to-bv-list state word-size))
                 (tag-array 'bitv
                            (nat-to-bv-list (computer-move state word-size)
                                            word-size)))
               (car (untag wa)) data-segment)))
           max-ctrl-stk-size
           max-temp-stk-size
           word-size
           'run)))
        ((disable smart-move)))

;;;;
;;;;
;;;;
;;;;
;;;; Having proved the behavior of the programs, we now introduce
;;;; the spec to which we've been writing code.
;;;;
;;;;
;;;;
```

```
(defn sum (list)
  (if (listp list)
      (plus (car list) (sum (cdr list)))
    0))

(prove-lemma sum-append (rewrite)
             (equal
              (sum (append x y))
              (plus (sum x) (sum y))))

;; returns a list of states that are valid moves from
(defn all-valid-moves-helper (old val origval new)
  (if (zerop val)
      (if (nlistp new)
          nil
        (all-valid-moves-helper (append old (list origval))
                                (car new) (car new) (cdr new)))
    (cons (append old (cons (sub1 val) new))
          (all-valid-moves-helper old (sub1 val) origval new)))
  ((ord-lessp (cons (add1 (length new)) (fix val)))))

(defn all-valid-moves (x)
  (all-valid-moves-helper nil (car x) (car x) (cdr x)))

(defn max-sum (list)
  (if (listp list)
      (if (lessp (sum (car list))
                 (max-sum (cdr list)))
          (max-sum (cdr list))
        (sum (car list)))
    0))

(prove-lemma nat-listp-append (rewrite)
             (implies
              (properp x)
             (equal
              (nat-listp (append x y) size)
              (and
               (nat-listp x size)
               (nat-listp y size)))))

(enable properp-append)

(defn nat-listp-simple (list)
  (if (listp list)
      (and
       (numberp (car list))
       (nat-listp-simple (cdr list)))
    (equal list nil)))

(prove-lemma nat-listp-simple-append (rewrite)
             (implies
              (properp x)
             (equal
              (nat-listp-simple (append x y))
              (and (nat-listp-simple x) (nat-listp-simple y)))))

(prove-lemma lessp-max-sum-helper nil
             (implies
              (and
               (not (lessp c temp))
               (properp x)
```

```
               (nat-listp-simple (append x (cons c y)))
               (not (all-zero-bitvp (append x (cons c y)))))
              (lessp
               (max-sum (all-valid-moves-helper x temp c y))
               (sum (append x (cons c y))))))

(prove-lemma lessp-max-sum-all-valid-moves (rewrite)
             (implies
              (and
               (nat-listp-simple s)
               (not (all-zero-bitvp s)))
              (equal (lessp (max-sum (all-valid-moves s)) (sum s)) t))
             ((use (lessp-max-sum-helper (c (car s)) (temp (car s)) (x nil)
                                         (y (cdr s))))))

(prove-lemma lessp-sum-max-sum (rewrite)
             (not (lessp (max-sum x) (sum (car x)))))

(disable all-valid-moves)


(defn wsp-measure (state flag)
  (cons (if flag (add1 (sum state)) (add1 (max-sum state)))
        (if flag 0 (length state))))

;; wsp searchs for a successor to  the current state on
;; a path to a guaranteed win.
;; if flag
;;      state is a nim state - return state if all zeros.
;;      Return a successor state not wsp if one exists, f otherwise
;; if not flag
;;      state is a list of states - return member of list if it is
;;      not wsp, f is no such member.

(defn wsp (state flag)
  (if flag
      (if (or (all-zero-bitvp state) (not (nat-listp-simple state)))
          state
        (wsp (all-valid-moves state) f))
    (if (listp state)
        (if (not (wsp (car state) t))
            (car state)
          (wsp (cdr state) f))
      f))
  ((ord-lessp (wsp-measure state flag))))

(defn green-statep (state wordsize)
  (equal
   (zerop (number-with-at-least state 2))
   (all-zero-bitvp (xor-bvs (nat-to-bv-list state wordsize)))))

(defn non-green-in-list (list wordsize)
  (if (listp list)
      (or
       (not (green-statep (car list) wordsize))
       (non-green-in-list (cdr list) wordsize))
    f))

(prove-lemma nat-listp-means-nat-listp-simple (rewrite)
             (implies
              (nat-listp x s)
              (nat-listp-simple x)))
```

```
(prove-lemma xor-bitv-zero-bit-vector (rewrite)
             (implies
              (all-zero-bitvp x)
              (equal (xor-bitv x y)
                     (make-list-from (length x) (fix-bitv y)))))

(prove-lemma fix-bitv-xor-bitv (rewrite)
             (equal (fix-bitv (xor-bitv x y)) (xor-bitv x y)))

(prove-lemma fix-bitv-and-bitv (rewrite)
             (equal (fix-bitv (and-bitv x y)) (and-bitv x y)))

(prove-lemma fix-bitv-xor-bvs (rewrite)
             (equal (fix-bitv (xor-bvs x)) (xor-bvs x)))

(prove-lemma all-zero-bitvp-make-list-from-simple (rewrite)
             (implies
              (all-zero-bitvp x)
              (all-zero-bitvp (make-list-from n x))))

(prove-lemma all-zero-bitvp-xor-bvs-nat-to-bv-list-zeros (rewrite)
             (implies
              (all-zero-bitvp x)
              (all-zero-bitvp (xor-bvs (nat-to-bv-list x s)))))

(prove-lemma number-with-at-least-of-all-zeros (rewrite)
             (implies
              (all-zero-bitvp x)
              (equal (number-with-at-least x m)
                     (if (zerop m) (length x) 0))))

(defn nat-listp-listp (list wordsize)
  (if (listp list)
      (and
       (nat-listp (car list) wordsize)
       (nat-listp-listp (cdr list) wordsize))
    t))

(prove-lemma nat-listp-listp-all-valid-moves-helper (rewrite)
             (implies
              (and
               (nat-listp a s)
               (lessp b (exp 2 s))
               (lessp c (exp 2 s))
               (numberp b)
               (numberp c)
               (nat-listp d s)
               (properp a))
             (nat-listp-listp (all-valid-moves-helper a b c d) s)))

(prove-lemma nat-listp-listp-all-valid-moves (rewrite)
             (implies
               (nat-listp a s)
               (nat-listp-listp (all-valid-moves a) s))
             ((enable all-valid-moves)))

(prove-lemma listp-all-valid-move-helper (rewrite)
             (implies
              (nat-listp-simple d)
              (equal (listp (all-valid-moves-helper a b c d))
                     (or
```

```
                    (not (all-zero-bitvp d))
                    (not (zerop b))))))

(prove-lemma listp-all-valid-move (rewrite)
             (implies
              (nat-listp-simple x)
              (equal (listp (all-valid-moves x))
                     (not (all-zero-bitvp x))))
             ((enable all-valid-moves)))

(prove-lemma number-with-at-least-append (rewrite)
             (equal
              (number-with-at-least (append x y) m)
              (plus
               (number-with-at-least x m)
               (number-with-at-least y m))))

(prove-lemma length-xor-bvs2 (rewrite)
             (implies
              (bit-vectorsp x s)
              (equal (length (xor-bvs x))
                     (if (listp x) (fix s) 0))))

(prove-lemma xor-bitv-xor-bvs-hack (rewrite)
             (implies
              (bit-vectorsp z (length y))
             (equal
              (xor-bitv (xor-bitv y (xor-bvs z)) x)
              (if (listp z)
                  (xor-bitv y (xor-bitv (xor-bvs z) x))
                (xor-bitv y x)))))

(prove-lemma xor-bitv-fix-bitv (rewrite)
             (and
              (equal (xor-bitv (fix-bitv x) y) (xor-bitv x y))
              (equal (xor-bitv x (fix-bitv y)) (xor-bitv x y))))

(prove-lemma xor-bvs-append (rewrite)
             (implies
              (and
               (bit-vectorsp x s)
               (bit-vectorsp y s)
               (numberp s))
             (equal
              (xor-bvs (append x y))
              (if (listp x)
                  (xor-bitv (xor-bvs x) (xor-bvs y))
                (xor-bvs y)))))

(prove-lemma xor-bvs-append-hack (rewrite)
             (implies
              (bit-vectorsp y ws)
             (equal
              (xor-bvs (append (nat-to-bv-list a ws) y))
              (if (listp a)
                  (xor-bitv (xor-bvs (nat-to-bv-list a ws))
                            (xor-bvs y))
                (xor-bvs y))))
             ((use (xor-bvs-append
                    (x (nat-to-bv-list a ws))
                    (s ws)))))
```

```
(prove-lemma nat-to-bv-list-append (rewrite)
             (equal
              (nat-to-bv-list (append x y) s)
              (append
               (nat-to-bv-list x s)
               (nat-to-bv-list y s))))

(prove-lemma bit-vectorp-nat-to-bv (rewrite)
             (equal
              (bit-vectorp (nat-to-bv x s) s2)
              (equal (fix s) (fix s2))))

(prove-lemma fix-bitv-nat-to-bv (rewrite)
             (equal (fix-bitv (nat-to-bv x s)) (nat-to-bv x s)))

(prove-lemma fix-bitv-zero-bit-vector (rewrite)
             (equal
              (fix-bitv (zero-bit-vector x))
              (zero-bit-vector x)))

(prove-lemma all-zero-bitvp-nat-to-bv (rewrite)
             (equal
              (all-zero-bitvp (nat-to-bv x s))
              (or
               (zerop x)
               (zerop s))))

(prove-lemma all-zero-bitvp-xor-bitv-better (rewrite)
             (equal
              (all-zero-bitvp (xor-bitv x y))
              (equal
               (fix-bitv x)
               (make-list-from (length x) (fix-bitv y)))))

(prove-lemma length-xor-bvs-nat-to-bv-list (rewrite)
             (equal
              (length (xor-bvs (nat-to-bv-list x s)))
              (if (listp x) (fix s) 0)))

(prove-lemma fix-bitv-make-list-from (rewrite)
             (equal
              (fix-bitv (make-list-from s x))
              (make-list-from s (fix-bitv x))))


(defn double-sub1-cdr (n1 n2 l)
  (if (or (zerop n1) (zerop n2))
      t
    (double-sub1-cdr (sub1 n1) (sub1 n2) (cdr l))))

(prove-lemma make-list-from-make-list-from (rewrite)
             (equal
              (make-list-from s1 (make-list-from s2 x))
              (if (lessp s2 s1)
                  (append (make-list-from s2 x)
                          (zero-bit-vector (difference s1 s2)))
                (make-list-from s1 x))))

(enable associativity-of-append)

(enable append-cons)
```

```
(prove-lemma properp-xor-bvs (rewrite)
             (properp (xor-bvs x)))

(prove-lemma properp-xor-bitv (rewrite)
             (properp (xor-bitv x y)))

(defn member-number-with-at-least (x min)
  (if (listp x)
      (if (not (zerop (number-with-at-least (car x) min)))
          (car x)
        (member-number-with-at-least (cdr x) min))
    f))

(prove-lemma xor-bvs-nat-to-bv-list-zerop-ws (rewrite)
             (implies
              (zerop ws)
              (equal (xor-bvs (nat-to-bv-list x ws)) nil)))

(defn nat-listp-listp-simple (x)
  (if (listp x)
      (and
       (nat-listp-simple (car x))
       (nat-listp-listp-simple (cdr x)))
    (equal x nil)))

(prove-lemma non-green-in-list-zerop-ws (rewrite)
             (implies
              (and
               (zerop ws)
               (nat-listp-listp-simple x))
              (iff
               (non-green-in-list x ws)
               (member-number-with-at-least x 2))))

(prove-lemma nat-listp-listp-simple-means-properp (rewrite)
             (implies
              (nat-listp-listp-simple x)
              (properp x)))

(prove-lemma nat-listp-simple-means-properp (rewrite)
             (implies
              (nat-listp-simple x)
              (properp x)))

(prove-lemma make-list-from-xor-bvs-nat-to-bv-list (rewrite)
             (implies
              (equal (fix ws) (fix s))
              (equal
               (make-list-from ws (xor-bvs (nat-to-bv-list x s)))
               (if (listp x)
                   (xor-bvs (nat-to-bv-list x s))
                 (zero-bit-vector ws)))))


(prove-lemma last-xor-bitv (rewrite)
             (equal
              (last (xor-bitv x y))
              (if (listp x)
                  (xor-bit (last x) (nth (sub1 (length x)) y))
                0)))

(prove-lemma last-one-bit-vector (rewrite)
```

```
             (equal (last (one-bit-vector x)) 1))

(prove-lemma nth-as-last (rewrite)
             (implies
              (equal (add1 n) (length x))
              (equal (nth n x) (last x))))

(prove-lemma listp-nat-to-bv (rewrite)
             (equal
              (listp (nat-to-bv x s))
              (not (zerop s))))

(enable remainder-plus-x-x-2)

(prove-lemma last-nat-to-bv (rewrite)
             (equal
              (last (nat-to-bv x s))
              (if  (or (zerop s)
                       (and
                        (lessp x (exp 2 s))
                        (equal (remainder x 2) 0)))
                  0
                1)))

(prove-lemma fix-bitv-one-bit-vector (rewrite)
             (equal
              (fix-bitv (one-bit-vector x))
              (one-bit-vector x)))

(prove-lemma nat-to-bv-1 (rewrite)
             (equal
              (nat-to-bv 1 x)
              (if (zerop x) nil (one-bit-vector x))))

(prove-lemma last-zero-bit-vector (rewrite)
             (equal (last (zero-bit-vector x)) 0))

(prove-lemma nat-to-bv-2 (rewrite)
             (equal
              (nat-to-bv x 2)
              (if (zerop x)
                  (list 0 0)
                (if (equal x 1)
                    (list 0 1)
                  (if (equal x 2)
                      (list 1 0)
                    (list 1 1)))))
             ((expand (nat-to-bv x 2))))

(prove-lemma all-zero-bitvp-all-but-last-nat-to-bv (rewrite)
             (equal
              (all-zero-bitvp (all-but-last (nat-to-bv x s)))
              (or
               (lessp x 2)
               (lessp s 2))))

(prove-lemma xor-bvs-of-list-of-0s-and-1s (rewrite)
             (implies
              (zerop (number-with-at-least c 2))
              (equal
               (xor-bvs (nat-to-bv-list c ws))
               (if (or (nlistp c) (zerop ws)) nil
```

```
                (if (equal (remainder (sum c) 2) 0)
                    (zero-bit-vector ws)
                  (one-bit-vector ws))))))

(prove-lemma equal-nat-to-bv-nlistp (rewrite)
             (implies
              (nlistp x)
              (equal
               (equal x (nat-to-bv y s))
               (and (equal x nil) (zerop s)))))

(prove-lemma different-lengths-means-different-hack nil
             (implies
              (not (equal (fix s1) (fix s2)))
              (not (equal (length (nat-to-bv x s1))
                          (length (nat-to-bv y s2))))))

(prove-lemma different-lengths-means-different (rewrite)
             (implies
              (not (equal (fix s1) (fix s2)))
              (not (equal (nat-to-bv x s1) (nat-to-bv y s2))))
             ((use (different-lengths-means-different-hack))
              (disable-theory t)
              (enable-theory ground-zero)))

(defn nat-to-bv-induct (x y s1 s2)
  (if (zerop s1) t
    (nat-to-bv-induct
     (if (lessp x (exp 2 (sub1 s1))) x
       (difference x (exp 2 (sub1 s1))))
     (if (lessp y (exp 2 (sub1 s2))) y
       (difference y (exp 2 (sub1 s2)) ))
     (sub1 s1)
     (sub1 s2))))


(defn all-ones-vector (x)
  (if (zerop x)
      nil
    (cons 1 (all-ones-vector (sub1 x)))))

(prove-lemma not-lessp-exp-means-all-ones  (rewrite)
             (implies
              (not (lessp x (sub1 (exp 2 s))))
              (equal (nat-to-bv x s) (all-ones-vector s))))

(prove-lemma lessp-sub1-plus-sub1-hack (rewrite)
             (implies
              (not (zerop y))
             (equal
              (lessp (sub1 (plus x y))
                     (plus (sub1 y) z))
              (lessp x z))))

(prove-lemma equal-cons-zero-bit-vector-nat-to-bv (rewrite)
             (equal
              (equal (cons 0 (zero-bit-vector x)) (nat-to-bv a b))
              (and
               (equal (add1 x) (fix b))
               (zerop a))))

(prove-lemma equal-all-ones-vector-all-ones-vector (rewrite)
```

```
             (equal
              (equal (all-ones-vector x) (all-ones-vector y))
              (equal (fix x) (fix y)))
             ((induct (double-sub1-cdr x y l))))

(prove-lemma equal-all-ones-vector-cons (rewrite)
             (equal
              (equal (all-ones-vector x) (cons a b))
              (and
               (not (zerop x))
               (equal a 1)
               (equal (all-ones-vector (sub1 x)) b))))

(prove-lemma different-lengths-obvious nil
             (implies
              (equal x y)
              (equal (length x) (length y))))

(prove-lemma equal-all-ones-vector-nlistp (rewrite)
             (implies
              (nlistp x)
              (equal
               (equal (all-ones-vector y) x)
               (and
                (equal x nil)
                (zerop y)))))


(prove-lemma length-all-ones-vector (rewrite)
             (equal (length (all-ones-vector x)) (fix x)))

(prove-lemma different-lengths-hack (rewrite)
             (implies
              (not (equal (fix x) (fix y)))
              (not (equal (all-ones-vector x) (nat-to-bv a y))))
             ((use (different-lengths-obvious
                    (x (all-ones-vector x))
                    (y (nat-to-bv a y))))
              (disable equal-all-ones-vector-nlistp)))

(prove-lemma lessp-difference-arg1 (rewrite)
             (implies
              (not (lessp x y))
              (equal (lessp (difference x y) z)
                     (lessp x (plus y z)))))

(prove-lemma equal-difference (rewrite)
             (implies
              (not (lessp x y))
              (equal
               (equal (difference x y) z)
               (and
                (equal (fix x) (plus y z))
                (numberp z)))))

(prove-lemma equal-exp (rewrite)
             (implies
              (equal (fix a) (fix b))
              (equal
               (equal (exp a c) (exp b d))
               (or
                (equal a 1)
```

```
                (and (zerop a) (equal (zerop c) (zerop d)))
                (equal (fix c) (fix d)))))
             ((induct (double-sub1-cdr c d l))))

(prove-lemma equal-all-ones-nat-to-bv (rewrite)
             (equal
              (equal (all-ones-vector x) (nat-to-bv a b))
              (and
               (equal (fix x) (fix b))
               (not (lessp a (sub1 (exp 2 b))))))
             ((induct (nat-to-bv-induct q a x b))))

(prove-lemma equal-nat-to-bv-nat-to-bv (rewrite)
             (equal
              (equal (nat-to-bv x s1) (nat-to-bv y s2))
              (and
               (equal (fix s1) (fix s2))
               (or
                (equal (fix x) (fix y))
                (and (not (lessp x (sub1 (exp 2 s1))))
                     (not (lessp y (sub1 (exp 2 s1))))))))
             ((induct (nat-to-bv-induct x y s1 s2))))

(prove-lemma listp-xor-bvs (rewrite)
             (equal
              (listp (xor-bvs x))
              (listp (car x))))

(prove-lemma car-nat-to-bv-list (rewrite)
             (equal
              (car (nat-to-bv-list x s))
              (if (listp x)
                  (nat-to-bv (car x) s)
                0)))

;; from later version of naturals that is used in this proof
(prove-lemma quotient-difference
             (rewrite)
             (equal (quotient (difference x y) z)
                    (if (lessp (remainder x z)
                               (remainder y z))
                        (sub1 (difference (quotient x z)
                                          (quotient y z)))
                        (difference (quotient x z)
                                    (quotient y z))))
             ((disable quotient-difference1 quotient-difference2 quotient-difference3)
              (induct (quotient y z))))

(enable DIFFERENCE-X-SUB1-X)

(prove-lemma all-but-last-nat-to-bv (rewrite)
             (equal
              (all-but-last (nat-to-bv x s))
              (if (zerop s)
                  nil
                (nat-to-bv (quotient x 2) (sub1 s)))))

(prove-lemma equal-last-xor-bvs-1 (rewrite)
             (equal
              (equal (last (xor-bvs x)) 1)
              (not (equal (last (xor-bvs x)) 0))))
```

```
(prove-lemma not-green-state-means (rewrite)
             (implies
              (and
               (not (green-statep (append a (cons b c)) ws))
               (lessp d (exp 2 ws))
               (lessp b d))
              (equal
               (green-statep (append a (cons d c)) ws)
               (or (not (zerop ws))
                   (zerop (number-with-at-least
                           (append a (cons d c)) 2))))))

(prove-lemma green-in-list-all-valid-moves-helper nil
             (implies
              (and
               (nat-listp a ws)
               (nat-listp d ws)
               (numberp c)
               (lessp c (exp 2 ws))
               (properp a)
               (non-green-in-list
                (all-valid-moves-helper a b c d) ws)
               (not (lessp c b))
               (numberp b))
              (green-statep (append a (cons c d)) ws))
             ((disable green-statep)))

(prove-lemma green-in-list-all-valid-moves-means (rewrite)
             (implies
              (and
               (nat-listp x ws)
               (non-green-in-list (all-valid-moves x) ws))
              (green-statep x ws))
             ((disable green-statep)
              (enable all-valid-moves)
              (use (green-in-list-all-valid-moves-helper
                    (a nil) (b (car x)) (c (car x)) (d (cdr x))))))

(defn valid-movep (s1 s2)
  (if (and (listp s1) (listp s2))
      (or
       (and
        (lessp (car s2) (car s1))
        (numberp (car s2))
        (equal (cdr s1) (cdr s2)))
       (and
        (equal (car s1) (car s2))
        (valid-movep (cdr s1) (cdr s2))))
    f))

(defn match-member (m list)
  (if (listp list)
      (if (not (all-zero-bitvp (and-bitv (car list) m)))
          (car list)
        (match-member m (cdr list)))
    f))

(prove-lemma xor-bvs-match-and-xor (rewrite)
             (implies
              (bit-vectorsp list (length value))
             (equal
              (xor-bvs (match-and-xor list match value))
```

```
              (if (match-member match list)
                  (xor-bitv value (xor-bvs list))
                (xor-bvs list)))))

(defn remove-highest-bits (x)
  (if (listp x)
      (cons (cdar x) (remove-highest-bits (cdr x)))
    nil))

(prove-lemma car-remove-highest-bits (rewrite)
             (equal
              (car (remove-highest-bits x))
              (cdr (car x))))

(prove-lemma equal-car-highest-bit-1 (rewrite)
             (equal
              (equal (car (highest-bit x)) 1)
              (equal
               (highest-bit x)
               (cons 1 (zero-bit-vector (sub1 (length x)))))))

(prove-lemma car-xor-bitv (rewrite)
             (equal
              (car (xor-bitv x y))
              (if (listp x)
                  (xor-bit (car x) (car y))
                0)))

(prove-lemma match-member-cons-0 (rewrite)
             (iff
              (match-member (cons 0 x) y)
              (match-member x (remove-highest-bits y))))

(prove-lemma match-member-cons (rewrite)
             (implies
              (and
               (bit-vectorsp v (length (cons a b)))
               (not (equal (car (xor-bvs v)) 0)))
              (iff
               (match-member (cons a b) v)
               (or
                (not (equal a 0))
                (match-member b (remove-highest-bits v))))))

(prove-lemma equal-highest-bit-cons-1 (rewrite)
             (equal
              (equal (highest-bit x) (cons 1 y))
              (and
               (not (equal (car x) 0))
               (equal y (zero-bit-vector (sub1 (length x)))))))

(prove-lemma length-cdr-xor-bitv (rewrite)
             (equal
              (length (cdr (xor-bitv x y)))
              (length (cdr x))))

(prove-lemma length-fix-bitv (rewrite)
             (equal (length (fix-bitv x)) (length x)))

(prove-lemma length-cdr-xor-bvs (rewrite)
             (implies
              (bit-vectorsp x s)
```

```
             (equal
              (length (cdr (xor-bvs x)))
              (if (listp x) (sub1 s) 0))))

(defn highest-bits-induct (x s)
  (if (listp x)
      (if (listp (car x))
          (if (equal (car (highest-bit (xor-bvs x))) 1)
              t
            (highest-bits-induct (remove-highest-bits x) (sub1 s)))
        t)
    t)
  ((lessp (length (car x)))))

(prove-lemma bit-vectorsp-remove-highest-bits (rewrite)
             (implies
              (bit-vectorsp x (add1 s))
              (bit-vectorsp (remove-highest-bits x) s)))

(prove-lemma xor-bvs-remove-highest-bits (rewrite)
             (implies
              (bit-vectorsp x s)
             (equal
              (xor-bvs (remove-highest-bits x))
              (if (and (listp x) (not (zerop s)))
                  (cdr (xor-bvs x))
                nil))))

(prove-lemma match-member-highest-bit-xor-bvs nil
             (implies
              (bit-vectorsp x s)
              (iff
              (match-member (highest-bit (xor-bvs x)) x)
              (not (all-zero-bitvp (xor-bvs x)))))
             ((induct (highest-bits-induct x s))))

(prove-lemma match-member-highest-bit-xor-bvs-rewrite (rewrite)
             (implies
              (bit-vectorsp x (length (xor-bvs x)))
              (iff
              (match-member (highest-bit (xor-bvs x)) x)
              (not (all-zero-bitvp (xor-bvs x)))))
             ((use (match-member-highest-bit-xor-bvs
                    (s (length (xor-bvs x)))))))

(prove-lemma bit-vectorsp-nat-to-bv-list-better (rewrite)
             (equal
              (bit-vectorsp (nat-to-bv-list x size) size2)
              (or
               (nlistp x)
               (equal (fix size) (fix size2)))))

(prove-lemma match-member-at-least-min-means (rewrite)
             (implies
              (and
               (match-member y (nat-to-bv-list x ws))
               (not (lessp
                     (bv-to-nat
                      (match-member y (nat-to-bv-list x ws)))
                     min)))
              (not (equal (number-with-at-least x min) 0))))
```

```
(prove-lemma bit-vectorp-highest-bit-xor-bvs (rewrite)
             (equal
              (bit-vectorp (highest-bit (xor-bvs x)) ws)
              (bit-vectorp (xor-bvs x) ws)))

(prove-lemma number-with-at-least-match-and-xor (rewrite)
             (implies
              (and
               (nat-listp x ws)
               (bit-vectorp y ws)
               (bit-vectorp z ws))
              (equal
               (number-with-at-least
                (bv-to-nat-list (match-and-xor
                                 (nat-to-bv-list x ws)
                                 y z))
                min)
              (if (match-member y (nat-to-bv-list x ws))
                  (difference
                   (plus
                    (number-with-at-least x min)
                    (if (lessp
                         (bv-to-nat
                          (xor-bitv z
                           (match-member y (nat-to-bv-list x ws))))
                         min) 0 1))
                   (if (lessp
                        (bv-to-nat
                         (match-member y (nat-to-bv-list x ws)))
                        min) 0 1))
                (number-with-at-least x min)))))

(prove-lemma number-with-at-least-replace-value (rewrite)
             (equal
              (number-with-at-least (replace-value x e v) min)
              (if (member e x)
                  (difference
                   (plus
                    (number-with-at-least x min)
                    (if (lessp v min) 0 1))
                   (if (lessp e min) 0 1))
                (number-with-at-least x min))))

(prove-lemma max-list-means-number-0 (rewrite)
             (implies
              (and
               (equal (max-list x) n)
               (lessp n m))
              (equal (number-with-at-least x m) 0)))

(prove-lemma member-max-list (rewrite)
             (implies
              (nat-listp-simple x)
              (equal
               (member (max-list x) x)
               (listp x))))

(prove-lemma listp-replace-value (rewrite)
             (equal
              (listp (replace-value x e v))
              (listp x)))
```

```
(prove-lemma member-means-lessp-sum (rewrite)
             (implies
              (member e x)
              (not (lessp (sum x) e))))

(prove-lemma sum-replace-value (rewrite)
             (equal
              (sum (replace-value x e v))
              (if (member e x)
                  (difference (plus (sum x) v) e)
                (sum x))))


(prove-lemma remainder-difference-2 (rewrite)
             (equal
              (remainder (difference x y) 2)
              (if (lessp x y) 0
                (if (equal (remainder x 2) (remainder y 2))
                    0
                  1))))

(prove-lemma lessp-max-list (rewrite)
             (not (lessp (sum x) (max-list x))))

(prove-lemma remainder-plus-remainder (rewrite)
             (and
              (equal (remainder (plus (remainder x y) z) y)
                     (remainder (plus x z) y))
              (equal (remainder (plus z (remainder x y)) y)
                     (remainder (plus x z) y))))

(prove-lemma remainder-plus-remainder2 (rewrite)
             (and
              (equal (remainder (plus z (plus a (remainder x y))) y)
                     (remainder (plus z (plus a x)) y))
              (equal (remainder (plus z (plus (remainder x y) a)) y)
                     (remainder (plus z (plus a x)) y)))
             ((use (remainder-plus-remainder (z (plus z a))))
              (disable remainder-plus-remainder)))

(prove-lemma lessp-max-list-from-number-with-at-least (rewrite)
             (implies
              (and
               (equal (number-with-at-least x m) 0)
               (not (zerop m)))
              (lessp (max-list x) m)))

(prove-lemma number-with-at-least-as-sum (rewrite)
             (implies
              (zerop (number-with-at-least x 2))
              (equal
               (number-with-at-least x 1)
               (sum x))))

(prove-lemma equal-remainder-add1-2 (rewrite)
             (equal
              (equal (remainder (add1 x) 2) (remainder (add1 y) 2))
              (equal (remainder x 2) (remainder y 2))))

(prove-lemma remainder-plus-sum-number-hack (rewrite)
             (implies
              (equal (number-with-at-least x 2) 1)
```

```
 (equal
  (remainder (plus (sum x) (number-with-at-least x 1)) 2)
  (remainder (add1 (max-list x)) 2))))

(prove-lemma equal-remainder-add1 (rewrite)
             (equal
              (equal (remainder (add1 x) y) (remainder x y))
              (equal y 1)))

(prove-lemma max-0-means-sum-0 (rewrite)
             (equal
              (equal (sum x) 0)
              (equal (max-list x) 0)))

(prove-lemma max-0-means-all-zero-bitvp (rewrite)
             (implies
              (and
               (equal (max-list x) 0)
               (nat-listp-simple x))
              (all-zero-bitvp x)))

(prove-lemma remainder-add1-2 (rewrite)
             (and
              (equal
               (equal (remainder (add1 x) 2) 0)
               (equal (remainder x 2) 1))
              (equal
               (equal (remainder (add1 x) 2) 1)
               (equal (remainder x 2) 0))))

(prove-lemma remainder-sub1-2 (rewrite)
              (implies
               (not (zerop x))
               (and
                (equal
                 (equal (remainder (sub1 x) 2) 0)
                 (equal (remainder x 2) 1))
                (equal
                 (equal (remainder (sub1 x) 2) 1)
                 (equal (remainder x 2) 0)))))

(prove-lemma equal-x-remainder-sub1-x (rewrite)
             (equal
              (equal (remainder (sub1 x) y) x)
              (equal x 0)))

(prove-lemma computer-move-makes-non-green nil
             (implies
              (and
               (green-statep x ws)
               (nat-listp x ws)
               (listp x)
               (not (zerop ws))
               (not (all-zero-bitvp x)))
              (not (green-statep (computer-move x ws) ws)))
             ((disable nat-to-bv bv-to-nat lessp-number-with-at-least)))

(prove-lemma nat-listp-simplify (rewrite)
             (implies
              (zerop ws)
              (equal (nat-listp x ws)
                     (and
```

```
                      (properp x)
                      (all-zero-bitvp x)))))

(prove-lemma computer-move-makes-non-green-rewrite (rewrite)
             (implies
              (and
               (green-statep x ws)
               (nat-listp x ws)
               (not (all-zero-bitvp x)))
              (not (green-statep (computer-move x ws) ws)))
             ((use (computer-move-makes-non-green))
              (disable-theory t)
              (enable nat-listp all-zero-bitvp nat-listp-simplify)
              (enable-theory ground-zero)))

(defn make-properp (x)
  (if (listp x)
      (cons (car x) (make-properp (cdr x)))
    nil))

(prove-lemma properp-make-properp (rewrite)
             (implies
              (properp x)
              (equal (make-properp x) x)))

(prove-lemma replace-value-simplify (rewrite)
             (implies
              (not (member x y))
              (equal (replace-value y x z) (make-properp y))))

(prove-lemma member-make-properp (rewrite)
             (equal
              (member x (make-properp y))
              (member x y)))

(prove-lemma valid-movep-x-x (rewrite)
             (not (valid-movep x x)))

(prove-lemma valid-movep-replace-value (rewrite)
             (implies
              (properp x)
             (equal
              (valid-movep x (replace-value x y z))
              (and
               (member y x)
               (lessp z y)
               (numberp z)))))

(prove-lemma number-with-at-least-max-list (rewrite)
             (implies
              (and
               (equal (number-with-at-least x m) v)
               (lessp 0 v))
              (not (lessp (max-list x) m))))

(prove-lemma valid-movep-match-and-xor (rewrite)
             (implies
              (and
               (nat-listp x ws)
               (bit-vectorp y ws)
               (bit-vectorp z ws))
             (equal
```

```
              (valid-movep x
                 (bv-to-nat-list
                  (match-and-xor (nat-to-bv-list x ws) y z)))
              (and
               (match-member y (nat-to-bv-list x ws))
               (lessp
                (bv-to-nat (xor-bitv z
                             (match-member y (nat-to-bv-list x ws))))
                (bv-to-nat (match-member y (nat-to-bv-list x ws)))))))
             ((induct (length x))))

(defn lessp-bv (x y)
  (if (and (listp x) (listp y))
      (if (equal (fix-bit (car x)) (fix-bit (car y)))
          (lessp-bv (cdr x) (cdr y))
        (equal (car x) 0))
    f))

(prove-lemma lessp-bv-to-nat (rewrite)
             (lessp (bv-to-nat x) (exp 2 (length x))))

(prove-lemma lessp-as-lessp-bv (rewrite)
             (implies
              (equal (length x) (length y))
              (equal
               (lessp (bv-to-nat x) (bv-to-nat y))
               (lessp-bv x y))))

(prove-lemma fix-bitv-highest-bit (rewrite)
             (equal (fix-bitv (highest-bit x)) (highest-bit x)))

(prove-lemma properp-highest-bit (rewrite)
             (properp (highest-bit x)))

(prove-lemma bit-vectorp-fix-bitv (rewrite)
             (equal
              (bit-vectorp (fix-bitv x) s)
              (equal (length x) (fix s))))

(prove-lemma lessp-bv-xor-bitv (rewrite)
             (implies
              (equal (length x) (length y))
              (equal
               (lessp-bv (xor-bitv x y) y)
               (not (all-zero-bitvp (and-bitv y (highest-bit x)))))))

(prove-lemma length-match-member-nat-to-bv-list (rewrite)
             (equal
              (length (match-member a (nat-to-bv-list x ws)))
              (if (match-member a (nat-to-bv-list x ws))
                  (fix ws)
                0)))

(prove-lemma bit-vectorsp-remove-highest-bits2 (rewrite)
             (implies
              (bit-vectorsp x s1)
              (equal
               (bit-vectorsp (remove-highest-bits x) s2)
               (or
                (equal (add1 s2) s1)
                (not (listp x))))))
```

```
(prove-lemma match-member-high-bit-xor-bvs-helper nil
             (implies
              (bit-vectorsp x ws)
             (iff
              (match-member (highest-bit (xor-bvs x)) x)
              (not (all-zero-bitvp (xor-bvs x)))))
             ((induct (highest-bits-induct x ws))))

(prove-lemma match-member-high-bit-xor-bvs (rewrite)
             (iff
              (match-member
               (highest-bit (xor-bvs (nat-to-bv-list y ws)))
               (nat-to-bv-list y ws))
              (not (all-zero-bitvp (xor-bvs (nat-to-bv-list y ws)))))
             ((use (match-member-high-bit-xor-bvs-helper
                    (x (nat-to-bv-list y ws))))
              (disable-theory t)
              (enable-theory ground-zero)
              (enable bit-vectorsp-nat-to-bv-list-better)))

(prove-lemma length-match-member (rewrite)
             (implies
              (match-member a (nat-to-bv-list x ws))
              (equal
               (length (match-member a (nat-to-bv-list x ws)))
               (fix ws))))

(prove-lemma all-zero-bitvp-and-match-member (rewrite)
             (implies
              (bit-vectorsp b (length a))
             (equal
              (all-zero-bitvp (and-bitv a (match-member a b)))
              (not (match-member a b))))
             ((induct (match-member a b))))

(prove-lemma valid-movep-computer-move-helper nil
           (implies
            (and
             (nat-listp x ws)
             (not (all-zero-bitvp x))
             (not (zerop ws))
             (listp x))
            (valid-movep x (computer-move x ws)))
           ((disable all-zero-bitvp nat-to-bv bv-to-nat-list match-and-xor)))

;;; PART OF SPECIFICATION
(prove-lemma valid-movep-computer-move (rewrite)
           (implies
            (and
             (nat-listp x ws)
             (not (all-zero-bitvp x)))
            (valid-movep x (computer-move x ws)))
           ((use (valid-movep-computer-move-helper))
            (disable-theory t)
            (enable-theory ground-zero)
            (enable nat-listp-simplify all-zero-bitvp)))

(prove-lemma nthcdr-cdr (rewrite)
             (equal
              (nthcdr n (cdr x))
              (cdr (nthcdr n x))))
```

```
(prove-lemma make-list-from-append (rewrite)
             (equal
              (make-list-from n (append a b))
              (if (lessp (length a) n)
                  (append a (make-list-from (difference n (length a)) b))
                (make-list-from n a))))

(prove-lemma make-list-from-simplify-better (rewrite)
             (implies
              (equal n (length x))
              (equal (make-list-from n x) (make-properp x))))

(enable length-cons)

(prove-lemma cdr-nthcdr-cons (rewrite)
             (equal
              (cdr (nthcdr n (cons a b)))
              (nthcdr n b)))

(prove-lemma equal-append-2 (rewrite)
             (equal
              (equal x (append a b))
              (and
               (not (lessp (length x) (length a)))
               (equal (make-list-from (length a) x) (make-properp a))
               (equal (nthcdr (length a) x) b)))
             ((induct (double-cdr-induct x a))
              (disable length)))

(prove-lemma member-cons-all-valid-moves-helper1 (rewrite)
             (implies
              (listp a)
             (equal
              (member (cons x y) (all-valid-moves-helper a b c d))
              (and
               (equal x (car a))
               (member y (all-valid-moves-helper (cdr a) b c d))))))

(prove-lemma member-all-valid-moves-means-prefix (rewrite)
             (implies
              (properp a)
              (implies
               (member x (all-valid-moves-helper a b c d))
               (equal (make-list-from (length a) x) a))))

(prove-lemma equal-nthcdr-cons (rewrite)
             (implies
              (not (lessp n (length x)))
              (not (equal (nthcdr n x) (cons a b)))))

(prove-lemma lessp-length-simple-member-all-valid-moves (rewrite)
             (implies
              (not (lessp (length a) (length x)))
              (not (member x (all-valid-moves-helper a b c d)))))

(prove-lemma nth-cons (rewrite)
             (equal (nth n (cons a b))
                    (if (zerop n) a (nth (sub1 n) b))))

(prove-lemma nth-1 (rewrite)
             (equal (nth 1 x) (cadr x)))
```

```
(prove-lemma get-as-nth (rewrite)
             (equal (get n x) (nth n x)))

(prove-lemma equal-cons-make-properp (rewrite)
             (equal
              (equal (cons a b) (make-properp x))
              (and
               (listp x)
               (equal a (car x))
               (equal b (make-properp (cdr x))))))

(prove-lemma nthcdr-cons-make-list-from-hack (rewrite)
             (equal
              (nthcdr n (cons a (make-list-from n z)))
              (if (zerop n) (list a) (list (nth (sub1 n) z)))))

(prove-lemma lessp-sub1-as-equal (rewrite)
              (implies
               (lessp a b)
               (equal (lessp a (sub1 b)) (not (equal (add1 a) b)))))

(prove-lemma equal-nthcdr-cons-better (rewrite)
             (equal
              (equal (nthcdr n x) (cons a b))
              (and
               (lessp n (length x))
               (equal (nth n x) a)
               (equal (nthcdr (add1 n) x) b))))

(prove-lemma member-all-valid-moves-helper (rewrite)
             (implies
              (and
               (nat-listp-simple a)
               (nat-listp-simple d)
               (numberp b)
               (numberp c))
             (equal
              (member x (all-valid-moves-helper a b c d))
              (and
               (equal (make-list-from (length a) x) (make-properp a))
               (or
                (and
                 (lessp (nth (length a) x) b)
                 (numberp (nth (length a) x))
                 (equal (nthcdr (add1 (length a)) x) d))
                (and
                 (equal (nth (length a) x) c)
                 (valid-movep d (cdr (nthcdr (length a) x)))))))))

(prove-lemma member-all-valid-moves (rewrite)
             (implies
              (nat-listp-simple x)
             (equal
              (member y (all-valid-moves x))
              (valid-movep x y)))
             ((enable all-valid-moves)))

(prove-lemma valid-movep-and-makes-nongreen-means (rewrite)
             (implies
              (and
               (member x y)
               (not (green-statep x ws)))
```

```
              (non-green-in-list y ws))
             ((disable green-statep)))

(prove-lemma green-means-non-green-in-valid-moves (rewrite)
             (implies
              (and
               (nat-listp s ws)
               (not (all-zero-bitvp s))
               (green-statep s ws))
              (non-green-in-list (all-valid-moves s) ws))
             ((use (valid-movep-and-makes-nongreen-means
                    (x (computer-move s ws)) (y (all-valid-moves s))))
              (disable green-statep computer-move)))

(prove-lemma green-in-list-all-valid-moves (rewrite)
             (implies
              (and
               (nat-listp s ws)
               (not (all-zero-bitvp s)))
              (iff
               (non-green-in-list (all-valid-moves s) ws)
               (green-statep s ws)))
             ((disable green-statep)))

(prove-lemma sum-when-all-zero (rewrite)
             (implies
              (all-zero-bitvp x)
              (equal (sum x) 0)))

(prove-lemma green-statep-all-zero-bitvp (rewrite)
             (implies
              (all-zero-bitvp x)
              (green-statep x s)))

(prove-lemma wsp-green-state-proof nil
             (implies
              (and
               (or
                (and flag (nat-listp s wordsize))
                (and (not flag) (nat-listp-listp s wordsize)))
               (listp s))
              (iff
               (wsp s flag)
               (if flag
                   (green-statep s wordsize)
                 (non-green-in-list s wordsize))))
             ((disable green-statep)))

(prove-lemma wsp-green-state (rewrite)
             (implies
              (nat-listp s wordsize)
              (iff
               (wsp s t)
               (green-statep s wordsize)))
             ((use (wsp-green-state-proof (flag t)))
              (disable green-statep)))

(prove-lemma nat-listp-replace-value (rewrite)
             (implies
              (and
               (nat-listp s ws)
               (lessp new (exp 2 ws))
```

```
                (numberp new))
               (nat-listp (replace-value s y new) ws)))

(prove-lemma nat-listp-bv-to-nat-list (rewrite)
             (implies
              (bit-vectorsp x s)
              (nat-listp (bv-to-nat-list x) s)))

(prove-lemma nat-listp-smart-move nil
             (implies
              (and
               (nat-listp s ws)
               (not (zerop ws)))
              (nat-listp (smart-move s ws) ws))
             ((disable-theory t)
              (enable-theory ground-zero naturals)
              (enable smart-move bit-vectorsp-nat-to-bv-list-better nat-listp-replace-value
                      nat-listp-bv-to-nat-list lessp-exp-simple lessp-remainder-x-exp-x)
              (use (bit-vectorsp-match-and-xor
                    (size ws)
                    (x (nat-to-bv-list s ws))
                    (y (highest-bit (xor-bvs (nat-to-bv-list s ws))))
                    (z (xor-bvs (nat-to-bv-list s ws)))))))

(prove-lemma all-zero-bitvp-max-list (rewrite)
             (implies
              (all-zero-bitvp s)
              (equal (max-list s) 0)))

(prove-lemma replace-value-x-x (rewrite)
             (implies
              (properp x)
              (equal (replace-value x y y) x)))

(prove-lemma smart-move-small-ws (rewrite)
             (implies
              (and
               (nat-listp s ws)
               (zerop ws))
              (equal (smart-move s ws) s)))

(prove-lemma lessp-max-list-from-nat-listp (rewrite)
             (implies
              (nat-listp s ws)
              (lessp (max-list s) (exp 2 ws))))

(prove-lemma nat-listp-computer-move (rewrite)
             (implies
              (nat-listp s ws)
              (nat-listp (computer-move s ws) ws))
             ((use (nat-listp-smart-move))
              (disable-theory t)
              (enable-theory ground-zero)
              (enable lessp-max-list-from-nat-listp smart-move-small-ws computer-move
                      nat-listp-replace-value)))

;; PART OF SPECIFICATION
(prove-lemma computer-move-works (rewrite)
             (implies
              (and
               (nat-listp state ws)
               (not (all-zero-bitvp state))
```

```
               (wsp state t))
              (not (wsp (computer-move state ws) t)))
             ((disable computer-move green-statep nat-listp-computer-move)
              (use (nat-listp-computer-move (s state)))))


(defn nim-piton-ctrl-stk-requirement ()
  25)

(defn nim-piton-temp-stk-requirement ()
  3)

(defn computer-move-implemented-input-conditionp (p0)
 (and
  (listp (p-ctrl-stk p0))
  (lessp 7 (p-word-size p0))
  (lessp 1 (p-word-size p0))  ; useful to prover, but subsumed

;; there is some room on the stacks

  (at-least-morep (length (p-temp-stk p0))
                  (nim-piton-temp-stk-requirement) (p-max-temp-stk-size p0))
  (at-least-morep (p-ctrl-stk-size (p-ctrl-stk p0))
                  (nim-piton-ctrl-stk-requirement) (p-max-ctrl-stk-size p0))

;; the third thing on the stack is an address to the state array
  (equal (car (top (cddr (p-temp-stk p0)))) 'addr)
  (equal (cddr (top (cddr (p-temp-stk p0)))) nil)
  (listp (untag (top (cddr (p-temp-stk p0)))))
  (equal (cdr (untag (top (cddr (p-temp-stk p0))))) 0)
  (definedp (car (untag (top (cddr (p-temp-stk p0))))) (p-data-segment p0))
  (nat-list-piton (array (car (untag (top (cddr (p-temp-stk p0)))))
                         (p-data-segment p0))
                  (p-word-size p0))

;; the second thing on the stack is the length of the state
  (equal (car (top (cdr (p-temp-stk p0)))) 'nat)
  (equal (cddr (top (cdr (p-temp-stk p0)))) nil)
  (equal (length (array (car (untag (top (cddr (p-temp-stk p0)))))
                        (p-data-segment p0)))
         (untag (top (cdr (p-temp-stk p0)))))
  (lessp (untag (top (cdr (p-temp-stk p0)))) (exp 2 (p-word-size p0)))
  (not (zerop (untag (top (cdr (p-temp-stk p0))))))

;; the top thing on the stack is a pointer to an array
;; that is the same size as the state array but distinct
  (equal (car (top (p-temp-stk p0))) 'addr)
  (equal (cddr (top (p-temp-stk p0))) nil)
  (listp (untag (top (p-temp-stk p0))))
  (equal (cdr (untag (top (p-temp-stk p0)))) 0)
  (definedp (car (untag (top (p-temp-stk p0)))) (p-data-segment p0))
  (array-pitonp (array (car (untag (top (p-temp-stk p0))))
                       (p-data-segment p0))
                (untag (top (cdr (p-temp-stk p0)))))
  (not (equal (car (untag (top (p-temp-stk p0))))
              (car (untag (top (cddr (p-temp-stk p0)))))))

;; at least one pile has one match
  (not (all-zero-bitvp
        (untag-array
         (array (car (untag (top (cddr (p-temp-stk p0)))))
                (p-data-segment p0)))))))
```

```
;; cm-prog is the Nim program.  It may be disappointing to see that it is
;; a function of one argument rather than a constant, as programs ought to
;; be.  This is because we wish to use bit vectors in our program, and
;; because of a weakness in the Piton design there is no way to push
;; a bit-vector on the stack without knowing the word-size.  The only
;; subprogram that uses the word-size is push-1-vector, which is a
;; one-line program that pushes a one-vector onto the stack.
(defn cm-prog (word-size)
  (list
   (xor-bvs-program)
   (push-1-vector-program word-size)
   (nat-to-bv-program)
   (bv-to-nat-program)
   (number-with-at-least-program)
   (highest-bit-program)
   (match-and-xor-program)
   (nat-to-bv-list-program)
   (bv-to-nat-list-program)
   (max-nat-program)
   (replace-value-program)
   (smart-move-program)
   (delay-program)
   (computer-move-program)))

(disable computer-move-program)
(disable *1*computer-move-program)

(prove-lemma car-xor-bvs-program (rewrite)
             (equal (car (xor-bvs-program)) 'xor-bvs)
             ((enable xor-bvs-program)))

(prove-lemma car-push-1-vector-program (rewrite)
             (equal (car (push-1-vector-program word-size))
                    'push-1-vector)
             ((enable push-1-vector-program)))

(prove-lemma car-nat-to-bv-program (rewrite)
             (equal (car (nat-to-bv-program)) 'nat-to-bv)
             ((enable nat-to-bv-program)))

(prove-lemma car-bv-to-nat-program (rewrite)
             (equal (car (bv-to-nat-program)) 'bv-to-nat)
             ((enable bv-to-nat-program)))

(prove-lemma car-number-with-at-least-program (rewrite)
             (equal (car (number-with-at-least-program))
                    'number-with-at-least)
             ((enable number-with-at-least-program)))

(prove-lemma car-highest-bit-program (rewrite)
             (equal (car (highest-bit-program)) 'highest-bit)
             ((enable highest-bit-program)))

(prove-lemma car-match-and-xor-program (rewrite)
             (equal (car (match-and-xor-program)) 'match-and-xor)
             ((enable match-and-xor-program)))

(prove-lemma car-nat-to-bv-list-program (rewrite)
             (equal (car (nat-to-bv-list-program))
                    'nat-to-bv-list)
```

```
             ((enable nat-to-bv-list-program)))

(prove-lemma car-bv-to-nat-list-program (rewrite)
             (equal (car (bv-to-nat-list-program)) 'bv-to-nat-list)
             ((enable bv-to-nat-list-program)))

(prove-lemma car-max-nat-program (rewrite)
             (equal (car (max-nat-program)) 'max-nat)
             ((enable max-nat-program)))

(prove-lemma car-replace-value-program (rewrite)
             (equal (car (replace-value-program)) 'replace-value)
             ((enable replace-value-program)))

(prove-lemma car-smart-move-program (rewrite)
             (equal (car (smart-move-program)) 'smart-move)
             ((enable smart-move-program)))

(prove-lemma car-delay-program (rewrite)
             (equal (car (delay-program)) 'delay))

(disable delay-program)

(prove-lemma car-computer-move-program (rewrite)
             (equal (car (computer-move-program)) 'computer-move)
             ((enable computer-move-program)))

(prove-lemma equal-untag-array-tag-array-x-x (rewrite)
             (equal
              (equal (untag-array (tag-array l x)) x)
              (properp x)))

(prove-lemma properp-replace-value (rewrite)
             (implies
              (properp x)
              (properp (replace-value x y z))))

(prove-lemma properp-bv-to-nat-list (rewrite)
             (properp (bv-to-nat-list x)))

(prove-lemma properp-nat-to-bv-list (rewrite)
             (properp (nat-to-bv-list x ws)))

(prove-lemma properp-computer-move (rewrite)
             (implies
              (properp x)
              (properp (computer-move x s)))
             ((disable-theory t)
              (enable-theory ground-zero)
              (enable properp-replace-value properp-bv-to-nat-list smart-move computer-move)))

(prove-lemma properp-untag-array (rewrite)
             (properp (untag-array x)))

(prove-lemma properp-tag-array (rewrite)
             (properp (tag-array l x)))

(prove-lemma computer-move-implemented (rewrite)
             (implies
              (and
               (equal p0 (p-state
                          pc
```

```
                       ctrl-stk
                       (cons wa (cons np (cons s temp-stk)))
                       (append (cm-prog word-size) prog-segment)
                       data-segment
                       max-ctrl-stk-size
                       max-temp-stk-size
                       word-size
                       'run))
            (equal (p-current-instruction p0)
                   '(call computer-move))
            (computer-move-implemented-input-conditionp p0))
           (let ((result
                  (p p0 (computer-move-clock
                         (untag-array (array (car (untag s)) data-segment))
                         word-size))))
             (and
              (equal (p-pc result) (add1-addr pc))
              (equal (p-psw result) 'run)
              (equal (untag-array
                      (array (car (untag s)) (p-data-segment result)))
                     (computer-move
                      (untag-array (array (car (untag s)) data-segment))
                      word-size)))))
             ((disable computer-move computer-move-clock p-current-instruction lessp-max-list
                       max-list all-zero-bitvp sum member-of-natlist-means
                       lessp-sub1-x-y-crock all-zero-bitvp-max-list max-list-not-too-big)))

(prove-lemma nim-piton-space-reasonable (rewrite)
             (not (lessp 1000 (plus (nim-piton-ctrl-stk-requirement)
                                    (nim-piton-temp-stk-requirement)))))

;; bind up defns for presentation purposes
(defn good-non-empty-nim-statep (state ws)
  (and
   (nat-listp state ws)
   (not (all-zero-bitvp state))))

(prove-lemma valid-movep-computer-move-better (rewrite)
             (implies
              (good-non-empty-nim-statep state ws)
              (valid-movep state (computer-move state ws)))
             ((use (valid-movep-computer-move (x state)))
              (disable-theory t)
              (enable good-non-empty-nim-statep)
              (enable-theory ground-zero)))

(prove-lemma computer-move-works-better (rewrite)
             (implies
              (and
               (good-non-empty-nim-statep state ws)
               (wsp state t))
              (not (wsp (computer-move state ws) t)))
             ((use (computer-move-works))
              (disable-theory t)
              (enable good-non-empty-nim-statep)
              (enable-theory ground-zero)))

;;; An initial p-state to run the program on a particular NIM state, then
;;; enter an infinite loop.
(defn example2-computer-move-p-state ()
  (p-state '(pc (main . 0))
           '((nil (pc (main . 0))))
```

```
          nil
          (cons '(main nil nil
                       (push-constant (addr (arr . 0)))
                       (push-constant (nat 4))
                       (push-constant (addr (arr5 . 0)))
                       (call computer-move)
                       (push-constant (nat 1))
                       (push-constant (addr (flag . 0)))
                       (deposit)
                       (dl loop () (jump loop))
                       (ret))
                (cm-prog 32))
          '((arr (nat 15) (nat 4) (nat 7) (nat 1))
            (arr5 (nat 3) (nat 4) (nat 2) (nat 1))
            (flag (nat 0)))
          30
          10
          32
          'run))

;;; Extra event that shows that the program is compilable and loadable
;;; onto FM9001, and that the correctness lemma for the Piton interpreter
;;; therefore holds.   (ref: J's e-mail of 10 April 92.)
(prove-lemma cm-prog-fm9001-loadable nil
             (let ((p0 (example2-computer-move-p-state)))
               (and (proper-p-statep p0)
                    (p-loadablep p0 0)
                    (equal (p-word-size p0) 32)))
             ((enable xor-bvs-program push-1-vector-program nat-to-bv-program
                      bv-to-nat-program number-with-at-least-program highest-bit-program
                      match-and-xor-program nat-to-bv-list-program bv-to-nat-list-program
                      max-nat-program replace-value-program smart-move-program delay-program
                      computer-move-program)))


;;; Some events written by J Moore that produce an FM9001 image
(defn pretty-load1 (p0 offset)
  (i->m (r->i (p->r p0)) nil offset))

(defn pretty-vector1 (lst)
  (cond ((nlistp lst) 0)
        ((car lst) (cons 49 (pretty-vector1 (cdr lst))))
        (t (cons 48 (pretty-vector1 (cdr lst))))))

(defn pretty-vector (lst)
  (pack (cons 66 (pretty-vector1 lst))))

(defn pretty-vector-lst (lst)
  (cond ((nlistp lst) nil)
        (t (cons (pretty-vector (car lst))
                 (pretty-vector-lst (cdr lst))))))

(defn pretty-state (m)
  (list 'fm9001-state
        (pretty-vector-lst (m-regs m))
        (m-c-flg m)
        (m-v-flg m)
        (m-n-flg m)
        (m-z-flg m)
        (pretty-vector-lst (m-mem m))))

(defn pretty-load (p offset) (pretty-state (pretty-load1 p offset)))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; We now prove a timing execution bound on the program.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(prove-lemma xor-bvs-clock-loop-upper (rewrite)
  (equal (xor-bvs-clock-loop numvecs)
 (plus 3 (times numvecs 12))))

(prove-lemma xor-bvs-clock-upper (rewrite)
  (equal (xor-bvs-clock numvecs) (plus 9 (times (sub1 numvecs) 12)))
  ((enable xor-bvs-clock)))


(defn quotient2-sub1-induct (x y)
  (if (zerop x) t
    (quotient2-sub1-induct (quotient x 2) (sub1 y))))

(prove-lemma nat-to-bv-loop-clock-upper (rewrite)
             (implies
              (and
               (lessp value (exp 2 ws))
               (not (zerop ws)))
              (not (lessp (add1 (add1 (add1 (times ws 14))))
                          (nat-to-bv-loop-clock value))))
             ((induct (quotient2-sub1-induct value ws))))

(prove-lemma sub1-plus-add1 (rewrite)
             (equal
              (sub1 (plus (add1 x) y))
              (plus x y)))

(prove-lemma lessp-add1-add1 (rewrite)
             (equal
              (lessp (add1 x) (add1 y))
              (lessp x y)))

(prove-lemma lessp-plus-add1 (rewrite)
             (and
              (equal (lessp (plus (add1 x) y) (plus (add1 a) b))
                     (lessp (plus x y) (plus a b)))
              (equal (lessp (plus (add1 x) y) (add1 a))
                     (lessp (plus x y) a))
              (equal (lessp (add1 x) (plus (add1 a) b))
                     (lessp x (plus a b)))))

(prove-lemma nat-to-bv-clock-upper (rewrite)
  (implies
   (and
    (lessp value (exp 2 ws))
    (not (zerop ws)))
   (not (lessp (plus 12 (times ws 14)) (nat-to-bv-clock value))))
  ((enable nat-to-bv-clock lessp-add1-add1  plus)
   (use (nat-to-bv-loop-clock-upper))
   (disable-theory t)
   (enable-theory naturals)))

(prove-lemma delay-loop-clock-upper (rewrite)
             (equal (delay-loop-clock n) (plus 9 (times 10 (sub1 n)))))

(prove-lemma delay-clock-upper (rewrite)
             (equal (delay-clock n)
```

```
                    (if (zerop n) 10 (times 10 n)))
             ((enable delay-clock)
              (disable delay-loop-clock)))

(disable trailing-zeros)

(prove-lemma lessp-trailing-zeros-length (rewrite)
             (equal
              (lessp (trailing-zeros x) (length x))
              (not (all-zero-bitvp x)))
             ((enable trailing-zeros)))

(prove-lemma bv-to-nat-loop-clock-simple (rewrite)
             (not (lessp (bv-to-nat-loop-clock cb bv) 9))
             ((expand (bv-to-nat-loop-clock cb bv))
              (disable-theory t)
              (enable-theory ground-zero)))

(prove-lemma equal-add1-sub1 (rewrite)
             (equal
              (equal (add1 x) (sub1 y))
              (equal (add1 (add1 x)) y)))

(prove-lemma trailing-zeros-cons (rewrite)
             (equal
              (trailing-zeros (cons a b))
              (if (equal (trailing-zeros b) (length b))
                  (if (equal a 0) (add1 (length b)) (length b))
                (trailing-zeros b)))
             ((enable trailing-zeros)))

(prove-lemma all-zero-bitvp-cdr-append (rewrite)
             (equal
              (all-zero-bitvp (cdr (append x y)))
              (if (listp x)
                  (and (all-zero-bitvp (cdr x)) (all-zero-bitvp y))
                (all-zero-bitvp (cdr y)))))

(prove-lemma trailing-zeros-simple (rewrite)
             (implies
              (all-zero-bitvp x)
              (equal (trailing-zeros x) (length x)))
             ((enable trailing-zeros)))

(prove-lemma equal-trailing-zeros-length-better (rewrite)
             (equal
              (equal (trailing-zeros x) (length x))
              (all-zero-bitvp x))
             ((enable trailing-zeros)))

(prove-lemma lessp-plus (rewrite)
             (and
              (implies
               (lessp a b)
               (and
                (equal (lessp a (plus c b)) t)
                (equal (lessp a (plus b c)) t)))
              (implies
               (not (lessp b a))
               (and
                (equal (lessp (plus b c) a) f)
                (equal (lessp (plus c b) a) f)))))
```

```
(disable lessp-plus)

(prove-lemma lessp-length-trailing (rewrite)
             (equal  (lessp (length x) (trailing-zeros x)) f)
             ((enable trailing-zeros)))

(prove-lemma bv-to-nat-loop-clock-upper (rewrite)
             (not (lessp (plus 11 (times (difference (length cb) (trailing-zeros cb)) 14))
                         (bv-to-nat-loop-clock cb bv)))
             ((induct (bv-to-nat-loop-clock cb bv))
              (disable-theory t)
              (enable-theory ground-zero)
              (enable bv-to-nat-loop-clock length-append trailing-zeros-append
                      trailing-zeros-simple plus-add1-arg2 times-add1 lessp-length-trailing
                      *1*length lessp-plus length-cons bv-to-nat-loop-clock-simple
                      equal-sub1-add1 equal-plus-0 trailing-zeros-cons plus-zero-arg2
                      equal-trailing-zeros-length-better commutativity-of-times
                      times-distributes-over-difference correctness-of-cancel-lessp-times
                      difference-leq-arg1 lessp-trailing-zeros-length all-zero-bitvp)))

(prove-lemma bv-to-nat-clock-upper (rewrite)
  (implies
   (not (zerop ws))
   (not (lessp (plus 19 (times ws 14)) (bv-to-nat-clock ws bv))))
  ((enable bv-to-nat-clock lessp-add1-add1 correctness-of-cancel-lessp-plus
   length-one-bit-vector trailing-zeros-one-bit-vector plus-add1
   one-bit-vector bv-to-nat-loop-clock)
   (disable-theory t)
   (use (bv-to-nat-loop-clock-upper (cb (one-bit-vector ws))))
   (enable-theory ground-zero)))

(prove-lemma difference-sub1-arg1 (rewrite)
             (equal
              (difference (sub1 a) b)
              (sub1 (difference a b))))

(prove-lemma number-with-at-least-clock-loop-upper (rewrite)
              (not (lessp (times (difference (length array) i) 17)
                          (number-with-at-least-clock-loop i min array)))
             ((induct (number-with-at-least-clock-loop i min array))
              (disable-theory t)
              (enable-theory ground-zero)
              (enable number-with-at-least-clock-loop times-add1 plus-zero-arg2
                      difference-leq-arg1 plus-add1-arg2 plus-add1-arg1 difference-sub1
                      times-zero equal-times-0 difference-sub1-arg1)))

(prove-lemma number-with-at-least-clock-upper (rewrite)
  (not (lessp (plus 15 (times (length array) 17))
      (number-with-at-least-clock min array)))
  ((disable-theory t)
   (enable number-with-at-least-clock)
   (enable-theory ground-zero naturals)
   (use (number-with-at-least-clock-loop-upper (i 0)))))


(prove-lemma highest-bit-loop-clock-upper (rewrite)
             (not (lessp (plus 3 (times (difference (length cb) (trailing-zeros cb)) 12))
                         (highest-bit-loop-clock cb bv)))
             ((induct (highest-bit-loop-clock cb bv))
              (disable-theory t)
              (enable-theory ground-zero)
```

```
              (enable highest-bit-loop-clock length-append trailing-zeros-append
                      trailing-zeros-simple plus-add1-arg2 times-add1 lessp-length-trailing
                      *1*length lessp-plus length-cons bv-to-nat-loop-clock-simple
                      equal-sub1-add1 equal-plus-0 trailing-zeros-cons plus-zero-arg2
                      equal-trailing-zeros-length-better commutativity-of-times
                      times-distributes-over-difference correctness-of-cancel-lessp-times
                      difference-leq-arg1 lessp-trailing-zeros-length all-zero-bitvp)))


(prove-lemma highest-bit-clock-upper (rewrite)
  (implies
   (listp bv)
   (not (lessp (plus 9 (times (length bv) 12)) (highest-bit-clock bv))))
  ((enable highest-bit-clock lessp-add1-add1 correctness-of-cancel-lessp-plus
   length-one-bit-vector trailing-zeros-one-bit-vector plus-add1
   equal-length-0
   one-bit-vector highest-bit-loop-clock)
   (disable-theory t)
   (use (highest-bit-loop-clock-upper (cb (one-bit-vector (length bv)))))
   (enable-theory ground-zero)))

(prove-lemma match-and-xor-loop-clock-upper (rewrite)
             (not (lessp (times 16 (length bvs)) (match-and-xor-loop-clock bvs match)))
             ((enable match-and-xor-loop-clock)))

(prove-lemma match-and-xor-clock-upper (rewrite)
             (not (lessp (plus 3 (times 16 (length bvs))) (match-and-xor-clock bvs match)))
             ((enable match-and-xor-clock)))

(prove-lemma lessp-nat-to-bv-loop-clock-rewrite (rewrite)
             (implies
              (and
               (not (lessp a 3))
               (nat-list-piton x ws))
              (equal (lessp (plus a (times 14 ws))
                            (nat-to-bv-loop-clock (cadr (nth n x)))) f))
             ((use (nat-to-bv-loop-clock-upper (value (cadr (nth n x)))))
              (disable nat-to-bv-loop-clock-upper)))

(prove-lemma plus-plus-add1 (rewrite)
             (equal
              (plus x (plus (add1 y) z))
              (plus (add1 x) (plus y z))))

(prove-lemma nat-to-bv-list-hack (rewrite)
              (implies
               (and (numberp v)
                    (lessp v (length x))
                    (not (lessp (difference (times (length x)
                                                   (plus 31 (times 14 ws)))
                                            (plus 31
                                                  (times 14 ws)
                                                  (times v (plus 31 (times 14 ws)))))
                                (nat-to-bv-list-loop-clock (add1 (add1 v))
                                                           (cons z x))))
                    (listp z)
                    (equal (car z) 'nat)
                    (numberp (cadr z))
                    (equal (cddr z) nil)
                    (lessp (cadr z) (exp 2 ws))
                    (nat-list-piton x ws)
                    (not (equal ws 0))
```

```
                (numberp ws)
                (not (equal (length x) 0))
                (lessp v (sub1 (length x))))
           (equal
            (lessp
             (difference (times (length x)
                                (plus 31 (times 14 ws)))
                         (times v (plus 31 (times 14 ws))))
             (plus 2
              (add1
               (add1
                    (plus 9
                          (add1 (plus 14
                                      (nat-to-bv-loop-clock (cadr (nth v x)))
                                      (nat-to-bv-list-loop-clock (add1 (add1 v))
                                                                 (cons z x)))))))))
            f))
          ((use (lessp-nat-to-bv-loop-clock-rewrite (n v) (a 3)))
           (disable-theory t)
           (enable-theory ground-zero)
           (enable commutativity-of-plus associativity-of-plus commutativity2-of-plus
                   plus-add1-arg2 plus correctness-of-cancel-lessp-plus
                   correctness-of-cancel-difference-plus
                   correctness-of-cancel-lessp-times)))


(prove-lemma nat-to-bv-list-loop-clock-upper (rewrite)
             (implies
              (and
               (nat-list-piton arr ws)
               (not (zerop ws)))
              (not (lessp (times (difference (length arr) i) (plus 31 (times ws 14)))
                          (nat-to-bv-list-loop-clock i arr))))
             ((enable nat-to-bv-list-loop-clock nat-to-bv-clock clock-plus nat-list-piton
                      lessp-nat-to-bv-loop-clock-rewrite length-append commutativity-of-plus
                      commutativity2-of-plus correctness-of-cancel-lessp-plus get-as-nth
                      nth-cons nat-to-bv-list-hack get-0 untag lessp-add1-add1
                      lessp-plus-add1 plus-plus-add1 times-add1 *1*length
                      associativity-of-plus equal-times-0 length-cons equal-sub1-add1
                      equal-plus-0 plus-zero-arg2 sub1-plus-add1 plus
                      nat-to-bv-loop-clock-upper commutativity-of-times
                      times-distributes-over-difference correctness-of-cancel-lessp-times
                      correctness-of-cancel-difference-plus difference-leq-arg1)
              (induct (nat-to-bv-list-loop-clock i arr))
              (disable-theory t)
              (enable-theory ground-zero)))

(prove-lemma nat-to-bv-list-clock-upper (rewrite)
  (implies
   (and
    (nat-list-piton arr ws)
    (not (zerop ws)))
   (not (lessp (add1 (times (length arr) (plus 31 (times ws 14))))
       (nat-to-bv-list-clock arr))))
  ((enable nat-to-bv-list-clock clock-plus plus)
   (enable-theory ground-zero)
   (use (nat-to-bv-list-loop-clock-upper (i 0)))))

(prove-lemma lessp-plus-add1-plus-add1 (rewrite)
             (and
              (equal
               (lessp (plus (add1 x) y) (plus (add1 a) b))
```

```
              (lessp (plus x y) (plus a b)))
             (equal
              (lessp (plus (add1 x) y) (plus a (add1 b)))
              (lessp (plus x y) (plus a b)))
             (equal
              (lessp (plus x (add1 y)) (plus (add1 a) b))
              (lessp (plus x y) (plus a b)))
             (equal
              (lessp (plus x (add1 y)) (plus a (add1 b)))
              (lessp (plus x y) (plus a b)))))

(prove-lemma promote-add1 (rewrite)
             (and
              (equal (plus a (add1 b)) (plus (add1 a) b))
              (equal (plus a (plus (add1 b) c)) (plus (add1 a) (plus b c)))))

(disable promote-add1)

(prove-lemma bv-to-nat-loop-clock-one-vector-upper (rewrite)
             (implies
              (not (zerop ws))
              (not (lessp (plus 11 (times ws 14))
                          (bv-to-nat-loop-clock (one-bit-vector ws) bv))))
              ((use (bv-to-nat-loop-clock-upper (cb (one-bit-vector ws))))
               (disable-theory t)
               (enable-theory ground-zero naturals)
               (enable length-one-bit-vector trailing-zeros-one-bit-vector)))

(prove-lemma bv-to-nat-list-loop-clock-upper (rewrite)
             (implies
              (not (zerop ws))
              (not (lessp (times (difference (length arr) i) (plus 38 (times ws 14)))
                          (bv-to-nat-list-loop-clock ws i arr))))
             ((enable bv-to-nat-list-loop-clock bv-to-nat-clock clock-plus nat-list-piton
                      length-append correctness-of-cancel-lessp-plus get-as-nth nth-cons
                      lessp-plus-add1-plus-add1 promote-add1
                      bv-to-nat-loop-clock-one-vector-upper get-0 untag lessp-add1-add1
                      lessp-plus-add1 plus-plus-add1 times-add1 *1*length
                      associativity-of-plus equal-times-0 length-cons equal-sub1-add1
                      equal-plus-0 plus-zero-arg2 sub1-plus-add1 plus
                      bv-to-nat-loop-clock-upper commutativity-of-times
                      times-distributes-over-difference correctness-of-cancel-lessp-times
                      correctness-of-cancel-difference-plus difference-leq-arg1)
              (induct (bv-to-nat-list-loop-clock ws i arr))
              (disable-theory t)
              (enable-theory ground-zero)))

(prove-lemma bv-to-nat-list-clock-upper (rewrite)
             (implies
              (not (zerop ws))
              (not (lessp (plus 1 (times (length arr) (plus 38 (times ws 14))))
                          (bv-to-nat-list-clock ws arr))))
     ((enable bv-to-nat-list-clock clock-plus)
      (use (bv-to-nat-list-loop-clock-upper (i 0)))))

(prove-lemma max-nat-loop-clock-upper (rewrite)
             (not (lessp (times 20 (difference (length arr) i))
                         (max-nat-loop-clock val i arr)))
             ((enable clock-plus max-nat-loop-clock)
              (disable-theory t)
              (enable-theory ground-zero naturals)))
```

```
(prove-lemma max-nat-clock-upper (rewrite)
  (not (lessp (plus 2 (times (length natlist) 20))
      (max-nat-clock natlist)))
  ((enable max-nat-clock clock-plus plus)
   (use (max-nat-loop-clock-upper (i 0) (arr natlist)))))

(prove-lemma replace-value-loop-clock-upper (rewrite)
             (not (lessp (times 10 (difference (length arr) i))
                         (replace-value-loop-clock i arr val)))
             ((enable clock-plus replace-value-loop-clock)
              (disable-theory t)
              (enable-theory ground-zero naturals)))

(prove-lemma replace-value-clock-upper (rewrite)
  (not (lessp (plus 1 (times (length list) 10))
      (replace-value-clock list val)))
  ((use (replace-value-loop-clock-upper (i 0) (arr list)))
   (enable replace-value-clock clock-plus)))

;; special rule to combine big constants
(prove-lemma plus-add1-plus-add1 (rewrite)
  (equal (plus (add1 x) (plus (add1 y) z))
 (plus (plus (add1 x) (add1 y)) z)))

(disable plus-add1-plus-add1)

(prove-lemma lessp-plus-big-plus-big-hack-helper nil
  (implies
   (and
    (lessp z a)
    (lessp z b))
   (equal (lessp a b)
  (lessp (difference a z) (difference b z))))
  ((disable-theory t)
   (enable-theory ground-zero naturals)))

;; special rule to simplify lessp expression with big constants
(prove-lemma lessp-plus-big-plus-big-hack (rewrite)
  (implies
   (and
    (lessp 100 x)
    (lessp 100 y))
   (equal (lessp (plus (add1 x) a) (plus (add1 y) b))
  (lessp (plus (difference x 100) a) (plus (difference y 100) b))))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable lessp-difference lessp-difference-arg1 plus-difference-arg1
   plus-difference-arg2)
   (use (lessp-plus-big-plus-big-hack-helper
 (a (plus (add1 x) a)) (b (plus (add1 y) b)) (z 100)))))

(prove-lemma nat-list-piton-tag-array (rewrite)
             (implies
              (nat-listp state ws)
              (nat-list-piton (tag-array 'nat state) ws)))

(prove-lemma equal-replace-value-clock-0 (rewrite)
  (not (equal (replace-value-clock a l) 0))
  ((enable replace-value-clock clock-plus)))

(prove-lemma equal-number-with-at-least-clock-0 (rewrite)
  (not (equal (number-with-at-least-clock m a) 0))
```

```
  ((enable number-with-at-least-clock clock-plus)))

(prove-lemma equal-sub1-number-with-at-least-clock-0 (rewrite)
  (not (equal (sub1 (number-with-at-least-clock m a)) 0))
  ((enable number-with-at-least-clock clock-plus)))

(prove-lemma number-with-at-least-clock-min (rewrite)
  (lessp 2 (number-with-at-least-clock m a))
  ((enable number-with-at-least-clock)))

(prove-lemma max-nat-clock-min (rewrite)
  (lessp 1 (max-nat-clock val))
  ((enable max-nat-clock clock-plus)))

(prove-lemma equal-sub1-nat-to-bv-list-clock-0 (rewrite)
  (equal (equal (sub1 (nat-to-bv-list-clock arr)) 0)
 (nlistp arr))
  ((enable nat-to-bv-list-clock clock-plus)
   (expand (NAT-TO-BV-LIST-LOOP-CLOCK 0 ARR))))

(prove-lemma equal-sub1-sub1-nat-to-bv-list-clock-0 (rewrite)
  (equal (equal (sub1 (sub1 (nat-to-bv-list-clock arr))) 0)
 (nlistp arr))
  ((enable nat-to-bv-list-clock clock-plus)
   (expand (NAT-TO-BV-LIST-LOOP-CLOCK 0 ARR))))

(prove-lemma equal-nat-to-bv-list-clock-0 (rewrite)
  (not (equal (nat-to-bv-list-clock arr) 0))
  ((enable nat-to-bv-list-clock clock-plus)))

(prove-lemma lessp-sub1-nat-to-bv-list-clock (rewrite)
  (implies
   (not (lessp a (sub1 (nat-to-bv-list-clock arr))))
   (not (lessp (add1 a) (nat-to-bv-list-clock arr)))))

(prove-lemma lessp-sub1-replace-value-clock (rewrite)
  (implies
   (not (lessp a (sub1 (replace-value-clock a l))))
   (not (lessp (add1 a) (replace-value-clock a l)))))

(prove-lemma times-add1-arg1 (rewrite)
             (equal (times (add1 x) y) (plus y (times x y))))

(disable times-add1-arg1)

(prove-lemma plus-zero-arg1 (rewrite)
  (equal (plus 0 x) (fix x)))

(prove-lemma equal-sub1-replace-value-clock-0 (rewrite)
  (equal (equal (sub1 (replace-value-clock arr val)) 0)
 (nlistp arr))
  ((enable replace-value-clock replace-value-loop-clock clock-plus)
   (expand (REPLACE-VALUE-LOOP-CLOCK 0 ARR VAL))))

(prove-lemma equal-sub1-sub1-replace-value-clock-0 (rewrite)
  (equal (equal (sub1 (sub1 (replace-value-clock arr val))) 0)
 (nlistp arr))
  ((enable replace-value-clock replace-value-loop-clock clock-plus)
   (expand (REPLACE-VALUE-LOOP-CLOCK 0 ARR VAL))))

(prove-lemma not-lessp-plus-balance nil
  (implies
```

```
   (and (not (lessp a x)) (not (lessp b y)))
   (not (lessp (plus a b) (plus x y)))))

;; do it by hand using above

(prove-lemma computer-move-helper1 nil
 (implies (and (nat-listp state ws)
               (not (zerop ws))
               (lessp 1 (length state))
       (equal (number-with-at-least state 2) 0))
          (not (lessp (plus 10078
                            (times 150 (length state))
                            (times 12 ws)
                            (times 24 (sub1 (length state)))
                            (times 42 ws (length state)))
                      (computer-move-clock state ws))))
 ((disable-theory t)
  (use (max-nat-clock-upper (natlist (TAG-ARRAY 'NAT STATE)))
       (nat-to-bv-list-clock-upper (arr (TAG-ARRAY 'NAT STATE)))
       (number-with-at-least-clock-upper (min 2) (array (tag-array 'nat state)))
       (number-with-at-least-clock-upper (min 1) (array (tag-array 'nat state)))
       (replace-value-clock-upper (list (tag-array 'nat state))
  (val (LIST 'NAT (MAX-LIST STATE)))))

  (enable-theory ground-zero)
  (enable computer-move-clock smart-move-clock clock-plus delay-clock-upper
  equal-length-0 commutativity-of-plus commutativity2-of-plus
  listp-tag-array  max-nat-clock-min
  nat-list-piton-tag-array equal-replace-value-clock-0
  plus-add1-arg2 sub1-plus-add1 equal-plus-0 equal-sub1-replace-value-clock-0
  equal-sub1-sub1-replace-value-clock-0
  correctness-of-cancel-lessp-plus equal-number-with-at-least-clock-0
  number-with-at-least-clock-min
  lessp-sub1-nat-to-bv-list-clock plus-zero-arg1
  lessp-sub1-replace-value-clock times-add1-arg1
  lessp-plus-big-plus-big-hack lessp-plus-add1-plus-add1
  associativity-of-plus plus-add1-plus-add1
  commutativity-of-times length-tag-array equal-nat-to-bv-list-clock-0
  xor-bvs-clock-upper)))

(prove-lemma computer-move-helper2 nil
 (implies (and (nat-listp state ws)
               (not (zerop ws))
               (lessp 1 (length state))
       (not (equal (number-with-at-least state 2) 0))
       (all-zero-bitvp (xor-bvs (nat-to-bv-list state ws))))
          (not (lessp (plus 10078
                            (times 150 (length state))
                            (times 12 ws)
                            (times 24 (sub1 (length state)))
                            (times 42 ws (length state)))
                      (computer-move-clock state ws))))
 ((disable-theory t)
  (use (max-nat-clock-upper (natlist (TAG-ARRAY 'NAT STATE)))
       (nat-to-bv-list-clock-upper (arr (TAG-ARRAY 'NAT STATE)))
       (number-with-at-least-clock-upper (min 2) (array (tag-array 'nat state)))
       (number-with-at-least-clock-upper (min 1) (array (tag-array 'nat state)))
       (replace-value-clock-upper (list (tag-array 'nat state))
  (val (LIST 'NAT (MAX-LIST STATE)))))

  (enable-theory ground-zero)
  (enable computer-move-clock smart-move-clock clock-plus delay-clock-upper
```

```
  equal-length-0 commutativity-of-plus commutativity2-of-plus
  listp-tag-array  max-nat-clock-min
  nat-list-piton-tag-array equal-replace-value-clock-0
  plus-add1-arg2 sub1-plus-add1 equal-plus-0 equal-sub1-replace-value-clock-0
  equal-sub1-sub1-replace-value-clock-0
  correctness-of-cancel-lessp-plus equal-number-with-at-least-clock-0
  number-with-at-least-clock-min times-distributes-over-plus
  lessp-sub1-nat-to-bv-list-clock plus-zero-arg1
  lessp-sub1-replace-value-clock times-add1-arg1
  lessp-plus-big-plus-big-hack lessp-plus-add1-plus-add1
  associativity-of-plus plus-add1-plus-add1
  commutativity-of-times length-tag-array equal-nat-to-bv-list-clock-0
  xor-bvs-clock-upper)))

(prove-lemma lessp-nat-to-bv-list-clock (rewrite)
  (implies
   (listp nats)
   (lessp 11 (nat-to-bv-list-clock nats)))
  ((enable nat-to-bv-list-clock nat-to-bv-list-loop-clock clock-plus)
   (expand (NAT-TO-BV-LIST-LOOP-CLOCK 0 NATS))))

(prove-lemma computer-move-helper3 nil
 (implies (and (nat-listp state ws)
               (not (zerop ws))
               (lessp 1 (length state))
       (equal (number-with-at-least state 2) 1)
       (not (all-zero-bitvp (xor-bvs (nat-to-bv-list state ws)))))
          (not (lessp (plus 10078
                            (times 150 (length state))
                            (times 12 ws)
                            (times 24 (sub1 (length state)))
                            (times 42 ws (length state)))
                      (computer-move-clock state ws))))
 ((disable-theory t)
  (use (max-nat-clock-upper (natlist (TAG-ARRAY 'NAT STATE)))
       (nat-to-bv-list-clock-upper (arr (TAG-ARRAY 'NAT STATE)))
       (number-with-at-least-clock-upper (min 2) (array (tag-array 'nat state)))
       (number-with-at-least-clock-upper (min 1) (array (tag-array 'nat state)))
       (replace-value-clock-upper (list (tag-array 'nat state))
  (val (LIST 'NAT (MAX-LIST STATE)))))
  (enable-theory ground-zero)
  (enable computer-move-clock smart-move-clock clock-plus delay-clock-upper
  equal-length-0 commutativity-of-plus commutativity2-of-plus
  listp-tag-array  max-nat-clock-min lessp-nat-to-bv-list-clock
  nat-list-piton-tag-array equal-replace-value-clock-0
  plus-add1-arg2 sub1-plus-add1 equal-plus-0 equal-sub1-replace-value-clock-0
  equal-sub1-sub1-replace-value-clock-0
  correctness-of-cancel-lessp-plus equal-number-with-at-least-clock-0
  number-with-at-least-clock-min times-distributes-over-plus
  lessp-sub1-nat-to-bv-list-clock plus-zero-arg1
  lessp-sub1-replace-value-clock times-add1-arg1
  lessp-plus-big-plus-big-hack lessp-plus-add1-plus-add1
  associativity-of-plus plus-add1-plus-add1
  commutativity-of-times length-tag-array equal-nat-to-bv-list-clock-0
  xor-bvs-clock-upper)))


(prove-lemma lessp-bv-to-nat-list-clock (rewrite)
  (implies
   (listp nats)
   (lessp 11 (bv-to-nat-list-clock ws nats)))
  ((enable bv-to-nat-list-clock bv-to-nat-list-loop-clock clock-plus)
```

```
   (expand (BV-TO-NAT-LIST-LOOP-CLOCK WS 0 NATS))))

(prove-lemma listp-match-and-xor (rewrite)
  (equal (listp (match-and-xor bvs x y)) (listp bvs)))

(prove-lemma lessp-match-and-xor-clock (rewrite)
  (lessp 2 (match-and-xor-clock bvs match))
  ((enable match-and-xor-clock)))


(prove-lemma computer-move-helper4 nil
 (implies (and (nat-listp state ws)
               (not (zerop ws))
               (lessp 1 (length state))
       (not (lessp (number-with-at-least state 2) 2))
       (not (all-zero-bitvp (xor-bvs (nat-to-bv-list state ws)))))
          (not (lessp (plus 10400
                            (times 150 (length state))
                            (times 12 ws)
                            (times 38 (sub1 (length state)))
                            (times 42 ws (length state)))
                      (computer-move-clock state ws))))
 ((disable-theory t)
  (use (nat-to-bv-list-clock-upper (arr (TAG-ARRAY 'NAT STATE)))
       (number-with-at-least-clock-upper (min 2) (array (tag-array 'nat state)))
       (highest-bit-clock-upper (bv (xor-bvs (nat-to-bv-list state ws))))
       (match-and-xor-clock-upper
(bvs (nat-to-bv-list state ws))
(match (highest-bit (xor-bvs (nat-to-bv-list state ws)))))
       (bv-to-nat-list-clock-upper (ws ws)
   (arr (tag-array
 'bitv
 (match-and-xor
  (nat-to-bv-list state ws)
  (highest-bit
   (xor-bvs (nat-to-bv-list state ws)))
  (xor-bvs (nat-to-bv-list state ws)))))))
  (enable-theory ground-zero)
  (enable computer-move-clock smart-move-clock clock-plus delay-clock-upper
  equal-length-0 commutativity-of-plus commutativity2-of-plus
  equal-sub1-nat-to-bv-list-clock-0 equal-sub1-sub1-nat-to-bv-list-clock-0
  length-tag-array equal-sub1-number-with-at-least-clock-0
  listp-match-and-xor length-xor-bvs bit-vectorsp-nat-to-bv-list-better
  listp-tag-array  max-nat-clock-min  length-match-and-xor
  length-nat-to-bv-list listp-nat-to-bv lessp-bv-to-nat-list-clock
  car-nat-to-bv-list listp-xor-bvs lessp-match-and-xor-clock
  nat-list-piton-tag-array equal-replace-value-clock-0
  length-nat-to-bv
  plus-add1-arg2 sub1-plus-add1 equal-plus-0 equal-sub1-replace-value-clock-0
  equal-sub1-sub1-replace-value-clock-0 listp-nat-to-bv-list
  correctness-of-cancel-lessp-plus equal-number-with-at-least-clock-0
  number-with-at-least-clock-min times-distributes-over-plus
  lessp-sub1-nat-to-bv-list-clock plus-zero-arg1
  lessp-sub1-replace-value-clock times-add1-arg1
  lessp-plus-big-plus-big-hack lessp-plus-add1-plus-add1
  associativity-of-plus plus-add1-plus-add1
  commutativity-of-times length-tag-array equal-nat-to-bv-list-clock-0
  xor-bvs-clock-upper)))

(prove-lemma computer-move-helper nil
 (implies (and (nat-listp state ws)
               (not (zerop ws))
```

```
               (lessp 1 (length state)))
          (not (lessp (plus 10400
                            (times 150 (length state))
                            (times 12 ws)
                            (times 38 (sub1 (length state)))
                            (times 42 ws (length state)))
                      (computer-move-clock state ws))))
 ((disable-theory t)
  (enable-theory ground-zero)
  (use (computer-move-helper1) (computer-move-helper2)
       (computer-move-helper3) (computer-move-helper4))))

(prove-lemma computer-move-helper-too (rewrite)
             (implies
              (and (nat-listp state ws)
                   (not (zerop ws))
                   (lessp 1 (length state)))
              (lessp 10000 (computer-move-clock state ws)))
             ((disable-theory t)
              (enable-theory ground-zero)
              (enable delay-clock-upper computer-move-clock clock-plus)))

(prove-lemma lessp-times nil
  (implies
   (and
    (not (lessp a b))
    (not (lessp c d)))
   (not (lessp (times a c) (times b d)))))


(prove-lemma computer-move-time nil
             (implies (and (nat-listp state ws)
                           (lessp 0 ws)
                           (not (lessp 32 ws))
                           (lessp 1 (length state))
                           (not (lessp 6 (length state))))
                      (and (lessp 10000
                                  (computer-move-clock state ws))
                           (lessp (computer-move-clock state ws)
                                  20000)))
     ((use (computer-move-helper) (computer-move-helper-too)
   (lessp-times (a 32) (b ws) (c 6) (d (length state))))
      (enable-theory ground-zero)
      (enable times-add1-arg1 times-distributes-over-plus
      commutativity-of-plus  associativity-of-plus
      commutativity2-of-plus)
      (disable-theory t)))

))
```

Appendix E

Verified Real-time System Examples

This appendix contains the events that lead Nqthm to the correctness proofs of the light-switch and quiz-show examples. The events are divided into five files, each presented in a separate section. The initial events list depends upon the definitions and lemmas of the FM9001 verification proof, and each of the other events lists depends on its predecessor.

light-switch.events contains the proof of the light-switch system, including the formalization of behaviors, the real-time FM9001 model, and many supporting lemmas.

quiz-abstract.events contains the proof of the "abstract" lemma about the quiz-show system.

quiz-prog.events contains the proof of the "reasonableness" lemma about the quiz-show system.

quiz-correct.events contains the proof of the "program correctness" lemma about the quiz-show system.

quiz-final.events contains the proof of the final quiz-show correctness theorem.

E.1 "light-switch.events"

```
(proveall "light-switch"
'(
#|
 Copyright (C) 1995 by Matthew Wilding and Computational Logic, Inc.
 All Rights Reserved.

 This script is hereby placed in the public domain, and therefore unlimited
 editing and redistribution is permitted.

 NO WARRANTY

 Matthew Wilding and Computational Logic, Inc. PROVIDES ABSOLUTELY NO
 WARRANTY.  THE EVENT SCRIPT IS PROVIDED "AS IS" WITHOUT WARRANTY OF
 ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
 ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE.  THE ENTIRE RISK AS TO THE QUALITY AND
 PERFORMANCE OF THE SCRIPT IS WITH YOU.  SHOULD THE SCRIPT PROVE
 DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR
 CORRECTION.

 IN NO EVENT WILL Matthew Wilding and Computational Logic, Inc. BE
 LIABLE TO YOU FOR ANY DAMAGES, ANY LOST PROFITS, LOST MONIES, OR
 OTHER SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE
 USE OR INABILITY TO USE THIS SCRIPT (INCLUDING BUT NOT LIMITED TO
 LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY
 THIRD PARTIES), EVEN IF YOU HAVE ADVISED US OF THE POSSIBILITY OF
 SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

This file contains the events that lead to the proof of
microprocessor-controlled light-switch.

|#


;; We use the definitions and lemmas of the FM9001 verification project
;; that is part of the Computational Logic, Inc. Nqthm-1992 release.
(note-lib "/slocal/src/nqthm-1992/examples/fm9001-piton/fm9001-replay" t)

;; An expression evaluator
;;
;; terms of the form (old litatom) are evaluated using old-alist

(defn my-eval (term alist old-alist)
  (if (listp term)

  (if (equal (car term) 'equal)
      (equal
       (my-eval (cadr term) alist old-alist)
       (my-eval (caddr term) alist old-alist))

  (if (equal (car term) 'lessp)
      (lessp
       (my-eval (cadr term) alist old-alist)
       (my-eval (caddr term) alist old-alist))

  (if (equal (car term) 'not)
      (not (my-eval (cadr term) alist old-alist))

  (if (equal (car term) 'plus)
      (plus
```

```
       (my-eval (cadr term) alist old-alist)
       (my-eval (caddr term) alist old-alist))

  (if (equal (car term) 'remainder)
      (remainder
       (my-eval (cadr term) alist old-alist)
       (my-eval (caddr term) alist old-alist))

  (if (equal (car term) 'difference)
      (difference
       (my-eval (cadr term) alist old-alist)
       (my-eval (caddr term) alist old-alist))

  (if (equal (car term) 'if)
      (if
        (my-eval (cadr term) alist old-alist)
          (my-eval (caddr term) alist old-alist)
        (my-eval (cadddr term) alist old-alist))

  (if (equal (car term) 'and)
      (and
       (my-eval (cadr term) alist old-alist)
       (my-eval (caddr term) alist old-alist))

  (if (equal (car term) 'or)
      (or
       (my-eval (cadr term) alist old-alist)
       (my-eval (caddr term) alist old-alist))

  (if (equal (car term) 'old)
      (cadr (assoc (cadr term) old-alist))

  (if (equal (car term) 'c)
      (cadr (assoc term alist))

  (if (equal (car term) 'quote)
      (cadr term)

  (if (equal (car term) 'true)
      t

  (if (equal (car term) 'false)
      f

    0))))))))))))))

  (if (numberp term) term

    (cadr (assoc term alist)))))


(defn state-name (state) (car state))
(defn state-pred (state) (cadr state))
(defn state-trans-list (state) (caddr state))

(defn trans-to (trans) (car trans))
(defn trans-resets (trans) (cadr trans))
(defn trans-pred (trans) (caddr trans))

(defn behavior-initial (behavior) (car behavior))
(defn behavior-state-list (behavior) (cadr behavior))
```

```
(defn update-counter-alist (counter-alist resets)
  (if (listp counter-alist)
      (if (member (caar counter-alist) resets)
          (cons (list (caar counter-alist) 0)
                (update-counter-alist (cdr counter-alist) resets))
        (cons (list (caar counter-alist) (add1 (cadar counter-alist)))
              (update-counter-alist (cdr counter-alist) resets)))
    nil))

(defn make-list-alist (list)
  (if (listp list)
      (cons (list (car list) 0) (make-list-alist (cdr list)))
    nil))

(defn set-union (s1 s2)
  (if (listp s1)
      (if (member (car s1) s2)
          (set-union (cdr s1) s2)
        (cons (car s1) (set-union (cdr s1) s2)))
    s2))

(defn counters-of-term (term)
  (if (listp term)
  (if (or (equal (car term) 'equal)
          (equal (car term) 'lessp)
          (equal (car term) 'plus)
          (equal (car term) 'and)
          (equal (car term) 'or)
          (equal (car term) 'remainder)
          (equal (car term) 'difference))
      (set-union (counters-of-term (cadr term))
                 (counters-of-term (caddr term)))
    (if (equal (car term) 'if)
        (set-union (counters-of-term (cadr term))
                   (set-union
                    (counters-of-term (caddr term))
                    (counters-of-term (cadddr term))))
      (if (or
           (equal (car term) 'not)
           (equal (car term) 'old))
        (counters-of-term (cadr term))
      (if (equal (car term) 'c)
          (list term)
        nil))))
    nil))

(defn counters-of-trans (trans)
  (set-union (trans-resets trans) (counters-of-term (trans-pred trans))))

(defn counters-of-translist (translist)
  (if (listp translist)
      (set-union (counters-of-trans (car translist))
                 (counters-of-translist (cdr translist)))
    nil))

(defn counters-of-state (state)
  (set-union (counters-of-term (state-pred state))
             (counters-of-translist (state-trans-list state))))

(defn counters-of-state-list (state-list)
```

```
  (if (listp state-list)
      (set-union (counters-of-state (car state-list))
                 (counters-of-state-list (cdr state-list)))
  nil))


#|
There are three different versions of the notion of satisfiesp.

satisfiesp is the "official" version

satisfiesp-eff is equivalent to satisfiesp, but uses if rather than
and and or to aid in efficient execution.

satisfiesp-with-path takes a third argument that represents the path
of states that demonstrates the satisfaction.  satisfiesp-with-path
implies satisfiesp

|#

(defn satisfiesp-helper (name pred trans h counters b-states)
  (if (listp h)
      (if (listp (cdr h))
          (if (listp trans)
              (and
               (assoc name b-states)
               (my-eval pred (append counters (car h)) nil)
               (let
                   ((new-state (assoc (trans-to (car trans)) b-states))
                    (new-counters (update-counter-alist
                                   counters (trans-resets (car trans)))))
                 (or
                  (and
                   (my-eval
                    (trans-pred (car trans))
                    (append new-counters (cadr h))
                    (append counters (car h)))
                   (satisfiesp-helper
                    (state-name new-state) (state-pred new-state)
                    (state-trans-list new-state) (cdr h)
                    new-counters b-states))
               (satisfiesp-helper name pred (cdr trans) h counters b-states))))
            f)
        (and
         (assoc name b-states)
         (my-eval pred (append counters (car h)) nil)))
    t)
  ((ord-lessp (cons (add1 (length h)) (length trans)))))


(defn satisfiesp (h b)
  (let ((init-state (assoc (behavior-initial b) (behavior-state-list b))))
    (satisfiesp-helper
     (state-name init-state)
     (state-pred init-state)
     (state-trans-list init-state)
     h
     (make-list-alist (counters-of-state-list (behavior-state-list b)))
     (behavior-state-list b))))

(defn satisfiesp-eff-helper (name pred trans h counters b-states)
  (if (listp h)
```

```
      (if (listp (cdr h))
          (if (listp trans)
              (if
                  (assoc name b-states)
                  (if
                      (my-eval pred (append counters (car h)) nil)
                      (let
                          ((new-state (assoc (trans-to (car trans)) b-states))
                           (new-counters (update-counter-alist
                                          counters (trans-resets (car trans)))))
                        (if
                            (if
                                (my-eval
                                 (trans-pred (car trans))
                                 (append new-counters (cadr h))
                                 (append counters (car h)))
                                (satisfiesp-eff-helper
                                 (state-name new-state)
                                 (state-pred new-state) (state-trans-list new-state)
                                 (cdr h) new-counters b-states)
                              f)
                            t
                          (satisfiesp-eff-helper name pred (cdr trans) h counters b-states)))
                    f)
                f)
            f)
        (if (assoc name b-states)
            (if (my-eval pred (append counters (car h)) nil) t f)
          f))
    t)
  ((ord-lessp (cons (add1 (length h)) (length trans)))))


(defn satisfiesp-eff (h b)
  (let ((init-state (assoc (behavior-initial b) (behavior-state-list b))))
    (satisfiesp-eff-helper
     (state-name init-state)
     (state-pred init-state)
     (state-trans-list init-state)
     h
     (make-list-alist (counters-of-state-list (behavior-state-list b)))
     (behavior-state-list b))))

(prove-lemma satisfiesp-eff-helper-equiv (rewrite)
  (equal
   (satisfiesp-helper n p tr h c bs)
   (satisfiesp-eff-helper n p tr h c bs))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable satisfiesp-eff-helper satisfiesp-helper)))

(prove-lemma satisfiesp-eff-equiv (rewrite)
  (equal (satisfiesp h b) (satisfiesp-eff h b))
  ((disable satisfiesp-eff-helper satisfiesp-helper)))

(defn satisfiesp-with-path-helper (pred trans h counters b-states path)
  (if (listp h)
      (if (listp (cdr h))
          (if (listp trans)
              (and
               (assoc (car path) b-states)
               (my-eval pred (append counters (car h)) nil)
```

```
              (let
                  ((new-state (assoc (trans-to (car trans)) b-states))
                   (new-counters (update-counter-alist
                                  counters (trans-resets (car trans)))))
                (if (equal (trans-to (car trans)) (cadr path))
                    (and
                     (my-eval
                      (trans-pred (car trans))
                      (append new-counters (cadr h))
                      (append counters (car h)))
                     (satisfiesp-with-path-helper
                      (state-pred new-state) (state-trans-list new-state)
                      (cdr h) new-counters b-states (cdr path)))
                  (satisfiesp-with-path-helper pred (cdr trans) h counters
                                               b-states path))))
           f)
       (and
        (assoc (car path) b-states)
        (my-eval pred (append counters (car h)) nil)))
   t)
  ((ord-lessp (cons (add1 (length h)) (length trans)))))

(defn satisfiesp-with-path (h b path)
  (and
   (implies (listp h) (equal (behavior-initial b) (car path)))
  (let ((init-state (assoc (behavior-initial b) (behavior-state-list b))))
    (satisfiesp-with-path-helper
     (state-pred init-state)
     (state-trans-list init-state)
     h
     (make-list-alist (counters-of-state-list (behavior-state-list b)))
     (behavior-state-list b)
    path))))

(prove-lemma car-assoc (rewrite)
  (equal (car (assoc k l)) (if (assoc k l) k 0)))

;; satisfiesp-with-path implies satisfiesp.  Also, satisfiesp implies
;; satisfiesp-with-path for a particular path calculated with
;; function find-path.

(prove-lemma satisfiesp-with-path-helper-means-satisfiesp-helper (rewrite)
  (implies
   (satisfiesp-with-path-helper p tr h c b path)
   (satisfiesp-helper (car path) p tr h c b))
  ((disable-theory t)
   (enable-theory ground-zero)
   (expand (satisfiesp-eff-helper (car path) p tr h c b))
   (enable state-name state-pred state-trans-list trans-to trans-resets trans-pred
           satisfiesp-helper satisfiesp-eff-helper satisfiesp-eff-helper-equiv
           satisfiesp-with-path-helper car-assoc)))

(disable satisfiesp-eff-equiv)
(disable satisfiesp-eff-helper-equiv)

(prove-lemma satisfiesp-with-path-means-satisfiesp (rewrite)
  (implies (satisfiesp-with-path h b path) (satisfiesp h b)))

(defn find-path-helper (name pred trans h counters b-states)
  (if (listp h)
      (if (listp (cdr h))
          (if (listp trans)
```

```
            (let
                ((new-state (assoc (trans-to (car trans)) b-states))
                 (new-counters (update-counter-alist
                                counters (trans-resets (car trans)))))
              (if
                  (and
                   (my-eval
                    (trans-pred (car trans))
                    (append new-counters (cadr h))
                    (append counters (car h)))
                   (satisfiesp-helper
                    (state-name new-state)
                    (state-pred new-state) (state-trans-list new-state)
                    (cdr h) new-counters b-states))
                  (cons name
                        (find-path-helper
                         (state-name new-state)
                         (state-pred new-state) (state-trans-list new-state)
                         (cdr h) new-counters b-states))
                (find-path-helper name pred (cdr trans) h counters b-states)))
          nil)
      (list name))
    nil)
  ((ord-lessp (cons (add1 (length h)) (length trans)))))

(defn find-path (h b)
  (let ((init-state (assoc (behavior-initial b) (behavior-state-list b))))
    (find-path-helper
     (state-name init-state)
     (state-pred init-state)
     (state-trans-list init-state)
     h
     (make-list-alist (counters-of-state-list (behavior-state-list b)))
     (behavior-state-list b))))

(defn member-cars (x l)
      (if (listp l)
          (if (equal x (caar l))
              t
              (member-cars x (cdr l)))
          f))

(defn cars-uniquep (list)
  (if (listp list)
      (and
       (not (member-cars (caar list) (cdr list)))
       (cars-uniquep (cdr list)))
    t))

(defn all-transitions-uniquep (state-list)
  (if (listp state-list)
      (and
       (cars-uniquep (state-trans-list (car state-list)))
       (all-transitions-uniquep (cdr state-list)))
    t))

(prove-lemma cars-uniquep-assoc (rewrite)
  (implies
   (all-transitions-uniquep b)
   (cars-uniquep (state-trans-list (assoc x b)))))

(defn all-cars-litatoms (l)
```

```
  (if (listp l)
      (and
       (litatom (caar l))
       (all-cars-litatoms (cdr l)))
    t))

(defn all-transitions-litatoms (state-list)
  (if (listp state-list)
      (and
       (all-cars-litatoms (state-trans-list (car state-list)))
       (all-transitions-litatoms (cdr state-list)))
    t))

(prove-lemma all-cars-litatoms-assoc (rewrite)
  (implies
   (all-transitions-litatoms b)
   (all-cars-litatoms (state-trans-list (assoc x b)))))

(prove-lemma assoc-of-all-cars-litatoms (rewrite)
  (implies
   (and
    (not (litatom x))
    (all-cars-litatoms l))
  (equal (assoc x l) f)))

(prove-lemma car-find-path-helper (rewrite)
  (implies
   (and
    (all-cars-litatoms x)
    (all-transitions-litatoms b)
    (satisfiesp-helper n p x h c b))
   (equal
    (car (find-path-helper n p x h c b))
    (if (listp h) n 0)))
  ((disable state-trans-list satisfiesp-eff-helper-equiv)))

(prove-lemma member-cars-find-path (rewrite)
  (implies
   (and
    (satisfiesp-helper n p tr h c b)
    (all-cars-litatoms tr)
    (all-cars-litatoms b)
    (all-transitions-litatoms b))
   (equal
    (member-cars (cadr (find-path-helper n p tr h c b)) tr)
    (and (listp h) (listp (cdr h)))))
  ((enable satisfiesp-eff-helper-equiv)
   (disable state-trans-list)))

(prove-lemma satisfiesp-helper-means-with-find-path (rewrite)
  (implies
   (and
    (satisfiesp-eff-helper n p tr h c b)
    (all-transitions-uniquep b)
    (all-transitions-litatoms b)
    (all-cars-litatoms b)
    (all-cars-litatoms tr)
    (cars-uniquep tr))
   (satisfiesp-with-path-helper p tr h c b (find-path-helper n p tr h c b)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable satisfiesp-eff-helper satisfiesp-with-path-helper find-path-helper
```

```
          satisfiesp-eff-helper-equiv cars-uniquep-assoc trans-to cars-uniquep member
          cars-uniquep assoc-of-all-cars-litatoms all-cars-litatoms-assoc
          *1*state-trans-list member-cars-find-path car-find-path-helper all-cars-litatoms
          state-name car-assoc)
   (induct (find-path-helper n p tr h c b))))

(prove-lemma satisfiesp-means-with-find-path (rewrite)
  (implies
   (and
    (satisfiesp h b)
    (all-transitions-uniquep (cadr b))
    (all-cars-litatoms (cadr b))
    (all-transitions-litatoms (cadr b)))
   (satisfiesp-with-path h b (find-path h b)))
  ((enable satisfiesp-eff-helper-equiv)
   (disable state-trans-list)))

#|
(setq simple '(s1 ((s1 (equal x 'on) ((s1 nil (equal y (old y))) (s2 ((c 1)) nil)))
                   (s2 (equal x 'off) ((s1 nil nil) (s2 nil (lessp (c 1) 3)))))))

(satisfiesp '(((x on) (y 2))
              ((x on) (y 2))
              ((x on) (y 2))
              ((x off) (y 1))
              ((x off) (y 2))
              ((x off) (y 1))
              ((x off) (y 2))
              ((x off) (y 2)))
            simple)

|#
;;;;
;;;;
;;;;

;;;;The mirrored button problem.

;;;;
;;;;
;;;;

(defn button-spec ()
  '(up
    ((up
      (and (equal signal 0) (equal action 'f))
      ((up nil (true))
       (going-down ((c 1))
        (and (equal signal 1)
             (and (equal action 'depress) (lessp 40000 (c 2)))))))
     (going-down
      (lessp (c 1) 30000)
      ((going-down nil (equal action 'f)) (down nil (true))))
     (down
      (and (equal signal 1) (equal action 'f))
      ((down nil (true))
       (going-up ((c 2))
        (and (equal signal 0)
             (and (equal action 'release) (lessp 40000 (c 1)))))))
     (going-up
      (lessp (c 2) 30000)
      ((going-up nil (equal action 'f)) (up nil (true))))))
```

```
(defn mirror-program-spec ()
  '(a0
    ((a0
      (lessp (c 7) 1000)
      ((a0 nil
           (and
            (equal newsig (old newsig))
            (equal output (old output))))
       (a1 ((c 8))
           (and (equal output (old output))
                (equal newsig (old signal))))))
     (a1
      (lessp (c 8) 1000)
      ((a0 ((c 7))
           (and
            (equal output (old newsig))
            (equal newsig (old newsig))))
       (a1 nil (and (equal output (old output))
                    (equal newsig (old newsig)))))))))

(defn mirror-system-spec ()
  '(on
    ((on
      (and (equal output 0) (equal action 'f))
      ((on nil (true)) (going-off ((c 3)) (equal action 'depress))))
     (going-off
      (lessp (c 3) 40000)
      ((going-off nil (equal action 'f)) (off nil (true))))
     (off
      (and (equal output 1) (equal action 'f))
      ((off nil (true))
       (going-on ((c 4)) (equal action 'release))))
     (going-on
      (lessp (c 4) 40000)
      ((going-on nil (equal action 'f)) (on nil (true)))))))


(defn n31000 () 31000)

(prove-lemma n31000-props (rewrite)
  (and
   (not (lessp (n31000) 31000))
   (not (lessp 31000 (n31000)))))

(disable n31000)
(disable *1*n31000)

(defn n32000 () 32000)

(prove-lemma n32000-props (rewrite)
  (and
   (not (lessp (n32000) 32000))
   (not (lessp 32000 (n32000)))))

(disable n32000)
(disable *1*n32000)

(defn n30000 () 30000)

(prove-lemma n30000-props (rewrite)
```

```
  (and
   (not (lessp (n30000) 30000))
   (not (lessp 30000 (n30000)))))

(disable n30000)
(disable *1*n30000)


(defn mirror-system-invariant
  (c1 c2 c3 c4 c7 c8 button-state program-state signal output newsig)
  (and
   (equal c1 c3)
   (equal c2 c4)
   (implies
    (equal button-state 'down)
    (and
     (equal signal 1)
     (implies
      (equal program-state 'a0)
      (and
       (implies (lessp (plus c7 (n31000)) c1) (equal newsig 1))
       (implies (lessp (plus c7 (n31000)) c1) (equal output 1))))
     (implies
      (equal program-state 'a1)
      (and
       (implies (lessp (plus (n30000) c8) c1) (equal newsig 1))
       (implies (lessp (plus (n32000) c8) c1) (equal output 1))))))
   (implies
    (equal button-state 'up)
    (and
     (equal signal 0)
     (implies
      (equal program-state 'a0)
      (and
       (implies (lessp (plus c7 (n31000)) c2) (equal newsig 0))
       (implies (lessp (plus c7 (n31000)) c2) (equal output 0))))
     (implies
      (equal program-state 'a1)
      (and
       (implies (lessp (plus (n30000) c8) c2) (equal newsig 0))
       (implies (lessp (plus (n32000) c8) c2) (equal output 0))))))))


;; produce a list describing path through mirror-system-spec
(defn mirror-map (h path1 path2)
  (if (listp h)
      (cons
       (cond
        ((equal (car path1) 'going-down) 'going-off)
        ((equal (car path1) 'going-up) 'going-on)
        ((equal (car path1) 'up)
         (if (equal (car path2) 'a1)
             (if (and
                  (equal (cadr (assoc 'newsig (car h))) 0)
                  (equal (cadr (assoc 'output (car h))) 0))
                 'on
               'going-on)
           (if (equal (cadr (assoc 'output (car h))) 0)
               'on
             'going-on)))
        (t    ;; 'down
         (if (equal (car path2) 'a1)
```

```
            (if (and
                 (equal (cadr (assoc 'newsig (car h))) 1)
                 (equal (cadr (assoc 'output (car h))) 1))
                'off
              'going-off)
          (if (equal (cadr (assoc 'output (car h))) 1)
              'off
            'going-off))))
      (mirror-map (cdr h) (cdr path1) (cdr path2)))
    nil))


(defn satisfiesp-with-path-assoc-helper (pred trans h counters b-states path)
  (if (listp h)
      (if (listp (cdr h))
          (and
           (assoc (car path) b-states)
           (my-eval pred (append counters (car h)) nil)
           (let
               ((pick-trans (assoc (cadr path) trans)))
             (let
                 ((new-state (assoc (trans-to pick-trans) b-states))
                  (new-counters (update-counter-alist
                                 counters (trans-resets pick-trans))))
               (and
                pick-trans
                (my-eval
                 (trans-pred pick-trans)
                 (append new-counters (cadr h))
                 (append counters (car h)))
                (satisfiesp-with-path-assoc-helper
                 (state-pred new-state) (state-trans-list new-state)
                 (cdr h) new-counters b-states (cdr path))))))
        (and
         (assoc (car path) b-states)
         (my-eval pred (append counters (car h)) nil)))
    t))

(defn s-w-p-a-induct (pred trans1 trans2 h counters b path)
  (if (listp h)
      (if (listp (cdr h))
          (if (listp trans2)
          (let
              ((new-state (assoc (trans-to (car trans2)) b))
               (new-counters (update-counter-alist
                              counters (trans-resets (car trans2)))))
            (if (equal (trans-to (car trans2)) (cadr path))
                (s-w-p-a-induct
                 (state-pred new-state)
                 (state-trans-list new-state)
                 (state-trans-list new-state)
                 (cdr h)
                 new-counters
                 b
                 (cdr path))
              (s-w-p-a-induct pred trans1 (cdr trans2) h counters b path)))
          t)
        t)
    t)
  ((ord-lessp (cons (add1 (length h)) (length trans2)))))

;; same as before, but useful if state-trans-list is enabled.
```

```
(prove-lemma all-cars-litatoms-assoc-trans-list-open (rewrite)
  (implies
   (all-transitions-litatoms b)
   (all-cars-litatoms (caddr (assoc x b)))))

(prove-lemma satisfiesp-with-path-assoc-same-helper nil
     (implies (and (equal (assoc (car (cdr path)) trans1)
                          (assoc (car (cdr path)) trans2))
                   (and (all-cars-litatoms trans2)
                        (all-transitions-litatoms b)))
              (equal (satisfiesp-with-path-assoc-helper pred trans1 h
                         counters b path)
                     (satisfiesp-with-path-helper pred trans2 h
                         counters b path)))
     ((induct (s-w-p-a-induct pred trans1 trans2 h counters b path))
      (expand (satisfiesp-with-path-helper pred trans2 h counters b
                  path)
              (satisfiesp-with-path-assoc-helper pred trans1 h counters
                  b path))))

(prove-lemma satisfiesp-with-path-assoc-same (rewrite)
     (implies (and (all-cars-litatoms trans)
                   (all-transitions-litatoms b))
              (equal (satisfiesp-with-path-assoc-helper pred trans h counters b path)
                     (satisfiesp-with-path-helper pred trans h counters b path)))
     ((use (satisfiesp-with-path-assoc-same-helper (trans1 trans) (trans2 trans)))))

(disable SATISFIESP-WITH-PATH-ASSOC-SAME)

(prove-lemma assoc-update-counter-alist (rewrite)
  (implies
   (not (equal k 0))
  (equal
   (cadr (assoc k (update-counter-alist counters list)))
   (if (or (not (assoc k counters)) (member k list))
       0
     (add1 (cadr (assoc k counters)))))))

(prove-lemma assoc-update-counter-alist-iff (rewrite)
  (implies
   (not (equal k 0))
  (iff
   (assoc k (update-counter-alist c l))
   (assoc k c))))

(defn sth-induct (C1 C2 C3 P1 PATH1 TR1 P2 PATH2 TR2 H P3 TR3)
  (if (listp h)
      (if (listp (cdr h))
          (sth-induct
          (UPDATE-COUNTER-ALIST C1
                                (TRANS-RESETS (ASSOC (CADR PATH1) TR1)))
          (UPDATE-COUNTER-ALIST C2
                                (TRANS-RESETS (ASSOC (CADR PATH2) TR2)))
          (UPDATE-COUNTER-ALIST C3
                         (TRANS-RESETS (ASSOC (CADR (MIRROR-MAP H PATH1 PATH2))
                                              TR3)))
          (STATE-PRED (ASSOC (TRANS-TO (ASSOC (CADR PATH1) TR1))
                             (CADR (BUTTON-SPEC))))
          (CDR PATH1)
          (STATE-TRANS-LIST (ASSOC (TRANS-TO (ASSOC (CADR PATH1) TR1))
                                   (CADR (BUTTON-SPEC))))
          (STATE-PRED (ASSOC (TRANS-TO (ASSOC (CADR PATH2) TR2))
```

```
                       (CADR (MIRROR-PROGRAM-SPEC))))
          (CDR PATH2)
          (STATE-TRANS-LIST (ASSOC (TRANS-TO (ASSOC (CADR PATH2) TR2))
                                   (CADR (MIRROR-PROGRAM-SPEC))))
          (CDR H)
          (STATE-PRED (ASSOC (TRANS-TO (ASSOC (CADR (MIRROR-MAP H PATH1 PATH2))
                                              TR3))
                             (CADR (MIRROR-SYSTEM-SPEC))))
          (STATE-TRANS-LIST
                      (ASSOC (TRANS-TO (ASSOC (CADR (MIRROR-MAP H PATH1 PATH2))
                                              TR3))
                             (CADR (MIRROR-SYSTEM-SPEC)))))
        t)
    t))

(prove-lemma unsatisfiesp-assoc-easy1 (rewrite)
  (implies
   (and
    (listp h)
    (listp (cdr h))
    (not (assoc (cadr path) tr)))
   (not (satisfiesp-with-path-assoc-helper p tr h c b path))))

(prove-lemma unsatisfiesp-assoc-easy2 (rewrite)
  (implies
   (and
    (listp h)
    (listp (cdr h))
    (let
        ((pick-trans (assoc (cadr path) tr)))
      (let
        ((new-state (assoc (car pick-trans) b))
         (new-counters (update-counter-alist c (cadr pick-trans))))
        (not (satisfiesp-with-path-assoc-helper
         (cadr new-state) (caddr new-state)
         (cdr h) new-counters b (cdr path))))))
   (not (satisfiesp-with-path-assoc-helper p tr h c b path))))

(prove-lemma cdr-mirror-map (rewrite)
  (equal (cdr (mirror-map h p1 p2))
         (if (listp h)
             (mirror-map (cdr h) (cdr p1) (cdr p2))
           0)))

(prove-lemma car-mirror-map (rewrite)
  (equal
   (car (mirror-map h path1 path2))
    (if (listp h)
       (cond
        ((equal (car path1) 'going-down) 'going-off)
        ((equal (car path1) 'going-up) 'going-on)
        ((equal (car path1) 'up)
         (if (equal (car path2) 'a1)
             (if (and
                  (equal (cadr (assoc 'newsig (car h))) 0)
                  (equal (cadr (assoc 'output (car h))) 0))
                 'on
               'going-on)
           (if (equal (cadr (assoc 'output (car h))) 0)
               'on
             'going-on)))
        (t    ;; 'down
```

```
        (if (equal (car path2) 'a1)
            (if (and
                 (equal (cadr (assoc 'newsig (car h))) 1)
                 (equal (cadr (assoc 'output (car h))) 1))
                'off
              'going-off)
          (if (equal (cadr (assoc 'output (car h))) 1)
              'off
            'going-off))))
      0))
  ((expand (mirror-map h path1 path2))))

(prove-lemma assoc-append (rewrite)
  (implies
   (not (equal k 0))
  (equal
   (assoc k (append a b))
   (if (assoc k a) (assoc k a) (assoc k b)))))

(prove-lemma assoc-cons (rewrite)
  (equal (assoc k (cons a b))
         (if (equal (car a) k)
             a
           (assoc k b))))

(defn no-litatom-cars (list)
  (if (listp list)
      (and
       (not (litatom (caar list)))
       (no-litatom-cars (cdr list)))
    t))

(prove-lemma no-litatom-cars-update-assoc (rewrite)
  (equal (no-litatom-cars (update-counter-alist c r))
         (no-litatom-cars c)))


(prove-lemma assoc-litatom-no-litatom-cars (rewrite)
  (implies
   (and
    (litatom k)
    (no-litatom-cars l))
   (equal (assoc k l) f)))


;;; Is the first trans and the preds of the first two states
;;; satisfied?
(defn satisfiesp-first-trans (pred trans h counters b-states path)
  (if (listp h)
      (if (listp (cdr h))
          (and
           (assoc (car path) b-states)
           (my-eval pred (append counters (car h)) nil)
           (let
               ((pick-trans (assoc (cadr path) trans)))
             (let
                 ((new-state (assoc (trans-to pick-trans) b-states))
                  (new-counters (update-counter-alist
                                 counters (trans-resets pick-trans))))
               (and
                pick-trans
                (my-eval
```

```
                (trans-pred pick-trans)
                (append new-counters (cadr h))
                (append counters (car h)))
               (assoc (cadr path) b-states)
               (my-eval (state-pred new-state) (append new-counters (cadr h)) nil)))))
        (and
         (assoc (car path) b-states)
         (my-eval pred (append counters (car h)) nil)))
    t))

(prove-lemma open-satisfiesp-with-path-assoc (rewrite)
  (implies
   (and
    (listp h)
    (listp (cdr h))
    (equal (cadr path) x))
   (equal
    (satisfiesp-with-path-assoc-helper p tr h c b path)
    (and
     (satisfiesp-first-trans p tr h c b path)
     (let ((pick-trans (assoc (cadr path) tr)))
       (let ((new-state (assoc (trans-to pick-trans) b))
             (new-counters (update-counter-alist
                            c (trans-resets pick-trans))))
         (satisfiesp-with-path-assoc-helper
          (state-pred new-state) (state-trans-list new-state)
          (cdr h) new-counters b (cdr path)))))))
  ((expand (satisfiesp-with-path-assoc-helper p tr h c b path))))


(defn possible-pathp (path h b-states)
  (if (listp h)
      (and
       (assoc (car path) b-states)
       (implies
        (listp (cdr h))
        (and
         (assoc (cadr path) b-states)
         (assoc (cadr path) (state-trans-list (assoc (car path) b-states)))
         (possible-pathp (cdr path) (cdr h) b-states))))
    t))

(prove-lemma satisfiesp-with-path-assoc-helper-possible (rewrite)
  (implies
   (and
    (satisfiesp-with-path-assoc-helper p tr h c b path)
    (equal tr (state-trans-list (assoc (car path) b))))
   (possible-pathp path h b)))

(prove-lemma assoc-cadr-button-spec (rewrite)
  (iff
   (assoc x (cadr (button-spec)))
   (or (equal x 'up)
       (equal x 'down)
       (equal x 'going-up)
       (equal x 'going-down))))

(prove-lemma assoc-cadr-mirror-program-spec (rewrite)
  (iff
   (assoc x (cadr (mirror-program-spec)))
   (or (equal x 'a0)
       (equal x 'a1))))
```

```
(prove-lemma possible-pathp-button-spec-open (rewrite)
  (implies
   (and
    (listp h)
    (listp (cdr h)))
    (equal
     (possible-pathp path h (cadr (button-spec)))
     (and
      (or
       (and (equal (car path) 'up)
            (or (equal (cadr path) 'up) (equal (cadr path) 'going-down)))
       (and (equal (car path) 'going-down)
            (or (equal (cadr path) 'going-down) (equal (cadr path) 'down)))
       (and (equal (car path) 'down)
            (or (equal (cadr path) 'down) (equal (cadr path) 'going-up)))
       (and (equal (car path) 'going-up)
            (or (equal (cadr path) 'up) (equal (cadr path) 'going-up))))
      (possible-pathp (cdr path) (cdr h) (cadr (button-spec)))))))

(prove-lemma possible-pathp-mirror-program-spec-open (rewrite)
  (implies
   (and
    (listp h)
    (listp (cdr h)))
    (equal
     (possible-pathp path h (cadr (mirror-program-spec)))
     (and
      (and
       (or (equal (car path) 'a0) (equal (car path) 'a1))
       (or (equal (cadr path) 'a0) (equal (cadr path) 'a1)))
      (possible-pathp (cdr path) (cdr h) (cadr (mirror-program-spec)))))))

(prove-lemma assoc-constant-constant (rewrite)
  (and
   (equal (assoc 'up (cadr (button-spec)))
          '(up
            (and (equal signal 0) (equal action 'f))
            ((up nil (true))
             (going-down ((c 1))
                         (and (equal signal 1)
                              (and (equal action 'depress) (lessp 40000 (c 2))))))))
   (equal (assoc 'going-down (cadr (button-spec)))
          '(going-down
            (lessp (c 1) 30000)
            ((going-down nil (equal action 'f)) (down nil (true)))))
   (equal (assoc 'down (cadr (button-spec)))
          '(down
            (and (equal signal 1) (equal action 'f))
            ((down nil (true))
             (going-up ((c 2))
                       (and (equal signal 0)
                            (and (equal action 'release) (lessp 40000 (c 1))))))))
   (equal (assoc 'going-up (cadr (button-spec)))
          '(going-up
           (lessp (c 2) 30000)
           ((going-up nil (equal action 'f)) (up nil (true)))))

   (equal (assoc 'a0 (cadr (mirror-program-spec)))
          '(a0
            (lessp (c 7) 1000)
```

```
          ((a0 nil
               (and
                (equal newsig (old newsig))
                (equal output (old output))))
           (a1 ((c 8))
               (and (equal output (old output))
                    (equal newsig (old signal)))))))

   (equal (assoc 'a1 (cadr (mirror-program-spec)))
          '(a1 (lessp (c 8) 1000)
               ((a0 ((c 7))
                    (and (equal output (old newsig))
                         (equal newsig (old newsig))))
                (a1 nil
                    (and (equal output (old output))
                         (equal newsig (old newsig)))))))))


(prove-lemma my-eval-open (rewrite)
  (and
   (equal
    (my-eval (list 'equal a b) a1 a2)
    (equal (my-eval a a1 a2) (my-eval b a1 a2)))
   (equal
    (my-eval (list 'lessp a b) a1 a2)
    (lessp (my-eval a a1 a2) (my-eval b a1 a2)))
   (equal
    (my-eval (list 'not a) a1 a2)
    (not (my-eval a a1 a2)))
   (equal
    (my-eval (list 'plus a b) a1 a2)
    (plus (my-eval a a1 a2) (my-eval b a1 a2)))
   (equal
    (my-eval (list 'remainder a b) a1 a2)
    (remainder (my-eval a a1 a2) (my-eval b a1 a2)))
   (equal
    (my-eval (list 'difference a b) a1 a2)
    (difference (my-eval a a1 a2) (my-eval b a1 a2)))
   (equal
    (my-eval (list 'if a b c) a1 a2)
    (if (my-eval a a1 a2) (my-eval b a1 a2) (my-eval c a1 a2)))
   (equal
    (my-eval (list 'and a b) a1 a2)
    (and (my-eval a a1 a2) (my-eval b a1 a2)))
   (equal
    (my-eval (list 'or a b) a1 a2)
    (or (my-eval a a1 a2) (my-eval b a1 a2)))
   (equal
    (my-eval (list 'old a) a1 a2)
    (cadr (assoc a a2)))
   (equal
    (my-eval (list 'c a) a1 a2)
    (cadr (assoc (list 'c a) a1)))
   (equal
    (my-eval (list 'quote a) a1 a2)
    a)
   (equal
    (my-eval '(true) a1 a2)
    t)
   (equal
    (my-eval '(false) a1 a2)
    f)
```

```
   (implies
    (numberp n)
    (equal
     (my-eval n a1 a2)
     n))
   (implies
    (litatom a)
    (equal (my-eval a a1 a2) (cadr (assoc a a1))))))


(prove-lemma paths-are-found-in-mirror (rewrite)
  (implies
   (and
    (listp h)
    (listp (cdr h))
    (assoc '(c 1) c1)
    (assoc '(c 2) c1)
    (assoc '(c 7) c2)
    (assoc '(c 8) c2)
    (assoc '(c 3) c3)
    (assoc '(c 4) c3)
    (no-litatom-cars c1)
    (no-litatom-cars c2)
    (no-litatom-cars c3)
    (satisfiesp-with-path-assoc-helper p1 tr1 h c1 (cadr (button-spec)) path1)
    (satisfiesp-with-path-assoc-helper p2 tr2 h c2 (cadr (mirror-program-spec)) path2)
    (equal p1 (state-pred (assoc (car path1) (cadr (button-spec)))))
    (equal tr1 (state-trans-list (assoc (car path1) (cadr (button-spec)))))
    (equal p2 (state-pred (assoc (car path2) (cadr (mirror-program-spec)))))
    (equal tr2 (state-trans-list (assoc (car path2) (cadr (mirror-program-spec)))))
    (mirror-system-invariant
       (cadr (assoc '(c 1) c1)) (cadr (assoc '(c 2) c1))
       (cadr (assoc '(c 3) c3)) (cadr (assoc '(c 4) c3))
       (cadr (assoc '(c 7) c2)) (cadr (assoc '(c 8) c2))
       (car path1) (car path2) (cadr (assoc 'signal (car h)))
       (cadr (assoc 'output (car h))) (cadr (assoc 'newsig (car h)))))
  (assoc
   (car (mirror-map (cdr h) (cdr path1) (cdr path2)))
   (caddr (assoc (car (mirror-map h path1 path2))
                 (cadr (mirror-system-spec))))))
  ((disable-theory t)
   (use (satisfiesp-with-path-assoc-helper-possible
         (tr tr1) (b (cadr (button-spec))) (p p1) (c c1) (path path1))
        (satisfiesp-with-path-assoc-helper-possible
         (tr tr2) (b (cadr (mirror-program-spec))) (p p2) (c c2) (path path2)))
   (enable-theory ground-zero)
   (enable car-cdr-if-cons plus-0 my-eval-open state-pred state-trans-list trans-to
           trans-resets trans-pred assoc-constant-constant
           possible-pathp-mirror-program-spec-open satisfiesp-first-trans
           possible-pathp-button-spec-open mirror-system-spec n31000-props n32000-props
           n30000-props assoc-cadr-button-spec assoc-cadr-mirror-program-spec
           mirror-system-invariant car-mirror-map assoc-update-counter-alist
           assoc-update-counter-alist-iff *1*all-cars-litatoms
           open-satisfiesp-with-path-assoc assoc-append assoc-cons
           no-litatom-cars-update-assoc assoc-litatom-no-litatom-cars)))


(prove-lemma mirror-invariant-preserved (rewrite)
  (implies
   (and
    (listp h)
    (listp (cdr h))
```

```
 (assoc '(c 1) c1)
 (assoc '(c 2) c1)
 (assoc '(c 7) c2)
 (assoc '(c 8) c2)
 (assoc '(c 3) c3)
 (assoc '(c 4) c3)
 (no-litatom-cars c1)
 (no-litatom-cars c2)
 (no-litatom-cars c3)
 (satisfiesp-with-path-assoc-helper p1 tr1 h c1 (cadr (button-spec)) path1)
 (satisfiesp-with-path-assoc-helper p2 tr2 h c2 (cadr (mirror-program-spec)) path2)
 (equal p1 (state-pred (assoc (car path1) (cadr (button-spec)))))
 (equal tr1 (state-trans-list (assoc (car path1) (cadr (button-spec)))))
 (equal p2 (state-pred (assoc (car path2) (cadr (mirror-program-spec)))))
 (equal tr2 (state-trans-list (assoc (car path2) (cadr (mirror-program-spec)))))
 (equal p3 (state-pred (assoc (car (mirror-map h path1 path2))
                              (cadr (mirror-system-spec)))))
 (equal tr3 (state-trans-list (assoc (car (mirror-map h path1 path2))
                                     (cadr (mirror-system-spec)))))
 (mirror-system-invariant
    (cadr (assoc '(c 1) c1)) (cadr (assoc '(c 2) c1))
    (cadr (assoc '(c 3) c3)) (cadr (assoc '(c 4) c3))
    (cadr (assoc '(c 7) c2)) (cadr (assoc '(c 8) c2))
    (car path1) (car path2) (cadr (assoc 'signal (car h)))
    (cadr (assoc 'output (car h))) (cadr (assoc 'newsig (car h)))))
(mirror-system-invariant
           (cadr
            (assoc
             '(c 1)
             (update-counter-alist c1
                      (cadr (assoc (cadr path1)
                                   (caddr (assoc (car path1)
                                                 (cadr (button-spec)))))))))
           (cadr
            (assoc
             '(c 2)
             (update-counter-alist c1
                      (cadr (assoc (cadr path1)
                                   (caddr (assoc (car path1)
                                                 (cadr (button-spec)))))))))
           (cadr
            (assoc
             '(c 3)
             (update-counter-alist c3
                                 (cadr (assoc (car (mirror-map (cdr h)
                                                               (cdr path1)
                                                               (cdr path2)))
                                              tr3)))))
           (cadr
            (assoc
             '(c 4)
             (update-counter-alist c3
                                 (cadr (assoc (car (mirror-map (cdr h)
                                                               (cdr path1)
                                                               (cdr path2)))
                                              tr3)))))
           (cadr
            (assoc
             '(c 7)
             (update-counter-alist c2
              (cadr (assoc (cadr path2)
                           (caddr (assoc (car path2)
```

```
                                        (cadr (mirror-program-spec)))))))))
              (cadr
               (assoc
                '(c 8)
                (update-counter-alist c2
                 (cadr (assoc (cadr path2)
                              (caddr (assoc (car path2)
                                            (cadr (mirror-program-spec)))))))))
              (cadr path1)
              (cadr path2)
              (cadr (assoc 'signal (cadr h)))
              (cadr (assoc 'output (cadr h)))
              (cadr (assoc 'newsig (cadr h)))))
  ((disable-theory t)
   (use (satisfiesp-with-path-assoc-helper-possible
         (tr tr1) (b (cadr (button-spec))) (p p1) (c c1) (path path1))
        (satisfiesp-with-path-assoc-helper-possible
         (tr tr2) (b (cadr (mirror-program-spec))) (p p2) (c c2) (path path2)))
   (enable-theory ground-zero)
   (enable car-cdr-if-cons plus-0 my-eval-open state-pred state-trans-list trans-to
           trans-resets trans-pred assoc-constant-constant
           possible-pathp-mirror-program-spec-open satisfiesp-first-trans
           possible-pathp-button-spec-open mirror-system-spec n31000-props n32000-props
           n30000-props assoc-cadr-button-spec assoc-cadr-mirror-program-spec
           mirror-system-invariant car-mirror-map assoc-update-counter-alist
           assoc-update-counter-alist-iff *1*all-cars-litatoms
           open-satisfiesp-with-path-assoc assoc-append assoc-cons
           no-litatom-cars-update-assoc assoc-litatom-no-litatom-cars)))

(prove-lemma satisfiesp-base-case (rewrite)
  (and
   (implies
    (not (listp h))
    (satisfiesp-with-path-assoc-helper p tr h c b path))
   (implies
    (and
     (not (listp (cdr h)))
     (listp h))
    (equal
     (satisfiesp-with-path-assoc-helper p tr h c b path)
     (and (assoc (car path) b) (my-eval p (append c (car h)) nil))))))

(prove-lemma mirror-base-case1 (rewrite)
          (implies
                (and (listp h)
                     (not (listp (cdr h)))
                     (assoc '(c 1) c1)
                     (assoc '(c 2) c1)
                     (assoc '(c 7) c2)
                     (assoc '(c 8) c2)
                     (assoc '(c 3) c3)
                     (assoc '(c 4) c3)
                     (no-litatom-cars c1)
                     (no-litatom-cars c2)
                     (no-litatom-cars c3)
                     (assoc (car path1) (cadr (button-spec)))
                     (my-eval (cadr (assoc (car path1)
                                           (cadr (button-spec))))
                              (append c1 (car h))
                              nil)
                     (assoc (car path2) (cadr (mirror-program-spec)))
                     (my-eval (cadr (assoc (car path2)
```

```
                                       (cadr (mirror-program-spec))))
                              (append c2 (car h))
                              nil)
                     (mirror-system-invariant (cadr (assoc '(c 1) c1))
                                              (cadr (assoc '(c 2) c1))
                                              (cadr (assoc '(c 3) c3))
                                              (cadr (assoc '(c 4) c3))
                                              (cadr (assoc '(c 7) c2))
                                              (cadr (assoc '(c 8) c2))
                                              (car path1)
                                              (car path2)
                                              (cadr (assoc 'signal (car h)))
                                              (cadr (assoc 'output (car h)))
                                              (cadr (assoc 'newsig (car h)))))
                (my-eval (cadr (assoc (car (mirror-map h path1 path2))
                                      (cadr (mirror-system-spec))))
                         (append c3 (car h))
                         nil)))

(prove-lemma mirror-base-case2 (rewrite)
  (assoc (car (mirror-map (cons z x) path1 path2))
         (cadr (mirror-system-spec))))

(prove-lemma mirror-map-open
 (rewrite)
 (implies
  (and (equal (car path1) x)
       (equal (car path2) y))
  (equal
   (mirror-map h path1 path2)
   (if
    (listp h)
    (cons
        (case (car path1)
              (going-down 'going-off)
              (going-up 'going-on)
              (up (cond ((equal (car path2) 'a1)
                         (if (and (equal (cadr (assoc 'newsig (car h)))
                                         0)
                                  (equal (cadr (assoc 'output (car h)))
                                         0))
                             'on
                             'going-on))
                        ((equal (cadr (assoc 'output (car h)))
                                0)
                         'on)
                        (t 'going-on)))
              (otherwise (cond ((equal (car path2) 'a1)
                                (if (and (equal (cadr (assoc 'newsig (car h)))
                                                1)
                                         (equal (cadr (assoc 'output (car h)))
                                                1))
                                    'off
                                    'going-off))
                               ((equal (cadr (assoc 'output (car h)))
                                       1)
                                'off)
                               (t 'going-off))))
        (mirror-map (cdr h)
                    (cdr path1)
                    (cdr path2)))
    nil))))
```

```
(prove-lemma open-satisfiesp-with-path-assoc2 (rewrite)
  (implies
   (and
    (listp h)
    (listp (cdr h)))
   (equal
    (satisfiesp-with-path-assoc-helper p tr h c b (cons x (cons y path)))
    (and
     (satisfiesp-first-trans p tr h c b (cons x (cons y path)))
     (let ((pick-trans (assoc y tr)))
       (let ((new-state (assoc (trans-to pick-trans) b))
             (new-counters (update-counter-alist
                            c (trans-resets pick-trans))))
         (satisfiesp-with-path-assoc-helper
          (state-pred new-state) (state-trans-list new-state)
          (cdr h) new-counters b (cons y path))))))))

(prove-lemma mirror-spec-satisfied (rewrite)
  (implies
   (and
    (listp h)
    (listp (cdr h))
    (assoc '(c 1) c1)
    (assoc '(c 2) c1)
    (assoc '(c 7) c2)
    (assoc '(c 8) c2)
    (assoc '(c 3) c3)
    (assoc '(c 4) c3)
    (no-litatom-cars c1)
    (no-litatom-cars c2)
    (no-litatom-cars c3)
    (satisfiesp-with-path-assoc-helper p1 tr1 h c1 (cadr (button-spec)) path1)
    (satisfiesp-with-path-assoc-helper p2 tr2 h c2 (cadr (mirror-program-spec)) path2)
    (satisfiesp-with-path-assoc-helper
             (cadr (assoc (car (mirror-map (cdr h)
                                           (cdr path1)
                                           (cdr path2)))
                          (cadr (mirror-system-spec))))
             (caddr (assoc (car (mirror-map (cdr h)
                                            (cdr path1)
                                            (cdr path2)))
                           (cadr (mirror-system-spec))))
             (cdr h)
             (update-counter-alist c3
                    (cadr (assoc (car (mirror-map (cdr h)
                                                  (cdr path1)
                                                  (cdr path2)))
                                 (caddr (assoc (car (mirror-map h path1 path2))
                                               (cadr (mirror-system-spec)))))))
             (cadr (mirror-system-spec))
             (mirror-map (cdr h)
                         (cdr path1)
                         (cdr path2)))
    (equal p1 (state-pred (assoc (car path1) (cadr (button-spec)))))
    (equal tr1 (state-trans-list (assoc (car path1) (cadr (button-spec)))))
    (equal p2 (state-pred (assoc (car path2) (cadr (mirror-program-spec)))))
    (equal tr2 (state-trans-list (assoc (car path2) (cadr (mirror-program-spec)))))
    (mirror-system-invariant
       (cadr (assoc '(c 1) c1)) (cadr (assoc '(c 2) c1))
       (cadr (assoc '(c 3) c3)) (cadr (assoc '(c 4) c3))
       (cadr (assoc '(c 7) c2)) (cadr (assoc '(c 8) c2))
```

```
     (car path1) (car path2) (cadr (assoc 'signal (car h)))
     (cadr (assoc 'output (car h))) (cadr (assoc 'newsig (car h))))
   (equal p3
        (cadr (assoc (car (mirror-map h path1 path2))
                     (cadr (mirror-system-spec)))))
  (equal tr3
        (caddr (assoc (car (mirror-map h path1 path2))
                      (cadr (mirror-system-spec))))))
 (satisfiesp-with-path-assoc-helper p3 tr3 h c3
                                    (cadr (mirror-system-spec))
                                    (mirror-map h path1 path2)))
 ((disable-theory t)
  (use (satisfiesp-with-path-assoc-helper-possible
       (tr tr1) (b (cadr (button-spec))) (p p1) (c c1) (path path1))
      (satisfiesp-with-path-assoc-helper-possible
       (tr tr2) (b (cadr (mirror-program-spec))) (p p2) (c c2) (path path2)))
  (enable-theory ground-zero)
  (enable car-cdr-if-cons plus-0 my-eval-open state-pred state-trans-list trans-to
         trans-resets trans-pred assoc-constant-constant
         possible-pathp-mirror-program-spec-open satisfiesp-first-trans mirror-map-open
         possible-pathp-button-spec-open mirror-system-spec n31000-props n32000-props
         n30000-props assoc-cadr-button-spec assoc-cadr-mirror-program-spec
         mirror-system-invariant assoc-update-counter-alist assoc-update-counter-alist-iff
         *1*all-cars-litatoms open-satisfiesp-with-path-assoc
         open-satisfiesp-with-path-assoc2 assoc-append assoc-cons
         no-litatom-cars-update-assoc assoc-litatom-no-litatom-cars)))


(prove-lemma satisfiesp-thm-helper (rewrite)
 (implies
  (and
   (assoc '(c 1) c1)
   (assoc '(c 2) c1)
   (assoc '(c 7) c2)
   (assoc '(c 8) c2)
   (assoc '(c 3) c3)
   (assoc '(c 4) c3)
   (no-litatom-cars c1)
   (no-litatom-cars c2)
   (no-litatom-cars c3)
   (satisfiesp-with-path-assoc-helper p1 tr1 h c1 (cadr (button-spec)) path1)
   (satisfiesp-with-path-assoc-helper p2 tr2 h c2 (cadr (mirror-program-spec)) path2)
   (equal p1 (state-pred (assoc (car path1) (cadr (button-spec)))))
   (equal tr1 (state-trans-list (assoc (car path1) (cadr (button-spec)))))
   (equal p2 (state-pred (assoc (car path2) (cadr (mirror-program-spec)))))
   (equal tr2 (state-trans-list (assoc (car path2) (cadr (mirror-program-spec)))))
   (implies
    (listp h)
    (and
     (mirror-system-invariant
      (cadr (assoc '(c 1) c1)) (cadr (assoc '(c 2) c1))
      (cadr (assoc '(c 3) c3)) (cadr (assoc '(c 4) c3))
      (cadr (assoc '(c 7) c2)) (cadr (assoc '(c 8) c2))
      (car path1) (car path2) (cadr (assoc 'signal (car h)))
      (cadr (assoc 'output (car h))) (cadr (assoc 'newsig (car h))))
     (equal p3 (state-pred (assoc (car (mirror-map h path1 path2))
                                  (cadr (mirror-system-spec)))))
     (equal tr3 (state-trans-list (assoc (car (mirror-map h path1 path2))
                                         (cadr (mirror-system-spec))))))))
  (satisfiesp-with-path-assoc-helper p3 tr3 h c3 (cadr (mirror-system-spec))
                                     (mirror-map h path1 path2)))
 ((disable-theory t)
```

```
   (enable-theory ground-zero)
   (induct (sth-induct c1 c2 c3 p1 path1 tr1 p2 path2 tr2 h p3 tr3))
   (enable satisfiesp-base-case mirror-base-case2 mirror-base-case1 mirror-spec-satisfied
           paths-are-found-in-mirror no-litatom-cars-update-assoc mirror-invariant-preserved
           car-assoc cdr-mirror-map unsatisfiesp-assoc-easy1 unsatisfiesp-assoc-easy2
           assoc-update-counter-alist-iff update-counter-alist assoc state-pred
           state-trans-list state-name trans-to trans-pred trans-resets)))


(prove-lemma satisfiesp-base-case-a (rewrite)
  (and
   (implies
    (not (listp h))
    (satisfiesp-with-path-helper p tr h c b path))
   (implies
    (and
     (not (listp (cdr h)))
     (listp h))
    (equal
     (satisfiesp-with-path-helper p tr h c b path)
     (and (assoc (car path) b) (my-eval p (append c (car h)) nil))))))

(prove-lemma satisfiesp-base-case-b (rewrite)
  (and
   (implies
    (not (listp h))
    (satisfiesp-eff-helper n p tr h c b))
   (implies
    (and
     (not (listp (cdr h)))
     (listp h))
    (equal
     (satisfiesp-eff-helper n p tr h c b)
     (and (assoc n b) (my-eval p (append c (car h)) nil))))))

(prove-lemma satisfiesp-with-path-helper-means-satisfiesp-helper-rewrite
             (rewrite)
             (implies (and (satisfiesp-with-path-helper p tr h c b path)
                           (equal (car path) n))
                      (satisfiesp-eff-helper n p tr h c b))
             ((use (satisfiesp-with-path-helper-means-satisfiesp-helper))
              (enable satisfiesp-eff-helper-equiv)))

(prove-lemma signal-starts-0-in-mirror nil
  (implies
   (satisfiesp h (button-spec))
   (equal (cadr (assoc 'signal (car h))) 0)))

(prove-lemma mirror-satisfiesp (rewrite)
  (implies
   (and
    (equal (cadr (assoc 'output (car h))) 0)
    (satisfiesp h (button-spec))
    (satisfiesp h (mirror-program-spec)))
   (satisfiesp h (mirror-system-spec)))
  ((disable-theory t)
   (use (satisfiesp-thm-helper
         (path1 (find-path h (button-spec)))
         (path2 (find-path h (mirror-program-spec)))
         (p1 (state-pred (assoc (car (find-path h (button-spec))) (cadr (button-spec)))))
         (tr1 (state-trans-list
               (assoc (car (find-path h (button-spec))) (cadr (button-spec)))))
```

```
        (p2 (state-pred (assoc (car (find-path h (mirror-program-spec)))
                               (cadr (mirror-program-spec)))))
        (tr2 (state-trans-list (assoc (car (find-path h (mirror-program-spec)))
                                      (cadr (mirror-program-spec)))))
        (p3 (state-pred (assoc (car (mirror-map h (find-path h (button-spec))
                                                (find-path h (mirror-program-spec))))
                               (cadr (mirror-system-spec)))))
        (tr3 (state-trans-list (assoc (car (mirror-map h (find-path h (button-spec))
                                                       (find-path h (mirror-program-spec))))
                                      (cadr (mirror-system-spec)))))
        (c1 '(((c 1) 0) ((c 2) 0)))
        (c2 '(((c 8) 0) ((c 7) 0)))
        (c3 '(((c 3) 0) ((c 4) 0))))
       (signal-starts-0-in-mirror))
  (enable-theory ground-zero)
  (enable state-name state-pred state-trans-list behavior-initial *1*counters-of-state-list
          behavior-state-list make-list-alist satisfiesp satisfiesp-eff-helper-equiv
          find-path member-cars cars-uniquep all-transitions-uniquep all-cars-litatoms
          all-transitions-litatoms car-find-path-helper
          satisfiesp-helper-means-with-find-path button-spec mirror-program-spec
          mirror-system-invariant car-mirror-map no-litatom-cars mirror-system-spec
          satisfiesp-thm-helper satisfiesp-with-path-assoc-same satisfiesp-base-case-a
          satisfiesp-base-case-b satisfiesp-with-path-helper-means-satisfiesp-helper-rewrite)
  ))

;;;;;
;;;;;
;;;;; FM9001-specific stuff
;;;;;
;;;;;

;; Given an FM9001 state at the programmer level, calculate the number
;; of microcycles needed to execute it assuming each memory access
;; requires k wait-for-memory cycles.
;;
;; Note the definition of microcycles from the FM9001 proof, which
;; takes a lower-level state and calculates the number of microcycles
;; and is used in the FM9001 correctness theorems
;;

(defn microcycles-constant (state k)
  (let ((mapped-down-state (map-down state)))
    (t_fetch1 (list k k k)
              (read-mem (read-mem (CADDDDDDDDDDAR mapped-down-STATE)
                                  (caaar mapped-down-state))
                        (caadr mapped-down-state))
              (cadar mapped-down-state))))

;; Calculate the number of cycles required to "initialize" when
;; executing the instruction ins assuming each memory access requires
;; k wait-for-memory cycles.
(defn init-time (state k)
  (let ((ireg (READ-MEM (READ-MEM (CADDDDDDDDDDAR (map-down STATE))
                                  (CAAAR (map-down STATE)))
                        (CAADR (map-down STATE))))
        (flags (cadar (map-down state))))
    (cond
     ((and
       (or (store-resultp (store-cc ireg) flags)
           (set-some-flags (set-flags ireg)))
       (and (not (a-immediate-p ireg)) (not (REG-DIRECT-P (MODE-A IREG)))))
      (plus k 8))
```

```
     ((and
       (store-resultp (store-cc ireg) flags)
       (or (a-immediate-p ireg) (REG-DIRECT-P (MODE-A IREG)))  ; redundant given previous case
       (not (reg-direct-p (mode-b ireg)))
;       (unary-op-code-p (op-code ireg))    we don't use memory->memory instr
       )
      (plus k 10))
     (t 1))))

(disable init-time)
(disable *1*init-time)

;; Does current instruction both read and write?
(defn read-write-statep (state)
  (let ((ireg (read-mem (read-mem (caddddddddddar (map-down state))
                                  (caaar (map-down state)))
                        (caadr (map-down state)))))
    (and (not (reg-direct-p (mode-b ireg)))
         (or (not (unary-op-code-p (op-code ireg)))
             (and (not (a-immediate-p ireg)) (not (reg-direct-p (mode-a ireg))))))))


(defn update-state-with-inputs (inputs state)
  (if (listp inputs)
      (update-state-with-inputs
       (cdr inputs)
       (list
        (car state)
        (write-mem (nat-to-v (caar inputs) 32) (cadr state)
                   (nat-to-v (cadar inputs) 32))))
    state))

(defn last (list)
  (if (listp list)
      (if (listp (cdr list))
          (last (cdr list))
        (car list))
    nil))

(prove-lemma lessp-init-time-total-time (rewrite)
  (implies
   (lessp 0 memory-access-time)
   (lessp (init-time state memory-access-time)
          (microcycles-constant state memory-access-time)))
  ((enable init-time microcycles-constant or* if*)))

(prove-lemma lessp-total-time-init-time (rewrite)
  (not (lessp (microcycles-constant state memory-access-time)
              (init-time state memory-access-time)))
  ((enable init-time microcycles-constant or* if*)))

(prove-lemma lessp-0-init-time (rewrite)
  (lessp 0 (init-time state memory-access-time))
  ((expand (init-time state memory-access-time))))

(prove-lemma lessp-1-microcycles-constant (rewrite)
  (lessp 1 (microcycles-constant state memory-access-time))
  ((expand (microcycles-constant state memory-access-time))))

(prove-lemma length-cdr-rewrite (rewrite)
  (equal (length (cdr x)) (sub1 (length x))))
```

```
(prove-lemma length-nthcdr (rewrite)
  (equal (length (nthcdr n l)) (difference (length l) n))
  ((induct (nthcdr n l))))

(disable nthcdr)
(disable *1*nthcdr)
(disable fm9001-step)
(disable microcycles-constant)

;; For an FM9001 state, a list of inputs, and a memory access time,
;; calculate a new FM9001 state.  If a read-write instruction is
;; executed, return 0.

(defn fm9001-rt (state inputs-list mat)
  (if (nlistp inputs-list)
      state
    (if (read-write-statep state) 0
      (let ((init-time (init-time state mat))
            (total-time (microcycles-constant state mat)))
        (if (not (lessp init-time (length inputs-list)))
            (update-state-with-inputs (last inputs-list) state)
          (let ((new-state
                 (fm9001-step
                  (update-state-with-inputs (nth (sub1 init-time) inputs-list) state)
                  (nat-to-v 15 (reg-size)))))
            (if (not (lessp total-time (length inputs-list)))
                (update-state-with-inputs (last inputs-list) new-state)
              (fm9001-rt new-state (nthcdr total-time inputs-list) mat)))))))
  ((lessp (length inputs-list))))

;; Extract values from an FM9001 state.  loclist is used to decide
;; which values are needed.  Each element of loclist is a pair of the
;; form (loc name).  loc is a natural that serves as the value address
;; or a list of one natural that is the register number.  name is the
;; name to be associated with that value in the output alist.

(defn extract-locs (loclist state)
  (if (listp loclist)
      (cons
       (list (cadar loclist)
             (if (listp (caar loclist))
                 (v-to-nat (read-mem (nat-to-v (caaar loclist) (reg-size))
                                     (regs (car state))))
               (v-to-nat (read-mem (nat-to-v (caar loclist) 32)
                                   (cadr state)))))
       (extract-locs (cdr loclist) state))
    nil))

(defn fm9001-rt-history-helper (state mat inputs-list loclist n)
  (if (lessp n (length inputs-list))
      (let ((next-state (fm9001-rt state (firstn (add1 n) inputs-list) mat)))
        (if (equal next-state 0)
            nil
          (cons
           (extract-locs loclist next-state)
           (fm9001-rt-history-helper state mat inputs-list loclist (add1 n)))))
    nil)
  ((lessp (difference (length inputs-list) n))))

(defn fm9001-rt-history (state mat inputs-list loclist)
  (fm9001-rt-history-helper state mat inputs-list loclist 0))
```

```
(defn repeat-with-inputs (state inputs-list loclist)
  (if (listp inputs-list)
      (cons
       (extract-locs loclist (update-state-with-inputs (car inputs-list) state))
       (repeat-with-inputs state (cdr inputs-list) loclist))
    nil))

(defn fm9001-rt-history-repeat-helper (state mat inputs-list loclist)
  (let
      ((init-time (init-time state mat))
       (total-time (microcycles-constant state mat)))
    (if (read-write-statep state) nil
      (if (not (lessp init-time (length inputs-list)))
          (repeat-with-inputs state inputs-list loclist)
        (let ((new-state
               (fm9001-step
                (update-state-with-inputs (nth (sub1 init-time) inputs-list) state)
                (nat-to-v 15 (reg-size)))))
          (if (not (lessp total-time (length inputs-list)))
              (append
               (repeat-with-inputs state (firstn init-time inputs-list) loclist)
               (repeat-with-inputs new-state (nthcdr init-time inputs-list) loclist))
            (append
             (repeat-with-inputs state (firstn init-time inputs-list) loclist)
             (append
              (repeat-with-inputs
               new-state (nthcdr init-time (firstn total-time inputs-list)) loclist)
              (fm9001-rt-history-repeat-helper
               new-state mat (nthcdr total-time inputs-list) loclist))))))))
  ((lessp (length inputs-list))))

(prove-lemma firstn-all-simplify (rewrite)
  (implies
   (properp x)
   (equal (firstn (length x) x) x))
  ((enable firstn)))

(prove-lemma car-append (rewrite)
  (equal (car (append x y)) (if (listp x) (car x) (car y))))

(prove-lemma cdr-append (rewrite)
  (equal (cdr (append x y)) (if (listp x) (append (cdr x) y) (cdr y))))

(prove-lemma listp-append (rewrite)
  (equal (listp (append x y)) (or (listp x) (listp y)))
  ((enable append)))

(prove-lemma equal-append-nlistp (rewrite)
  (implies
   (not (listp x))
   (equal
    (equal (append a b) x)
    (and
     (nlistp a)
     (equal b x)))))

(prove-lemma equal-append-as-firstn-nthcdr (rewrite)
  (implies
   (and
    (properp x)
    (properp b)
    (properp a))
```

```
  (equal
   (equal (append a b) x)
   (and
    (equal (plus (length a) (length b)) (length x))
    (equal (firstn (length a) x) a)
    (equal (nthcdr (length a) x) b))))
  ((enable firstn nthcdr)
    (induct (double-cdr-induction a x))))

(prove-lemma properp-repeat-with-inputs (rewrite)
  (properp (repeat-with-inputs s n l)))

(prove-lemma properp-fm9001-rt-history-helper (rewrite)
  (properp (fm9001-rt-history-helper s m i l n)))

(prove-lemma length-repeat-with-inputs (rewrite)
  (equal (length (repeat-with-inputs s i l)) (length i)))

(prove-lemma equal-repeat-with-inputs-nil (rewrite)
  (equal
   (equal (repeat-with-inputs s i l) nil)
   (not (listp i))))

(prove-lemma equal-fm9001-repeat-nil (rewrite)
  (equal
  (equal (fm9001-rt-history-repeat-helper s m i l) nil)
  (or (read-write-statep s) (not (listp i)))))


(prove-lemma listp-firstn (rewrite)
  (equal (listp (firstn n l)) (and (lessp 0 n) (listp l)))
  ((enable firstn)))

(prove-lemma nth-nlistp (rewrite)
  (implies
   (nlistp l)
   (equal (nth n l) 0))
  ((enable nth)))

(defn make-properp (l)
  (append l nil))

(prove-lemma firstn-length-better (rewrite)
  (equal (firstn (length l) l) (make-properp l))
  ((enable firstn)))

(prove-lemma last-append (rewrite)
  (equal (last (append x y))
         (if (listp y) (last y) (last x)))
  ((induct (append x y))))

(prove-lemma firstn-open-on-length-1 (rewrite)
  (implies
   (equal (length l) 1)
   (equal (firstn n l) (if (zerop n) nil (list (car l)))))
  ((expand (firstn n l))))

(prove-lemma last-firstn (rewrite)
  (equal
   (last (firstn n l))
   (if (lessp 0 n)
       (if (lessp n (length l))
```

```
          (nth (sub1 n) l)
        (last l))
    nil))
  ((induct (firstn n l))
   (enable firstn-add1-cons)))

(prove-lemma nth-too-big (rewrite)
  (implies
   (not (lessp n (length l)))
   (equal (nth n l) 0))
  ((induct (nth n l))))


(prove-lemma nth-repeat-with-inputs (rewrite)
  (equal
   (nth n (repeat-with-inputs s i l))
   (if (lessp n (length i))
       (extract-locs l (update-state-with-inputs (nth n i) s))
     0))
  ((enable nth)
   (induct (nth n i))))


(prove-lemma nth-means-last-when (rewrite)
  (implies
   (and
    (equal (length l) x)
    (equal (add1 y) x))
   (equal (nth y l) (last l)))
  ((enable nth)))

(prove-lemma nth-nthcdr (rewrite)
  (equal
   (nth n1 (nthcdr n2 l))
   (nth (plus n1 n2) l))
  ((enable nth nthcdr)))

(defn sub1-sub1-cdr-induct (n1 n2 l)
  (if (zerop n1) t
    (sub1-sub1-cdr-induct (sub1 n1) (sub1 n2) (cdr l))))

(prove-lemma nth-firstn (rewrite)
  (equal
   (nth n1 (firstn n2 l))
   (if (and (lessp n1 n2) (lessp n1 (length l)))
       (nth n1 l)
     0))
  ((enable nth firstn)
   (induct (sub1-sub1-cdr-induct n1 n2 l))))

(prove-lemma equal-sub1-init-time-0-hack (rewrite)
  (implies
   (equal (sub1 (init-time s k)) n)
   (equal (init-time s k) (add1 n))))

(prove-lemma plus-difference-hack (rewrite)
  (equal
   (plus (difference a b) b)
   (if (lessp a b) b (fix a))))

(prove-lemma equal-update-state-with-inputs-0 (rewrite)
  (equal
```

```
   (equal (update-state-with-inputs i s) 0)
   (and
    (nlistp i)
    (equal s 0))))

(prove-lemma nth-append-better (rewrite)
  (equal (nth n (append x y))
         (if (lessp n (length x))
             (nth n x)
           (nth (difference n (length x)) y)))
  ((enable nth)))

(prove-lemma car-repeat-with-inputs (rewrite)
  (equal
   (car (repeat-with-inputs s i l))
   (if (listp i) (extract-locs l (update-state-with-inputs (car i) s)) 0)))

(prove-lemma car-nthcdr (rewrite)
  (equal (car (nthcdr n l)) (nth n l))
  ((enable nth nthcdr)))

(prove-lemma listp-repeat-with-inputs (rewrite)
  (equal (listp (repeat-with-inputs s i l)) (listp i)))

(prove-lemma nth-make-properp (rewrite)
  (equal (nth n (make-properp l)) (nth n l)))

(prove-lemma nth-cdr-make-properp-hack (rewrite)
  (equal (nth n (cons a (make-properp l))) (nth n (cons a l)))
  ((enable nth)))

(prove-lemma nth-cons-firstn-hack (rewrite)
  (equal
   (nth n (cons a (firstn n2 b)))
   (if (zerop n) a
     (if (lessp n2 n)
         0
       (nth (sub1 n) b))))
  ((enable firstn nth)))

(prove-lemma difference-difference1 (rewrite)
  (equal
   (difference (difference a b) c)
   (difference a (plus b c))))

(prove-lemma difference-difference2 (rewrite)
  (equal
   (difference a (difference b c))
   (if (lessp b c)
       (fix a)
     (difference (plus a c) b))))

(prove-lemma equal-nthcdr-nil (rewrite)
  (equal
   (equal (nthcdr n l) nil)
   (and
    (equal (fix n) (length l))
    (properp l)))
  ((enable nthcdr)))

(prove-lemma nthcdr-too-big (rewrite)
  (implies
```

```
   (lessp (length l) n)
   (equal (nthcdr n l) 0))
  ((enable nthcdr)
   (induct (nthcdr n l))))

(prove-lemma nthcdr-nlistp (rewrite)
  (implies
   (nlistp l)
   (equal (nthcdr n l) (if (zerop n) l 0)))
  ((enable nthcdr)))

(prove-lemma firstn-too-big (rewrite)
  (implies
   (not (lessp x (length l)))
   (equal (firstn x l) (make-properp l)))
  ((enable firstn)
   (induct (firstn x l))))

(prove-lemma nthcdr-firstn (rewrite)
  (equal
   (nthcdr n1 (firstn n2 l))
   (if (lessp (length l) n2)
       (nthcdr n1 (make-properp l))
     (if (lessp n2 n1)
         0
       (firstn (difference n2 n1) (nthcdr n1 l)))))
  ((enable firstn nthcdr)))

(prove-lemma nthcdr-append (rewrite)
  (equal
   (nthcdr n (append x y))
   (if (lessp (length x) n)
       (nthcdr (difference n (length x)) y)
     (append (nthcdr n x) y)))
  ((enable nthcdr)))


(defn fm9001-rt-history-repeat-helper-induct (n state mat inputs-list)
  (if (lessp n (microcycles-constant state mat))
      t
    (fm9001-rt-history-repeat-helper-induct
     (difference n (microcycles-constant state mat))
     (fm9001-step
      (update-state-with-inputs (nth (sub1 (init-time state mat)) inputs-list) state)
      (nat-to-v 15 (reg-size)))
     mat
     (nthcdr (microcycles-constant state mat) inputs-list))))

(prove-lemma equal-length-small (rewrite)
  (and
   (equal (equal (length l) 0) (not (listp l)))
   (equal (equal (length l) 1) (and (listp l) (not (listp (cdr l)))))))

(prove-lemma lessp-init-time-1 (rewrite)
  (equal (lessp (init-time state mat) 1) f))

(prove-lemma firstn-1 (rewrite)
  (equal (firstn 1 x) (if (listp x) (list (car x)) nil)) ((enable firstn)))

(prove-lemma firstn-2 (rewrite)
  (equal (firstn 2 x) (if (listp x) (cons (car x) (firstn 1 (cdr x))) nil)) ((enable firstn)))
```

```
(defn prefixp (l1 l2)
  (if (listp l1)
      (and
       (equal (car l1) (car l2))
       (listp l2)
       (prefixp (cdr l1) (cdr l2)))
    t))

(prove-lemma prefixp-firstn (rewrite)
  (implies
   (not (lessp n2 n1))
   (prefixp (firstn n1 l) (firstn n2 l)))
  ((enable firstn)))

(prove-lemma prefixp-append (rewrite)
  (equal
   (prefixp (append a b) x)
   (and
    (prefixp a x)
    (prefixp b (nthcdr (length a) x)))))

(prove-lemma prefixp-nthcdr-helper nil
  (implies
   (prefixp (append x y) b)
   (prefixp y (nthcdr (length x) b))))

(defn sub1-cdr-induct (n l)
  (if (listp l)
      (sub1-cdr-induct (sub1 n) (cdr l))
    t))

(prove-lemma equal-x-append-firstn-x (rewrite)
  (implies
   (and
    (equal (nthcdr n x) y)
    (not (lessp (length x) n)))
   (equal (append (firstn n x) y) x))
  ((enable nthcdr firstn)
  (induct (sub1-cdr-induct n x))))

(prove-lemma append-firstn-nthcdr (rewrite)
  (implies
   (not (lessp (length x) n))
   (equal
    (append (firstn n x) (nthcdr n x))
    x))
  ((enable nthcdr firstn)))

(prove-lemma prefixp-nthcdr-nthcdr (rewrite)
  (implies
   (and
    (prefixp a b)
    (not (lessp (length a) n)))
   (prefixp (nthcdr n a) (nthcdr n b)))
  ((use (prefixp-nthcdr-helper (x (firstn n a)) (y (nthcdr n a))))
   (disable prefixp-append)))

(defn sub1-cdr-cdr-induct (n l1 l2)
  (if (zerop n) t
    (sub1-cdr-cdr-induct (sub1 n) (cdr l1) (cdr l2))))

(prove-lemma nth-of-prefix (rewrite)
```

```
  (implies
   (and
    (prefixp x y)
    (lessp n (length x)))
   (equal (nth n y) (nth n x)))
  ((induct (sub1-cdr-cdr-induct n x y))))


(defn fm9001-rt-prefixp-means-induct (s i1 i2 mat)
  (if (not (lessp (microcycles-constant s mat) (length i1))) t
    (fm9001-rt-prefixp-means-induct
     (FM9001-STEP
      (UPDATE-STATE-WITH-INPUTS (NTH (SUB1 (INIT-TIME S MAT)) I1) S)
      (NAT-TO-V 15 (REG-SIZE)))
     (nthcdr (microcycles-constant s mat) i1)
     (nthcdr (microcycles-constant s mat) i2)
     mat))
  ((lessp (length i1))))

(prove-lemma prefixp-means-lessp-length (rewrite)
  (implies
   (prefixp l1 l2)
   (not (lessp (length l2) (length l1)))))

(prove-lemma fm9001-rt-prefixp-means (rewrite)
  (implies
   (and
    (equal (fm9001-rt s i1 mat) 0)
    (prefixp i1 i2))
   (equal (fm9001-rt s i2 mat) 0))
  ((induct (fm9001-rt-prefixp-means-induct s i1 i2 mat))
   (disable open-nthcdr)
   (expand (fm9001-rt s i2 mat))))

(prove-lemma car-fm9001-rt-history-helper (rewrite)
  (equal
   (car (fm9001-rt-history-helper state mat inputs-list loclist n))
   (if (lessp n (length inputs-list))
       (let ((next-state (fm9001-rt state (firstn (add1 n) inputs-list) mat)))
         (if (equal next-state 0)
             0
           (extract-locs loclist next-state)))
     0))
  ((expand (fm9001-rt-history-helper state mat inputs-list loclist n))))

(prove-lemma nth-fm9001-rt-history-helper-helper nil
  (implies
   (not (lessp n n2))
   (equal
    (nth (difference n n2) (fm9001-rt-history-helper state mat inputs-list loclist n2))
    (if (and
         (lessp n (length inputs-list))
         (not (equal (fm9001-rt state (firstn (add1 n) inputs-list) mat) 0)))
        (extract-locs loclist (fm9001-rt state (firstn (add1 n) inputs-list) mat))
      0)))
  ((disable-theory t)
   (disable extract-locs)
   (enable-theory ground-zero naturals)
   (enable nth nthcdr fm9001-rt-history-helper car-fm9001-rt-history-helper
           equal-update-state-with-inputs-0 nth-nlistp length-firstn listp-firstn
           fm9001-rt-prefixp-means prefixp-firstn fm9001-rt length last firstn-1 firstn-2
           lessp-init-time-1 lessp-1-microcycles-constant firstn-too-big make-properp append
```

```
          lessp-0-init-time equal-length-small)))

;;;; nth of "public" fm9001 model
(prove-lemma nth-fm9001-rt-history-helper (rewrite)
  (equal
    (nth n (fm9001-rt-history-helper state mat inputs-list loclist 0))
    (if (and
         (lessp n (length inputs-list))
         (not (equal (fm9001-rt state (firstn (add1 n) inputs-list) mat) 0)))
        (extract-locs loclist (fm9001-rt state (firstn (add1 n) inputs-list) mat))
      0))
  ((use (nth-fm9001-rt-history-helper-helper (n2 0)))
   (disable-theory t)
   (enable nth)
   (enable-theory ground-zero naturals)))

(prove-lemma open-fm9001-rt (rewrite)
  (implies
   (not (lessp (microcycles-constant state mat) (length inputs-list)))
   (equal
    (fm9001-rt state inputs-list mat)
  (if (nlistp inputs-list)
      state
    (if (read-write-statep state) 0
      (let ((init-time (init-time state mat))
            (total-time (microcycles-constant state mat)))
        (if (not (lessp init-time (length inputs-list)))
            (update-state-with-inputs (last inputs-list) state)
          (let ((new-state
                 (fm9001-step
                  (update-state-with-inputs (nth (sub1 init-time) inputs-list) state)
                  (nat-to-v 15 (reg-size)))))
            (if (not (lessp total-time (length inputs-list)))
                (update-state-with-inputs (last inputs-list) new-state)
              (fm9001-rt new-state (nthcdr total-time inputs-list) mat)))))))))

(prove-lemma nth-length-as-last (rewrite)
  (implies
   (equal (length x) (add1 n))
   (equal (nth n x) (last x))))

(prove-lemma nth-cons-append-hack (rewrite)
  (implies
   (nlistp x)
   (equal (nth n (cons a (append y x))) (nth n (cons a y))))
  ((enable nth)))

(prove-lemma length-nlistp-cdr (rewrite)
  (implies
   (and
    (not (listp (cdr x)))
    (equal x y))
   (equal (length y) (if (listp y) 1 0))))

(prove-lemma last-repeat-with-inputs (rewrite)
  (implies
   (listp inputs-list)
   (equal
    (last (repeat-with-inputs state inputs-list loclist))
    (extract-locs loclist
                  (update-state-with-inputs (last inputs-list)
                                            state)))))
```

```
(prove-lemma equal-microcycles-constant-0 (rewrite)
  (equal (equal (microcycles-constant s m) 0) f))

(prove-lemma equal-init-time-0 (rewrite)
  (equal (equal (init-time s m) 0) f))

(prove-lemma nth-fm9001-rt-history-repeat-helper (rewrite)
  (equal
   (nth n (fm9001-rt-history-repeat-helper state mat inputs-list loclist))
   (if (and
        (lessp n (length inputs-list))
        (not (equal (fm9001-rt state (firstn (add1 n) inputs-list) mat) 0)))
       (extract-locs loclist (fm9001-rt state (firstn (add1 n) inputs-list) mat))
     0))
  ((induct (fm9001-rt-history-repeat-helper-induct n state mat inputs-list))
   (expand (fm9001-rt state (list (car inputs-list)) mat)
           (fm9001-rt state (firstn (add1 n) inputs-list) mat)
           (fm9001-rt-history-repeat-helper state mat inputs-list loclist))
   (disable-theory t)
   (enable-theory ground-zero naturals)
   (enable car-cons cdr-cons nth-too-big last equal-init-time-0 equal-microcycles-constant-0
           open-fm9001-rt nth-cons-append-hack firstn-1 firstn-2 length-bottom last
           update-state-with-inputs equal-update-state-with-inputs-0 nth-length-as-last
           length-append last-repeat-with-inputs nth-repeat-with-inputs last-firstn
           open-nthcdr nthcdr *1*read-write-statep *1*map-down lessp-1-microcycles-constant
           nth-firstn length-firstn nth-cons-firstn-hack lessp-0-init-time firstn-too-big
           our-equal-difference-0 *1*nat-to-v car-append car-firstn car-nthcdr nth-nthcdr
           plus-0 plus-add1 nthcdr-nlistp equal-length-small lessp-x-1
           firstn-open-on-length-1 reg-size length-repeat-with-inputs length-nthcdr
           firstn-length-better listp-nthcdr nthcdr-firstn cdr-append listp-append
           length-cons firstn-append nthcdr-append lessp-difference=0 append-nlistp
           listp-repeat-with-inputs listp-firstn nth-make-properp plus-difference-hack
           make-properp length-nlistp-cdr sub1-add1 listp-firstn length-firstn1
           difference-x-1 length-cdr-rewrite nth-repeat-with-inputs last-append
           nth-append-better zerop nth-cdr-make-properp-hack nth lessp-total-time-init-time
           nth-means-last-when repeat-with-inputs car-repeat-with-inputs properp-append-nil
           properp-cons properp open-nth equal-sub1-init-time-0-hack)))

(defn no-zeros-in-listp (l)
  (if (listp l)
      (and
       (not (equal (car l) 0))
       (no-zeros-in-listp (cdr l)))
    t))

(prove-lemma no-zeros-in-listp-append (rewrite)
  (equal
   (no-zeros-in-listp (append x y))
   (and
    (no-zeros-in-listp x)
    (no-zeros-in-listp y))))

(prove-lemma no-zeros-in-listp-repeat-with-inputs
             (rewrite)
             (implies (not (equal s 0))
                      (no-zeros-in-listp (repeat-with-inputs s l n))))

(prove-lemma nth-fm9001-rt-history-repeat-no-zeros (rewrite)
  (no-zeros-in-listp (fm9001-rt-history-repeat-helper state mat inputs-list loclist)))

(prove-lemma nth-fm9001-rt-history-no-zeros (rewrite)
```

```
  (no-zeros-in-listp (fm9001-rt-history-helper state mat inputs-list loclist n)))

(prove-lemma firstn-as-nth (rewrite)
  (implies
   (and
   (equal (firstn (sub1 k) l1) (firstn (sub1 k) l2))
   (not (zerop k))
   (no-zeros-in-listp l1)
   (no-zeros-in-listp l2))
   (equal
    (equal (firstn k l1) (firstn k l2))
    (equal (nth (sub1 k) l1) (nth (sub1 k) l2))))
  ((enable firstn)))

(prove-lemma fm9001-rt-repeat-helper-equiv-helper nil
  (equal
   (firstn k (fm9001-rt-history-helper state mat inputs-list loclist 0))
   (firstn k (fm9001-rt-history-repeat-helper state mat inputs-list loclist)))
  ((induct (plus k n))
   (disable-theory t)
   (enable-theory ground-zero naturals)
   (enable firstn-as-nth nth-fm9001-rt-history-helper nth-fm9001-rt-history-repeat-helper
           firstn nth-fm9001-rt-history-repeat-no-zeros nth-fm9001-rt-history-no-zeros)))

(prove-lemma properp-fm9001-history-repeat-helper (rewrite)
  (properp (fm9001-rt-history-repeat-helper s m i l)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Finally, the two FM9001 models are equivalent
;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(prove-lemma fm9001-rt-repeat-helper-equiv (rewrite)
  (equal
   (fm9001-rt-history-helper state mat inputs-list loclist 0)
   (fm9001-rt-history-repeat-helper state mat inputs-list loclist))
  ((use (fm9001-rt-repeat-helper-equiv-helper
         (k (if (lessp
                 (length (fm9001-rt-history-helper state mat inputs-list loclist 0))
                 (length (fm9001-rt-history-repeat-helper state mat inputs-list loclist)))
                (length (fm9001-rt-history-repeat-helper state mat inputs-list loclist))
              (length (fm9001-rt-history-helper state mat inputs-list loclist 0))))))
   (disable fm9001-rt-history-repeat-helper fm9001-rt-history-helper)))

(defn rembinding (k alist)
  (if (listp alist)
      (if (equal k (caar alist))
          (rembinding k (cdr alist))
        (cons (car alist) (rembinding k (cdr alist))))
    alist))

(defn rembinding-history (k h)
  (if (listp h)
      (cons
       (rembinding k (car h))
       (rembinding-history k (cdr h)))
    h))

(defn remove-if-cadr (k list)
  (if (listp list)
      (if (equal k (cadar list))
          (remove-if-cadr k (cdr list))
        (cons (car list) (remove-if-cadr k (cdr list))))
    list))
```

```
(prove-lemma rembinding-history-append (rewrite)
  (equal
   (rembinding-history k (append h1 h2))
   (append (rembinding-history k h1) (rembinding-history k h2))))

(prove-lemma rembinding-extract-locs (rewrite)
  (equal
   (rembinding k (extract-locs l s))
   (extract-locs (remove-if-cadr k l) s)))

(prove-lemma rembinding-history-repeat-with-inputs (rewrite)
  (equal
   (rembinding-history k (repeat-with-inputs s i l))
   (repeat-with-inputs s i (remove-if-cadr k l))))

(prove-lemma rembinding-history-fm9001-rt-history-repeat-helper (rewrite)
  (equal
   (rembinding-history k (fm9001-rt-history-repeat-helper s m i l))
   (fm9001-rt-history-repeat-helper s m i (remove-if-cadr k l))))


;;; Satisfiesp unaffected by rembinding-history

(defn variable-in-termp (v term)
  (if (listp term)

  (if (equal (car term) 'equal)
      (or
       (variable-in-termp v (cadr term))
       (variable-in-termp v (caddr term)))

  (if (equal (car term) 'lessp)
      (or
       (variable-in-termp v (cadr term))
       (variable-in-termp v (caddr term)))

  (if (equal (car term) 'not)
      (variable-in-termp v (cadr term))

  (if (equal (car term) 'plus)
      (or
       (variable-in-termp v (cadr term))
       (variable-in-termp v (caddr term)))

  (if (equal (car term) 'remainder)
      (or
       (variable-in-termp v (cadr term))
       (variable-in-termp v (caddr term)))

  (if (equal (car term) 'difference)
      (or
       (variable-in-termp v (cadr term))
       (variable-in-termp v (caddr term)))

  (if (equal (car term) 'if)
      (or
       (variable-in-termp v (cadr term))
       (variable-in-termp v (caddr term))
       (variable-in-termp v (cadddr term)))

  (if (equal (car term) 'and)
```

```
      (or
       (variable-in-termp v (cadr term))
       (variable-in-termp v (caddr term)))

  (if (equal (car term) 'or)
      (or
       (variable-in-termp v (cadr term))
       (variable-in-termp v (caddr term)))

  (if (equal (car term) 'old)
      (equal (cadr term) v)

  (if (equal (car term) 'c)
      f

  (if (equal (car term) 'quote) f

    f))))))))))))

  (if (numberp term) f

    (equal term v))))

(defn variable-in-translist (v translist)
  (if (listp translist)
      (or
       (variable-in-termp v (trans-pred (car translist)))
       (variable-in-translist v (cdr translist)))
    f))

(defn variable-in-state-listp (v state-list)
  (if (listp state-list)
      (or
       (variable-in-termp v (state-pred (car state-list)))
       (variable-in-translist v (state-trans-list (car state-list)))
       (variable-in-state-listp v (cdr state-list)))
    f))

(prove-lemma variable-in-translist-assoc (rewrite)
  (implies
   (not (variable-in-state-listp v b-states))
   (not (variable-in-translist v (caddr (assoc x b-states))))))

(prove-lemma variable-in-pred-assoc (rewrite)
  (implies
   (not (variable-in-state-listp v b-states))
   (not (variable-in-termp v (cadr (assoc x b-states))))))

(prove-lemma assoc-rembinding (rewrite)
  (equal
   (assoc x (rembinding y a))
   (if (equal x y) f (assoc x a))))

(prove-lemma my-eval-rembinding1 (rewrite)
   (implies
    (and
     (not (variable-in-termp v term))
     (not (listp v)))
    (equal (my-eval term (rembinding v a1) a2) (my-eval term a1 a2))))

(prove-lemma my-eval-rembinding2 (rewrite)
   (implies
```

```
    (and
     (not (variable-in-termp v term))
     (not (listp v)))
    (equal (my-eval term a1 (rembinding v a2)) (my-eval term a1 a2))))

(defn no-fs-in-listp (list)
  (if (listp list)
      (and
       (not (equal (car list) f))
       (no-fs-in-listp (cdr list)))
    t))

(prove-lemma assoc-k-append-better (rewrite)
  (equal
   (assoc k (append x y))
   (if (and (equal k 0) (member f x))
       (assoc k x)
     (if (assoc k x) (assoc k x) (assoc k y))))
  ((induct (append x y))))

(prove-lemma my-eval-rembinding-hack1 (rewrite)
   (implies
    (and
     (not (variable-in-termp v term))
     (not (listp v))
     (not (equal v 0)))
    (equal (my-eval term (append z1 (rembinding v a1)) a2) (my-eval term (append z1 a1) a2))))

(prove-lemma my-eval-rembinding-hack2 (rewrite)
   (implies
    (and
     (not (variable-in-termp v term))
     (not (listp v))
     (not (equal v 0)))
    (equal (my-eval term a1 (append z2 (rembinding v a2))) (my-eval term a1 (append z2 a2)))))

(prove-lemma satisfiesp-helper-rembinding-history (rewrite)
  (implies
   (and
    (not (variable-in-state-listp v b-states))
    (not (variable-in-translist v trans))
    (not (variable-in-termp v pred))
    (not (listp v))
    (not (equal v 0)))
   (equal
    (satisfiesp-helper name pred trans (rembinding-history v h) counters b-states)
    (satisfiesp-helper name pred trans  h counters b-states))))

(prove-lemma satisfiesp-rembinding-history (rewrite)
  (implies
   (and
    (not (equal v 0))
    (not (listp v))
    (not (variable-in-state-listp v (behavior-state-list b))))
   (equal
    (satisfiesp (rembinding-history v h) b)
    (satisfiesp h b))))

(defn bound-in-assignment (a)
  (if (listp a)
      (cons (caar a) (bound-in-assignment (cdr a)))
    nil))
```

```
(defn bound-in-history (h)
  (if (listp h)
      (if (listp (cdr h))
          (intersection
           (bound-in-assignment (car h))
           (bound-in-history (cdr h)))
        (bound-in-assignment (car h)))
    nil))


(defn all-variables-in-term (term)
  (if (listp term)

  (if (equal (car term) 'equal)
      (union
       (all-variables-in-term (cadr term))
       (all-variables-in-term (caddr term)))

  (if (equal (car term) 'lessp)
      (union
       (all-variables-in-term (cadr term))
       (all-variables-in-term (caddr term)))

  (if (equal (car term) 'not)
      (all-variables-in-term (cadr term))

  (if (equal (car term) 'plus)
      (union
       (all-variables-in-term (cadr term))
       (all-variables-in-term (caddr term)))

  (if (equal (car term) 'remainder)
      (union
       (all-variables-in-term (cadr term))
       (all-variables-in-term (caddr term)))

  (if (equal (car term) 'difference)
      (union
       (all-variables-in-term (cadr term))
       (all-variables-in-term (caddr term)))

  (if (equal (car term) 'if)
      (union
       (all-variables-in-term (cadr term))
       (union
       (all-variables-in-term (caddr term))
       (all-variables-in-term (cadddr term))))

  (if (equal (car term) 'and)
      (union
       (all-variables-in-term (cadr term))
       (all-variables-in-term (caddr term)))

  (if (equal (car term) 'or)
      (union
       (all-variables-in-term (cadr term))
       (all-variables-in-term (caddr term)))

  (if (equal (car term) 'old)
      (list (cadr term))
```

```
  (if (equal (car term) 'c)
      nil

    nil)))))))))))

  (if (numberp term) nil

    (list term))))

(defn all-variables-in-translist (translist)
  (if (listp translist)
      (union
       (all-variables-in-term (trans-pred (car translist)))
       (all-variables-in-translist (cdr translist)))
    nil))

(defn all-variables-in-state-list (state-list)
  (if (listp state-list)
      (union
       (all-variables-in-term (state-pred (car state-list)))
       (union
        (all-variables-in-translist (state-trans-list (car state-list)))
        (all-variables-in-state-list (cdr state-list))))
    nil))

(prove-lemma member-union (rewrite)
  (equal
   (member x (union a b))
   (or (member x a) (member x b))))

(prove-lemma member-intersection (rewrite)
  (equal
   (member x (intersection a b))
   (and (member x a) (member x b)))
  ((enable intersection)))

(prove-lemma member-all-variables-in-state-list (rewrite)
  (equal
   (member x (all-variables-in-term term))
   (variable-in-termp x term)))

(defn extension-assignmentp (a1 a2)
  (if (listp a1)
      (and
       (equal (cadr (assoc (caar a1) a2)) (cadar a1))
       (extension-assignmentp (cdr a1) a2))
    t))

(defn extension-historyp (h1 h2)
  (if (listp h1)
      (and
       (listp h2)
       (extension-assignmentp (car h1) (car h2))
       (extension-historyp (cdr h1) (cdr h2)))
    (nlistp h2)))

(prove-lemma subset-union (rewrite)
  (equal
   (subset (union a b) c)
   (and
    (subset a c)
    (subset b c))))
```

```
(prove-lemma subset-intersection (rewrite)
  (equal
   (subset a (intersection b c))
   (and
    (subset a b)
    (subset a c)))
  ((enable intersection subset)))

(prove-lemma subset-union2 (rewrite)
  (implies
   (subset a x)
   (subset a (union x y)))
  ((enable subset)))

(prove-lemma subset-union3 (rewrite)
  (implies
   (subset a x)
   (subset a (union y x)))
  ((enable subset)))

(prove-lemma subset-all-variables-in-translist (rewrite)
  (subset (all-variables-in-translist (caddr (assoc x b-states)))
          (all-variables-in-state-list b-states)))

(prove-lemma subset-all-variables-in-term (rewrite)
  (subset (all-variables-in-term (cadr (assoc x b-states)))
          (all-variables-in-state-list b-states)))

(prove-lemma transitivity-of-subset (rewrite)
  (implies
   (and
    (subset b c)
    (subset a b))
   (subset a c))
  ((enable subset)))

(prove-lemma subset-all-variables-in-translist-assoc-hack (rewrite)
  (implies
   (subset (all-variables-in-state-list b-states) y)
   (subset (all-variables-in-translist (caddr (assoc x b-states))) y)))

(prove-lemma subset-all-variables-in-term-assoc-hack (rewrite)
  (implies
   (subset (all-variables-in-state-list b-states) y)
   (subset (all-variables-in-term (cadr (assoc x b-states))) y)))

(prove-lemma subset-set-union (rewrite)
  (equal
   (subset (set-union a b) c)
   (and
    (subset a c)
    (subset b c))))

(prove-lemma subset-counters-of-term-assoc-hack (rewrite)
  (implies
   (subset (counters-of-state-list b-states) y)
   (subset (counters-of-term (cadr (assoc x b-states))) y)))

(prove-lemma subset-counters-of-translist-assoc-hack (rewrite)
  (implies
   (subset (counters-of-state-list b-states) y)
```

```
   (subset (counters-of-translist (caddr (assoc x b-states))) y)))

(prove-lemma my-eval-extension-append1 (rewrite)
  (implies
   (and
    (extension-assignmentp e1 e2)
    (subset (all-variables-in-term pred) (bound-in-assignment e1))
    (subset (counters-of-term pred) (bound-in-assignment c)))
   (equal (my-eval pred (append c e2) y)
          (my-eval pred (append c e1) y))))

(prove-lemma my-eval-extension-append2 (rewrite)
  (implies
   (and
    (extension-assignmentp e1 e2)
    (subset (all-variables-in-term pred) (bound-in-assignment e1))
    (subset (counters-of-term pred) (bound-in-assignment c)))
   (equal (my-eval pred y (append c e2))
          (my-eval pred y (append c e1)))))

(prove-lemma bound-in-assignment-update-counter-alist (rewrite)
  (equal (bound-in-assignment (update-counter-alist c trans))
         (bound-in-assignment c)))

(prove-lemma extension-satisfiesp-helper (rewrite)
  (implies
   (and
    (subset (all-variables-in-state-list b-states) (bound-in-history h1))
    (subset (all-variables-in-translist trans) (bound-in-history h1))
    (subset (all-variables-in-term pred) (bound-in-history h1))
    (subset (counters-of-state-list b-states) (bound-in-assignment c))
    (subset (counters-of-translist trans) (bound-in-assignment c))
    (subset (counters-of-term pred) (bound-in-assignment c))
    (extension-historyp h1 h2))
   (equal
    (satisfiesp-helper name pred trans h2 c b-states)
    (satisfiesp-helper name pred trans h1 c b-states))))

(prove-lemma bound-in-assignment-make-list-alist (rewrite)
  (equal
   (bound-in-assignment (make-list-alist x))
   (make-properp x)))

(prove-lemma extension-satisfiesp (rewrite)
  (implies
   (and
    (extension-historyp h1 h2)
    (subset (all-variables-in-state-list (behavior-state-list b))
            (bound-in-history h1)))
   (equal (satisfiesp h2 b) (satisfiesp h1 b))))

; extract from an assignment a list of (name value) pairs.  The needed
; list of values is represented by a pair: (loc name).  The output
; association list associates the value of loc in the original
; assignment with the name name.

(defn extract-from-assignment (list a)
  (if (listp list)
      (cons
       (list (caar list) (cadr (assoc (cadar list) a)))
       (extract-from-assignment (cdr list) a))
    nil))
```

```
(defn extract-from-history (list h)
  (if (listp h)
      (cons
       (extract-from-assignment list (car h))
       (extract-from-history list (cdr h)))
    nil))

(defn list-of-cadrs (list)
  (if (listp list)
      (cons (cadar list) (list-of-cadrs (cdr list)))
    nil))

(defn list-of-cars (list)
  (if (listp list)
      (cons (caar list) (list-of-cars (cdr list)))
    nil))

(prove-lemma intersection-x-x (rewrite)
  (implies
   (subset x y)
   (equal (intersection x y) (make-properp x)))
  ((enable intersection)))

(prove-lemma bound-in-assignment-extract-locs (rewrite)
  (equal
   (bound-in-assignment (extract-locs l x))
   (list-of-cadrs l)))

(prove-lemma properp-list-of-cadrs (rewrite)
  (properp (list-of-cadrs x)))

(prove-lemma properp-list-of-cars (rewrite)
  (properp (list-of-cars x)))

(prove-lemma bound-in-history-repeat-with-inputs (rewrite)
  (equal
   (bound-in-history (repeat-with-inputs s i l))
   (if (listp i) (list-of-cadrs l) nil)))

(prove-lemma intersection-associative (rewrite)
  (equal
   (intersection (intersection a b) c)
   (intersection a (intersection b c)))
  ((enable intersection)))

(prove-lemma bound-in-history-append (rewrite)
  (equal (bound-in-history (append x y))
         (if (listp x)
             (if (listp y)
                 (intersection (bound-in-history x) (bound-in-history y))
               (bound-in-history x))
           (bound-in-history y))))

(prove-lemma bound-in-history-fm9001-rt-history-repeat-helper (rewrite)
  (equal
   (bound-in-history (fm9001-rt-history-repeat-helper s m i l))
   (if (and (listp i) (not (read-write-statep s))) (list-of-cadrs l) nil))
  ((disable read-write-statep)
   (induct (fm9001-rt-history-repeat-helper s m i l))))

(defn set-difference (s1 s2)
```

```
  (if (listp s1)
      (if (member (car s1) s2)
          (set-difference (cdr s1) s2)
        (cons (car s1) (set-difference (cdr s1) s2)))
    nil))

(prove-lemma subset-delete* (rewrite)
  (implies
   (subset x y)
   (subset (delete* a x) y))
  ((enable subset delete*)))

(prove-lemma extract-locs-nlistp (rewrite)
  (implies
   (nlistp l)
   (equal (extract-locs l s) nil)))

(prove-lemma subset-extract-locs (rewrite)
  (implies
   (subset x y)
   (subset (extract-locs x s) (extract-locs y s)))
  ((induct (subset x y))
   (disable-theory t)
   (enable extract-locs subset)
   (enable-theory naturals ground-zero)))

(defn repeat-with-inputs-step (phase state inputs-list loclist)
  (if (listp inputs-list)
      (cons
       (cons
        (list 'phase phase)
        (cons
         (list 'step (length inputs-list))
         (extract-locs loclist (update-state-with-inputs (car inputs-list) state))))
       (repeat-with-inputs-step phase state (cdr inputs-list) loclist))
    nil))

(defn fm9001-rt-history-repeat-step-helper (state mat inputs-list loclist)
  (let
      ((init-time (init-time state mat))
       (total-time (microcycles-constant state mat)))
    (if (read-write-statep state) nil
      (if (not (lessp init-time (length inputs-list)))
          (repeat-with-inputs-step 'a state inputs-list loclist)
        (let ((new-state
               (fm9001-step
                (update-state-with-inputs (nth (sub1 init-time) inputs-list) state)
                (nat-to-v 15 (reg-size)))))
          (if (not (lessp total-time (length inputs-list)))
              (append
               (repeat-with-inputs-step 'a state (firstn init-time inputs-list) loclist)
               (repeat-with-inputs-step 'b new-state (nthcdr init-time inputs-list) loclist))
            (append
             (repeat-with-inputs-step 'a state (firstn init-time inputs-list) loclist)
             (append
              (repeat-with-inputs-step 'b
               new-state (nthcdr init-time (firstn total-time inputs-list)) loclist)
              (fm9001-rt-history-repeat-step-helper
               new-state mat (nthcdr total-time inputs-list) loclist))))))))
  ((lessp (length inputs-list))))

(prove-lemma rembinding-history-repeat-with-inputs-step (rewrite)
```

```
  (equal
   (rembinding-history
    'step
    (rembinding-history
     'phase
     (repeat-with-inputs-step p s i l)))
   (repeat-with-inputs s i (remove-if-cadr 'step (remove-if-cadr 'phase l)))))

(prove-lemma rembinding-history-rt-history-repeat-step (rewrite)
  (equal
   (rembinding-history
    'step
    (rembinding-history
     'phase
     (fm9001-rt-history-repeat-step-helper s m i l)))
   (fm9001-rt-history-repeat-helper s m i (remove-if-cadr 'step (remove-if-cadr 'phase l))))
  ((disable read-write-statep)
   (induct (fm9001-rt-history-repeat-step-helper s m i l))))



;Some functions useful for creating FM9001 state constants

(defn repeat (n val)
  (if (zerop n) nil
    (cons val (repeat (sub1 n) val))))

;; create an empty memory
(defn empty-memory (addr-size)
  (list-to-mem nil addr-size (make-list 32 f)))

;; write to memory, expanding "stubbed" memory as needed
(defn write-mem1-ignore-stubs (v-addr mem value)
  (COND
   ((NLISTP V-ADDR)
    (IF (not (ROMP MEM)) (RAM VALUE) MEM))
   ((STUBP MEM)
    (if (car v-addr)
        (cons mem (write-mem1-ignore-stubs (cdr v-addr) mem value))
      (cons (write-mem1-ignore-stubs (cdr v-addr) mem value) mem)))
   ((CAR V-ADDR)
    (CONS (CAR MEM) (WRITE-MEM1-ignore-stubs (CDR V-ADDR) (CDR MEM) VALUE)))
   (T (CONS (WRITE-MEM1-ignore-stubs (CDR V-ADDR) (CAR MEM) VALUE) (CDR MEM)))))

(defn write-mem-ignore-stubs (v-addr mem value)
  (write-mem1-ignore-stubs (reverse v-addr) mem value))

;; write an array to memory, expanding "stubbed" memory as needed
(defn write-array-memory (list offset mem)
  (if (listp list)
      (write-array-memory
       (cdr list)
       (add1 offset)
       (write-mem-ignore-stubs (nat-to-v offset 32) mem (car list)))
    mem))



;The initial state of the FM9001

(defn mirror-prog ()
```

```
  (asm
   '((move t f r1 (pc+))
    #xC0000000
    (move t f r0 (r1))
    (move t f r1 (pc+))
    #xC0000001
    (move t f (r1) r0)
    (move t f pc (pc+))
    #x40000000)))

(defn mirror-initial-state ()
  (list
   (list (write-mem-ignore-stubs
          (nat-to-v 15 4)
          (list-to-mem (repeat 15 (nat-to-v 0 32)) 4 (repeat 32 f))
          (nat-to-v #x40000000 32))
         (list t f f f))
   (write-array-memory
    (repeat 2 (nat-to-v 0 32))
    #xc0000000
    (write-array-memory
     (mirror-prog) #x40000000
     (empty-memory 32)))))




(prove-lemma satisfiesp-rembinding-rembinding-hack nil
  (implies
   (and
    (not (equal v1 0))
    (not (equal v2 0))
    (not (listp v1))
    (not (listp v2))
    (not (variable-in-state-listp v1 (behavior-state-list b)))
    (not (variable-in-state-listp v2 (behavior-state-list b))))
   (equal
    (satisfiesp (rembinding-history v1 (rembinding-history v2 h)) b)
    (satisfiesp h b))))


;; get the first element of list that has x as a cadr.
(defn get-member-cadr (x list)
  (if (listp list)
      (if (equal (cadar list) x)
          (car list)
        (get-member-cadr x (cdr list)))
    f))

(prove-lemma car-fm9001-rt-history (rewrite)
  (equal
   (car (fm9001-rt-history init k in out))
   (if (and (listp in) (not (read-write-statep init)))
       (extract-locs out (fm9001-rt init (list (car in)) k))
     0))
  ((expand (fm9001-rt-history-helper init k in out 0))
   (disable fm9001-rt-repeat-helper-equiv)
   (enable firstn)))

(prove-lemma extension-historyp-cons (rewrite)
  (equal
```

```
   (extension-historyp x (cons a b))
   (and
    (listp x)
    (extension-assignmentp (car x) a)
    (extension-historyp (cdr x) b))))

(prove-lemma listp-extract-from-history (rewrite)
  (equal
   (listp (extract-from-history list h))
   (listp h)))

(prove-lemma fm9001-rt-simple-open (rewrite)
  (implies
   (not (lessp (init-time state mat) (length inputs-list)))
   (equal
    (fm9001-rt state inputs-list mat)
    (if (nlistp inputs-list) state
    (if (read-write-statep state) 0
    (update-state-with-inputs (last inputs-list) state))))))


(prove-lemma car-extract-from-history
             (rewrite)
             (equal (car (extract-from-history list h))
                    (if (listp h)
                        (extract-from-assignment list (car h))
                        0)))

(prove-lemma car-when-extension-hack (rewrite)
  (implies
   (and
    (extension-historyp x y)
    (equal (car x) (cons z '((output 0) (newsig 0)))))
   (equal (cadr (assoc 'output (car y))) 0)))

(prove-lemma extract-locs-silly-hack
 (rewrite)
 (equal
  (extract-locs
             '((3221225472 signal)
               (3221225473 output)
               ((0) newsig))
             (fm9001-rt (mirror-initial-state)
                        (list (car (extract-from-history '((3221225472 signal))
                                                         h)))
                        3))
  (cons (list 'signal
              (remainder (cadr (assoc 'signal (car h)))
                         (exp 2 32)))
        '((output 0) (newsig 0))))
 ((enable write-mem read-mem)))

(prove-lemma read-write-statep-mirror-initial-state (rewrite)
  (not (read-write-statep (mirror-initial-state))))

(prove-lemma behavior-state-list-mirror-program-spec (rewrite)
  (equal
   (behavior-state-list (mirror-program-spec))
   '((a0
      (lessp (c 7) 1000)
      ((a0 nil
           (and
```

```
            (equal newsig (old newsig))
            (equal output (old output))))
       (a1 ((c 8))
           (and (equal output (old output))
                (equal newsig (old signal))))))
     (a1
      (lessp (c 8) 1000)
      ((a0 ((c 7))
           (and
            (equal output (old newsig))
            (equal newsig (old newsig))))
       (a1 nil (and (equal output (old output))
                    (equal newsig (old newsig))))))))

(prove-lemma length-extract-from-history (rewrite)
  (equal (length (extract-from-history e h)) (length h)))

(prove-lemma listp-fm9001-rt-history-repeat-helper  (rewrite)
  (equal
   (listp (fm9001-rt-history-repeat-helper s m i l))
   (and (listp i) (not (read-write-statep s)))))

(prove-lemma car-fm9001-rt-history-repeat-helper (rewrite)
  (equal
   (car (fm9001-rt-history-repeat-helper state mat inputs-list loclist))
   (if (listp inputs-list)
       (if (read-write-statep state)
             0
         (extract-locs loclist (update-state-with-inputs (car inputs-list) state)))
     0))
  ((expand (fm9001-rt-history-repeat-helper state mat inputs-list loclist))
   (disable read-write-statep)))

(prove-lemma extract-locs-append (rewrite)
  (equal (extract-locs (append a b) x)
         (append (extract-locs a x) (extract-locs b x))))

(prove-lemma extension-assignmentp-append (rewrite)
  (equal
   (extension-assignmentp (append a b) x)
   (and
    (extension-assignmentp a x)
    (extension-assignmentp b x))))

(prove-lemma extension-historyp-repeat-with-inputs-append (rewrite)
  (equal
   (extension-historyp (repeat-with-inputs s i (append a b)) h)
   (and
    (extension-historyp (repeat-with-inputs s i a) h)
    (extension-historyp (repeat-with-inputs s i b) h)))
  ((induct (double-cdr-induction i h))))

(defn extension-history-repeat-induct (s m i h)
  (if (or (read-write-statep s) (not (lessp (microcycles-constant s m) (length i))))
      t
    (extension-history-repeat-induct
     (fm9001-step
      (update-state-with-inputs (nth (sub1 (init-time s m)) i) s)
      (nat-to-v 15 (reg-size)))
     m
     (nthcdr (microcycles-constant s m) i)
     (nthcdr (microcycles-constant s m) h)))
```

```
  ((lessp (length i))))

(prove-lemma extension-historyp-append (rewrite)
  (equal
   (extension-historyp (append a b) x)
   (and
    (extension-historyp a (firstn (length a) x))
    (extension-historyp b (nthcdr (length a) x))))
  ((enable firstn nthcdr)
   (induct (double-cdr-induction a x))))

(prove-lemma nthcdr-nthcdr (rewrite)
  (equal (nthcdr n1 (nthcdr n2 x))
         (nthcdr (plus n1 n2) x))
  ((enable nthcdr)))

(prove-lemma firstn-firstn (rewrite)
  (equal
   (firstn n1 (firstn n2 l))
   (if (lessp n1 n2) (firstn n1 l) (firstn n2 l)))
  ((enable firstn)))

(prove-lemma extension-history-repeat-append (rewrite)
  (equal
   (extension-historyp (fm9001-rt-history-repeat-helper s m i (append a b)) h)
   (and
    (extension-historyp (fm9001-rt-history-repeat-helper s m i a) h)
    (extension-historyp (fm9001-rt-history-repeat-helper s m i b) h)))
  ((induct (extension-history-repeat-induct s m i h))
   (disable read-write-statep open-nthcdr)
   (expand (fm9001-rt-history-repeat-helper s m i (append a b))
           (fm9001-rt-history-repeat-helper s m i a)
           (fm9001-rt-history-repeat-helper s m i b))))

(prove-lemma light-switch-helper-helper (rewrite)
  (implies
   (and
    (listp h)
    (extension-historyp
     (fm9001-rt-history (mirror-initial-state)
                                  3
                                  (extract-from-history '((3221225472 signal))
                                                        h)
                                  '((3221225472 signal)
                                    (3221225473 output)
                                    ((0) newsig)))
                h)
    (satisfiesp
           (fm9001-rt-history-repeat-helper
                                  (mirror-initial-state)
                                  3
                                  (extract-from-history '((3221225472 signal))
                                                        h)
                                  '((3221225472 signal)
                                    (3221225473 output)
                                    ((0) newsig)))
           (mirror-program-spec)))
   (satisfiesp h (mirror-program-spec)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable fm9001-rt-history bound-in-history-fm9001-rt-history-repeat-helper
           extension-satisfiesp listp-extract-from-history *1*subset
```

```
          read-write-statep-mirror-initial-state list-of-cadrs fm9001-rt-repeat-helper-equiv
          *1*all-variables-in-state-list *1*behavior-state-list mirror-program-spec)))


(defn zipper-lists (l1 l2)
  (if (and (listp l1) (listp l2))
      (cons
       (append (car l1) (car l2))
       (zipper-lists (cdr l1) (cdr l2)))
    nil))

(defn satisfiesp-zipper-list-induct (name pred trans x y counters b-states)
  (if (listp y)
      (if (listp (cdr y))
          (if (listp trans)
               (let
                   ((new-state (assoc (trans-to (car trans)) b-states))
                    (new-counters (update-counter-alist
                                   counters (trans-resets (car trans)))))
                 (and
                  (satisfiesp-zipper-list-induct
                   (state-name new-state) (state-pred new-state)
                   (state-trans-list new-state) (cdr x) (cdr y)
                   new-counters b-states)
                 (satisfiesp-zipper-list-induct name pred (cdr trans) x y counters b-states)))
            f)
        f)
    f)
  ((ord-lessp (cons (add1 (length y)) (length trans)))))


(prove-lemma car-zipper-lists (rewrite)
  (equal
   (car (zipper-lists a b))
   (if (and (listp a) (listp b))
       (append (car a) (car b))
     0)))

(prove-lemma listp-zipper-lists (rewrite)
  (equal
   (listp (zipper-lists a b))
   (and (listp a) (listp b))))

(defn all-bound-in-history (h)
  (if (listp h)
      (if (listp (cdr h))
          (union
           (bound-in-assignment (car h))
           (all-bound-in-history (cdr h)))
        (bound-in-assignment (car h)))
    nil))

(prove-lemma intersection-union-nil1 (rewrite)
  (equal
   (equal (intersection c (union a b)) nil)
   (and
    (equal (intersection c a) nil)
    (equal (intersection c b) nil)))
  ((enable intersection)))

(prove-lemma intersection-union-nil2 (rewrite)
```

```
  (equal
   (equal (intersection (union a b) c) nil)
   (and
    (equal (intersection a c) nil)
    (equal (intersection b c) nil)))
  ((enable intersection)))

(prove-lemma member-bound-in-assignment (rewrite)
  (implies
   (and
    (not (assoc x z))
    (equal z y)
    (not (equal x 0)))
   (not (member x (bound-in-assignment y)))))

(prove-lemma not-member-bound-in-assignment-means (rewrite)
  (implies
   (and
    (not (member pred (bound-in-assignment x)))
    (equal x y))
   (not (assoc pred y))))

(prove-lemma intersection-single-element (rewrite)
  (equal
   (intersection (list a) b)
   (if (member a b) (list a) nil))
  ((enable intersection)))

(prove-lemma my-eval-append-hack (rewrite)
  (implies
   (and
    (equal (intersection (all-variables-in-term pred) (bound-in-assignment x)) nil)
    (subset (counters-of-term pred) (bound-in-assignment c)))
   (equal
    (my-eval pred (append c (append x y)) z)
    (my-eval pred (append c y) z))))

(prove-lemma my-eval-append-hack2 (rewrite)
  (implies
   (and
    (equal (intersection (all-variables-in-term pred) (bound-in-assignment x)) nil)
    (subset (counters-of-term pred) (bound-in-assignment c)))
   (equal
    (my-eval pred z (append c (append x y)))
    (my-eval pred z (append c y)))))

(prove-lemma subset-delete*-special (rewrite)
  (implies
   (subset (delete* x a) b)
   (equal (subset a b) (or (not (member x a)) (member x b))))
  ((enable delete* subset)))

(prove-lemma intersection-subsetp-nil1 (rewrite)
  (implies
   (and
    (equal (intersection x y) nil)
    (subset z x))
   (equal (intersection z y) nil))
  ((enable intersection subset)))

(prove-lemma intersection-variables-in-translist-nil (rewrite)
  (implies
```

```
   (equal (intersection (all-variables-in-state-list b-states) x) nil)
   (equal (intersection (all-variables-in-translist (caddr (assoc a b-states))) x) nil)))

(prove-lemma intersection-variables-in-term-nil (rewrite)
  (implies
   (equal (intersection (all-variables-in-state-list b-states) x) nil)
   (equal (intersection (all-variables-in-term (cadr (assoc a b-states))) x) nil)))

(prove-lemma satisfiesp-helper-zipper-lists (rewrite)
  (implies
   (and
    (equal (length x) (length y))
    (equal (intersection (all-variables-in-state-list b-states)
                         (all-bound-in-history x))
           nil)
    (equal (intersection (all-variables-in-translist trans)
                         (all-bound-in-history x))
           nil)
    (equal (intersection (all-variables-in-term pred)
                         (all-bound-in-history x))
           nil)
    (subset (counters-of-state-list b-states) (bound-in-assignment c))
    (subset (counters-of-translist trans) (bound-in-assignment c))
    (subset (counters-of-term pred) (bound-in-assignment c)))
   (equal (satisfiesp-helper name pred trans (zipper-lists x y) c b-states)
          (satisfiesp-helper name pred trans y c b-states)))
   ((induct (satisfiesp-zipper-list-induct name pred trans x y c b-states))
    (expand (satisfiesp-helper name pred trans y c b-states))))

(prove-lemma satisfiesp-zipper-lists (rewrite)
  (implies
   (and
    (equal (length x) (length y))
    (equal (intersection (all-variables-in-state-list (behavior-state-list b))
                         (all-bound-in-history x))
           nil))
   (equal (satisfiesp (zipper-lists x y) b)
          (satisfiesp y b))))

(prove-lemma zipper-lists-repeat-with-inputs (rewrite)
  (equal
   (zipper-lists (repeat-with-inputs a b x) (repeat-with-inputs a b y))
   (repeat-with-inputs a b (append x y))))

(prove-lemma zipper-lists-append (rewrite)
   (equal
    (zipper-lists (append x a) y)
    (append
     (zipper-lists x y)
     (zipper-lists a (nthcdr (length x) y)))))

(prove-lemma length-zipper-lists (rewrite)
  (equal (length (zipper-lists a b))
         (if (lessp (length a) (length b))
             (length a) (length b))))

(prove-lemma zipper-lists-make-properp1 (rewrite)
   (equal
    (zipper-lists (make-properp x) y)
    (zipper-lists x y)))

(prove-lemma zipper-lists-make-properp2 (rewrite)
```

```
  (equal
   (zipper-lists x (make-properp y))
   (zipper-lists x y))
  ((induct (zipper-lists x y))))

(prove-lemma properp-zipper-lists (rewrite)
  (properp (zipper-lists a b)))

(prove-lemma cdr-firstn-cons (rewrite)
  (equal (cdr (firstn n (cons a l))) (if (zerop n) 0 (firstn (sub1 n) l)))
  ((enable firstn)))

(prove-lemma zipper-lists-smaller-x (rewrite)
  (implies
   (lessp (length x) (length y))
   (equal (zipper-lists x y) (zipper-lists x (firstn (length x) y))))
  ((induct (zipper-lists x y))
   (enable firstn)))

(prove-lemma zipper-lists-append-special (rewrite)
  (implies
   (equal (length x) (length y))
   (equal
    (zipper-lists x (append y z))
    (zipper-lists x y))))

(prove-lemma firstn-0 (rewrite)
  (equal (firstn 0 x) nil)
  ((enable firstn)))

(prove-lemma zipper-lists-fm9001-rt-history (rewrite)
  (equal
   (zipper-lists (fm9001-rt-history s mat i x) (fm9001-rt-history s mat i y))
   (fm9001-rt-history s mat i (append x y)))
  ((disable fm9001-step read-write-statep equal-append-as-firstn-nthcdr)))

(prove-lemma extension-historyp-means-same-length (rewrite)
  (implies
   (extension-historyp a b)
   (and
    (equal (equal (length a) (length b)) t)
    (equal (equal (length b) (length a)) t))))

(prove-lemma subset-means-union (rewrite)
  (implies
   (subset x y)
   (equal (union x y) y)))

(prove-lemma union-nil (rewrite)
  (equal (union x nil) (make-properp x)))

(prove-lemma properp-bound-in-assignment (rewrite)
  (properp (bound-in-assignment x)))

(prove-lemma properp-union (rewrite)
  (equal (properp (union a b)) (properp b)))

(prove-lemma properp-all-bound-in-history (rewrite)
  (properp (all-bound-in-history a)))

(prove-lemma union-union (rewrite)
  (equal
```

```
   (union (union a b) c)
   (union a (union b c))))

(prove-lemma all-bound-in-history-append (rewrite)
  (equal
   (all-bound-in-history (append x y))
   (make-properp (union (all-bound-in-history x) (all-bound-in-history y))))
  ((induct (append x y))
   (enable union)))

(prove-lemma all-bound-in-history-repeat-with-inputs (rewrite)
  (equal
   (all-bound-in-history (repeat-with-inputs s i l))
   (if (listp i) (list-of-cadrs l) nil)))

(prove-lemma all-bound-in-history-fm9001-rt-history-repeat-helper (rewrite)
  (equal (all-bound-in-history (fm9001-rt-history-repeat-helper s m i l))
         (if (and (listp i)
                  (not (read-write-statep s)))
             (list-of-cadrs l)
             nil))
  ((disable read-write-statep)
   (induct (fm9001-rt-history-repeat-helper s m i l))))

(prove-lemma all-bound-in-history-fm9001-rt-history (rewrite)
  (equal
   (all-bound-in-history (fm9001-rt-history s m i l))
   (if (and (listp i) (not (read-write-statep s))) (list-of-cadrs l) nil))
  ((disable read-write-statep)))

(prove-lemma equal-length-fm9001-rt-length-fm9001-rt-repeat-helper (rewrite)
  (equal
   (equal (length (fm9001-rt-history-repeat-helper s m i l1))
          (length (fm9001-rt-history-repeat-helper s m i l2)))
   t)
  ((induct (fm9001-rt-history-repeat-helper s m i l1))
   (disable read-write-statep fm9001-step)
   (expand (fm9001-rt-history-repeat-helper s m i l1)
           (fm9001-rt-history-repeat-helper s m i l2))))

(prove-lemma equal-length-fm9001-rt-length-fm9001-rt (rewrite)
  (equal
   (equal (length (fm9001-rt-history s m i l1)) (length (fm9001-rt-history s m i l2)))
   t))

(prove-lemma listp-fm9001-rt-history (rewrite)
  (equal
   (listp (fm9001-rt-history s m i l))
   (and (not (read-write-statep s)) (listp i))))

(prove-lemma car-extract-from-assignment (rewrite)
  (equal
   (car (extract-from-assignment list h))
   (if (listp list)
       (list (caar list) (cadr (assoc (cadar list) h)))
     0)))

(prove-lemma extract-from-assignment-zipper-lists (rewrite)
  (implies
   (and
    (equal (intersection (list-of-cadrs l) (bound-in-assignment a)) nil)
    (not (member 0 (list-of-cadrs l))))
```

```
  (equal
   (extract-from-assignment l (append a b))
   (extract-from-assignment l b)))
  ((enable intersection)))

(prove-lemma extract-from-history-zipper-lists (rewrite)
  (implies
   (and
    (equal (intersection (list-of-cadrs l) (all-bound-in-history a)) nil)
    (not (member 0 (list-of-cadrs l))))
   (equal
    (extract-from-history l (zipper-lists a b))
    (extract-from-history l (firstn (length a) b))))
  ((enable firstn)
   (induct (zipper-lists a b))))

(prove-lemma extract-from-history-make-properp (rewrite)
  (equal
   (extract-from-history l (make-properp h))
   (extract-from-history l h)))


(prove-lemma cadr-assoc-when-extension-helper nil
  (implies
   (and
    (member x l1)
    (extension-assignmentp (extract-locs l1 v) z))
   (equal (cadr (assoc (cadr x) z))
          (if (listp (car x))
              (v-to-nat (read-mem (nat-to-v (caar x) 4) (regs (car v))))
            (v-to-nat (read-mem (nat-to-v (car x) 32) (cadr v)))))))

(prove-lemma cadr-assoc-when-extension (rewrite)
  (implies
   (and
    (subset l2 l1)
    (member x l1)
    (extension-assignmentp (extract-locs l1 v) z))
   (equal (cadr (assoc (cadr x) z))
          (if (listp (car x))
              (v-to-nat (read-mem (nat-to-v (caar x) 4) (regs (car v))))
           (v-to-nat (read-mem (nat-to-v (car x) 32) (cadr v))))))
  ((use (cadr-assoc-when-extension-helper))))


(prove-lemma member-car-when-subset (rewrite)
  (implies
   (subset l1 l2)
   (equal (member (car l1) l2)
          (or (listp l1) (member 0 l2)))))


(prove-lemma extension-assignementp-extract-locs-subset1 (rewrite)
  (implies
   (and
    (subset l2 l1)
    (extension-assignmentp (extract-locs l1 v) z))
   (extension-assignmentp (extract-locs l2 v) z))
  ((induct (subset l2 l1))
   (disable read-mem)))
```

```
(prove-lemma extension-historyp-subset-repeat-with-inputs (rewrite)
  (implies
   (and
    (subset l2 l1)
    (extension-historyp (repeat-with-inputs s i l1) h))
   (extension-historyp (repeat-with-inputs s i l2) h))
  ((induct (double-cdr-induction i h))))

(defn fm9001-rt-history-repeat-helper-and-h-induct (state mat inputs-list h)
  (let
      ((init-time (init-time state mat))
       (total-time (microcycles-constant state mat)))
    (if (read-write-statep state) nil
      (if (not (lessp init-time (length inputs-list)))
          t
        (let ((new-state
               (fm9001-step
                (update-state-with-inputs (nth (sub1 init-time) inputs-list) state)
                (nat-to-v 15 (reg-size)))))
          (if (not (lessp total-time (length inputs-list)))
              t
            (fm9001-rt-history-repeat-helper-and-h-induct
             new-state mat (nthcdr total-time inputs-list)
             (nthcdr total-time h)))))))
  ((lessp (length inputs-list))))

(prove-lemma plus-difference-x-y-sub1-y (rewrite)
  (equal (plus (difference x y) (sub1 y))
         (if (lessp x y)
             (sub1 y)
           (if (zerop y) (fix x) (sub1 x)))))

; useful theorem for sorting loc lists
(prove-lemma extension-history-repeat-helper-subset-fm9001-helper (rewrite)
  (implies
   (and
    (extension-historyp (fm9001-rt-history-repeat-helper s m i l1) h)
    (subset l2 l1))
   (extension-historyp (fm9001-rt-history-repeat-helper s m i l2) h))
  ((disable fm9001-step read-write-statep)
   (expand (fm9001-rt-history-repeat-helper s m i l2)
           (fm9001-rt-history-repeat-helper s m i l1))
   (induct (fm9001-rt-history-repeat-helper-and-h-induct s m i h))))


(defn no-dup-cars (list)
  (if (listp list)
      (and
       (not (member (caar list) (list-of-cars (cdr list))))
       (no-dup-cars (cdr list)))
    t))

(defn no-dup-cadrs (list)
  (if (listp list)
      (and
       (not (member (cadar list) (list-of-cadrs (cdr list))))
       (no-dup-cadrs (cdr list)))
    t))

(prove-lemma extension-assignmentp-cancel (rewrite)
  (implies
   (no-dup-cars x)
```

```
   (extension-assignmentp x (append x y))))

(defn list-of-no-dup-cars (list)
  (if (listp list)
      (and
       (no-dup-cars (car list))
       (list-of-no-dup-cars (cdr list)))
    t))

(prove-lemma extension-historyp-zipper-lists-cancel (rewrite)
  (implies
   (list-of-no-dup-cars a)
  (equal
   (extension-historyp (zipper-lists a b) (zipper-lists a c))
   (extension-historyp (firstn (length a) b) (zipper-lists a c)))))

(prove-lemma intersection-nil (rewrite)
  (and
   (equal (intersection x nil) nil)
   (equal (intersection nil x) nil))
  ((enable intersection)))

(defn all-litatoms (list)
  (if (listp list)
      (and
       (litatom (car list))
       (all-litatoms (cdr list)))
    t))

(prove-lemma extension-assignmentp-append-unnecc (rewrite)
  (implies
   (and
    (equal (intersection (bound-in-assignment a) (bound-in-assignment x)) nil)
    (all-litatoms (bound-in-assignment a)))
   (equal (extension-assignmentp a (append x y))
          (extension-assignmentp a y)))
  ((enable intersection)))

(prove-lemma all-litatoms-union (rewrite)
  (equal
   (all-litatoms (union a b))
   (and (all-litatoms a) (all-litatoms b)))
  ((enable union)))

(defn triple-cdr-induction (a b c)
  (if (and (listp a) (listp b) (listp c))
      (triple-cdr-induction (cdr a) (cdr b) (cdr c))
    t))

(prove-lemma extension-historyp-zipper-lists-unnecc (rewrite)
  (implies
   (and
    (equal (intersection (all-bound-in-history a) (all-bound-in-history b)) nil)
    (all-litatoms (all-bound-in-history a)))
   (equal (extension-historyp a (zipper-lists b c))
          (extension-historyp a (firstn (length b) c))))
  ((induct (triple-cdr-induction a b c))))

(prove-lemma permutation-means-extension-fm9001-rt-history-same nil
  (implies
   (and
    (subset x y)
```

```
    (subset y x))
   (equal (extension-historyp (fm9001-rt-history s m i x) z)
          (extension-historyp (fm9001-rt-history s m i y) z))))

(prove-lemma member-set-difference (rewrite)
  (equal
   (member x (set-difference a b))
   (and (member x a) (not (member x b))))
  ((enable set-difference)))

(prove-lemma subset-append-difference-hack (rewrite)
  (equal (subset x (append y (set-difference z y)))
         (subset x (append y z)))
  ((enable subset set-difference)))

(prove-lemma subset-set-difference (rewrite)
  (implies
   (subset a b)
   (subset (set-difference a c) b)))

(prove-lemma subset-append2 (rewrite)
  (implies
   (and (subset a c) (subset b c))
   (subset (append a b) c))
  ((enable subset)))

;;; reverse of normal version!
(prove-lemma zipper-lists-fm9001-rt-history-reverse (rewrite)
  (equal
   (fm9001-rt-history s mat i (append x y))
   (zipper-lists (fm9001-rt-history s mat i x) (fm9001-rt-history s mat i y)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable zipper-lists-fm9001-rt-history)))

(disable zipper-lists-fm9001-rt-history-reverse)

(prove-lemma list-of-cars-extract-locs (rewrite)
  (equal
   (list-of-cars (extract-locs a b))
   (list-of-cadrs a)))

(prove-lemma no-dups-cars-extract-locs (rewrite)
  (equal
   (no-dup-cars (extract-locs l s))
   (no-dup-cadrs l)))

(prove-lemma list-of-no-dups-cars-repeat-with-inputs (rewrite)
  (equal
   (list-of-no-dup-cars (repeat-with-inputs s i l))
   (or (nlistp i) (no-dup-cadrs l)))
  ((induct (repeat-with-inputs s i l))))

(prove-lemma list-of-no-dup-cars-append (rewrite)
  (equal
   (list-of-no-dup-cars (append x y))
   (and
    (list-of-no-dup-cars x)
    (list-of-no-dup-cars y))))

(prove-lemma list-of-no-dups-cars-fm9001-rt-history-repeat-helper (rewrite)
  (equal
```

```
   (list-of-no-dup-cars (fm9001-rt-history-repeat-helper s m i l))
   (or (not (listp i)) (read-write-statep s)
       (no-dup-cadrs l)))
  ((induct (fm9001-rt-history-repeat-helper s m i l))
   (disable read-write-statep fm9001-step)))

(prove-lemma list-of-no-dups-cars-fm9001-rt-history (rewrite)
  (equal
   (list-of-no-dup-cars (fm9001-rt-history s m i l))
   (or (not (listp i)) (read-write-statep s) (no-dup-cadrs l))))


(prove-lemma lessp-length-fm9001-rt-length-fm9001-rt (rewrite)
  (equal
   (lessp (length (fm9001-rt-history s m i l1)) (length (fm9001-rt-history s m i l2)))
   f)
  ((use (equal-length-fm9001-rt-length-fm9001-rt))
   (disable-theory t) (enable-theory ground-zero)))

(prove-lemma extension-historyp-make-properp (rewrite)
  (and
   (equal (extension-historyp (make-properp x) y)
          (extension-historyp x y))
   (equal (extension-historyp x (make-properp y))
          (extension-historyp x y))))

(prove-lemma member-means-member-cadrs (rewrite)
  (implies
   (member a l)
   (member (cadr a) (list-of-cadrs l))))

(prove-lemma intersection-cons2 (rewrite)
  (implies
   (not (member b a))
   (equal (intersection a (cons b c))
          (intersection a c)))
  ((enable intersection)))

(prove-lemma member-list-of-cadrs (rewrite)
  (implies
   (and
    (no-dup-cadrs l)
    (member x l))
  (equal (member (cadr x) (list-of-cadrs (set-difference l d)))
         (not (member x d))))
  ((induct (member x l))))

(prove-lemma member-subset-transitive (rewrite)
  (implies
   (and
    (member a x)
    (subset x y))
   (member a y)))

(prove-lemma subset-set-difference-set-difference (rewrite)
  (implies
   (subset b c)
   (subset (set-difference a c) (set-difference a b))))

(prove-lemma subset-list-of-cadrs (rewrite)
  (implies
   (subset a b)
```

```
   (subset (list-of-cadrs a) (list-of-cadrs b))))

(prove-lemma intersection-list-of-cadrs-set-difference (rewrite)
  (implies
   (and
    (subset l2 l)
    (no-dup-cadrs l2)
    (no-dup-cadrs l))
   (equal (intersection (list-of-cadrs (set-difference l l2)) (list-of-cadrs l2)) nil))
  ((enable set-difference subset intersection)))

(prove-lemma all-litatoms-list-of-cadrs-set-difference (rewrite)
  (implies
   (all-litatoms (list-of-cadrs l))
   (all-litatoms (list-of-cadrs (set-difference l l2)))))

(prove-lemma extension-historyp-extract-from-history-ziplist-helper nil
  (implies
   (and
    (no-dup-cadrs l)
    (no-dup-cadrs l2)
    (all-litatoms (list-of-cadrs l))
    (listp i)
    (equal (length (fm9001-rt-history s m i (set-difference l l2))) (length h))
    (equal (length (fm9001-rt-history s m i l2)) (length h))
    (not (read-write-statep s))
    (subset l2 l))
   (equal
    (extension-historyp
     (fm9001-rt-history s m i l)
     (zipper-lists (fm9001-rt-history s m i l2) h))
    (extension-historyp (fm9001-rt-history s m i (set-difference l l2)) h)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable subset-append-difference-hack subset-x-x subset-append subset-append2
           subset-set-difference intersection-nil all-litatoms-list-of-cadrs-set-difference
           intersection-list-of-cadrs-set-difference firstn-too-big
           lessp-length-fm9001-rt-length-fm9001-rt extension-historyp-zipper-lists-unnecc
           all-bound-in-history-fm9001-rt-history extension-historyp-make-properp
           list-of-no-dups-cars-fm9001-rt-history extension-historyp-zipper-lists-cancel
           zipper-lists-fm9001-rt-history-reverse)
   (use (permutation-means-extension-fm9001-rt-history-same
         (x l)
         (z (zipper-lists (fm9001-rt-history s m i l2) h))
         (y (append l2 (set-difference l l2)))))))

(prove-lemma nlistp-fm9001-rt-history (rewrite)
  (implies
   (not (listp i))
   (equal (fm9001-rt-history-helper s m i l n) nil)))

(prove-lemma read-write-statep-fm9001-rt-history (rewrite)
  (implies
   (read-write-statep s)
   (equal (fm9001-rt-history s m i l) nil)))

(prove-lemma extension-historyp-extract-from-history-ziplist (rewrite)
  (implies
   (and
    (no-dup-cadrs l)
    (no-dup-cadrs l2)
    (all-litatoms (list-of-cadrs l))
```

```
    (listp i)
    (equal (length (fm9001-rt-history s m i (set-difference l l2))) (length h))
    (subset l2 l))
   (equal
    (extension-historyp
     (fm9001-rt-history s m i l)
     (zipper-lists (fm9001-rt-history s m i l2) h))
    (extension-historyp (fm9001-rt-history s m i (set-difference l l2)) h)))
  ((use (extension-historyp-extract-from-history-ziplist-helper)
        (equal-length-fm9001-rt-length-fm9001-rt (l1 (set-difference l l2))))
   (disable-theory t)
   (enable read-write-statep-fm9001-rt-history zipper-lists extension-historyp length)
   (enable-theory ground-zero)))


(prove-lemma satisfiesp-nlistp (rewrite)
  (implies
   (not (listp h))
   (satisfiesp h b)))


(prove-lemma equal-length-from-extension-historyp-fm9001 (rewrite)
  (implies
   (extension-historyp (fm9001-rt-history s m i l2) h)
   (equal
    (equal (length (fm9001-rt-history s m i l1)) (length h))
    t))
  ((use (extension-historyp-means-same-length
        (a (fm9001-rt-history s m i l2))
        (b h)))
   (disable-theory t)
   (induct (length h))  ; just to get it to throw away equality!
   (enable-theory ground-zero)
   (enable equal-length-fm9001-rt-length-fm9001-rt)))

(prove-lemma not-lessp-length-from-extension-historyp-fm9001 (rewrite)
  (implies
   (extension-historyp (fm9001-rt-history s m i l2) h)
   (equal
    (lessp (length (fm9001-rt-history s m i l1)) (length h))
    f))
  ((use (extension-historyp-means-same-length
        (a (fm9001-rt-history s m i l2))
        (b h)))
   (disable-theory t)
   (induct (length h))  ; just to get it to throw away equality !
   (enable lessp-length-fm9001-rt-length-fm9001-rt)
   (enable-theory ground-zero)))


(defn list-nat-to-v (list length)
  (if (listp list)
      (cons (nat-to-v (car list) length)
            (list-nat-to-v (cdr list) length))
    nil))

(defn member-v-iff (x list)
  (if (listp list)
      (or
       (v-iff x (car list))
       (member-v-iff x (cdr list)))
    f))
```

```
;; a few events inexplicibly not in memory.events.

(prove-lemma write-mem1-write-mem1-different (rewrite)
  (implies
   (and
    (equal (length v-addr1) (length v-addr2))
    (not (v-iff v-addr1 v-addr2)))
   (equal
    (write-mem1 v-addr1 (write-mem1 v-addr2 mem value2) value1)
    (write-mem1 v-addr2 (write-mem1 v-addr1 mem value1) value2)))
  ((enable write-mem1 v-iff)))

(prove-lemma write-mem1-write-mem1-same (rewrite)
  (implies
   (and
    (equal (length v-addr1) (length v-addr2))
    (v-iff v-addr1 v-addr2))
   (equal
    (write-mem1 v-addr1 (write-mem1 v-addr2 mem value2) value1)
    (write-mem1 v-addr1 mem value1)))
  ((enable write-mem1 v-iff)))

(prove-lemma write-mem-write-mem-same (rewrite)
  (implies
   (and
    (equal (length v-addr1) (length v-addr2))
    (v-iff v-addr1 v-addr2))
   (equal (write-mem v-addr1 (write-mem v-addr2 mem value2) value1)
          (write-mem v-addr1 mem value1)))
  ((enable write-mem)))


(prove-lemma write-mem-write-mem-different (rewrite)
  (implies
   (and
    (equal (length v-addr1) (length v-addr2))
    (not (v-iff v-addr1 v-addr2)))
   (equal (write-mem v-addr1 (write-mem v-addr2 mem value2) value1)
          (write-mem v-addr2 (write-mem v-addr1 mem value1) value2)))
  ((enable write-mem)
   (use (write-mem1-write-mem1-different (v-addr1 (reverse v-addr1))
                                         (v-addr2 (reverse v-addr2))))))

;; Nqthm does not use a lexicographic ordering to decide if it wants
;; to rewrite a permutative rule.  So, write-mem-write-mem-different
;; sometimes doesn't work the way it should.  So, we add this other
;; rule that sometimes will help.

(prove-lemma equal-write-mem-write-mem-different (rewrite)
  (implies
   (and
    (equal (length v-addr1) (length v-addr2))
    (not (v-iff v-addr1 v-addr2))
    (equal
     (equal (write-mem v-addr1 (write-mem v-addr2 mem value2) value1)
            (write-mem v-addr2 (write-mem v-addr1 mem value1) value2))
     t))

  ((enable write-mem)
   (use (write-mem1-write-mem1-different (v-addr1 (reverse v-addr1))
                                         (v-addr2 (reverse v-addr2))))))
```

```
(defn list-of-vs (list length)
  (if (listp list)
      (and
       (equal (length (car list)) length)
       (bvp (car list))
       (list-of-vs (cdr list) length))
    t))

(prove-lemma v-iff-commutative (rewrite)
  (implies
   (equal (length x) (length y))
   (equal (v-iff x y) (v-iff y x)))
  ((enable v-iff)))

(defn update-state-with-inputs-write-mem-induct (i b regs mem val)
  (if (listp i)
      (and
      (update-state-with-inputs-write-mem-induct
       (cdr i)
       b
       regs
       mem
       val)
      (update-state-with-inputs-write-mem-induct
       (cdr i)
       (nat-to-v (caar i) 32)
       regs
       (write-mem b mem val)
       (nat-to-v (cadar i) 32))
      (update-state-with-inputs-write-mem-induct
       (cdr i)
       (nat-to-v (caar i) 32)
       regs
       mem
       (nat-to-v (cadar i) 32)))
    t))

(prove-lemma update-state-with-inputs-write-mem (rewrite)
  (implies
   (equal (length b) 32)
   (equal
    (update-state-with-inputs i (list regs (write-mem b mem val)))
    (if (member-v-iff b (list-nat-to-v (list-of-cars i) 32))
        (update-state-with-inputs i (list regs mem))
      (list regs (write-mem b (cadr (update-state-with-inputs i (list regs mem))) val)))))
  ((induct (update-state-with-inputs-write-mem-induct i b regs mem val))
   (enable length nat-to-v length-nat-to-v v-iff v-iff-commutative
           equal-write-mem-write-mem-different length-nat-to-v write-mem-write-mem-same
           update-state-with-inputs list-of-cars list-nat-to-v member-v-iff
           write-mem-write-mem-different)
   (disable-theory t)
   (enable-theory ground-zero)))

(prove-lemma read-mem-write-mem-rewrite (rewrite)
  (implies
   (equal (length v-addr1) (length v-addr2))
   (equal (read-mem v-addr1 (write-mem v-addr2 mem value))
          (if (and (v-iff v-addr1 v-addr2)
                   (ramp-mem v-addr2 mem))
```

```
              value
            (read-mem v-addr1 mem))))
  ((enable read-mem write-mem ramp-mem)))

(prove-lemma read-mem-nlistp (rewrite)
  (implies
   (and (not (listp x)) (not (romp x)) (not (ramp x)) (not (stubp x)))
   (equal (read-mem a x) 0))
  ((enable read-mem read-mem1)))

(prove-lemma write-mem1-nlistp (rewrite)
  (implies
   (and (not (listp mem)) (not (ramp mem)))
   (equal (write-mem1 a mem val) mem))
  ((enable write-mem1)))

(prove-lemma write-mem-nlistp (rewrite)
  (implies
   (and (not (listp mem)) (not (ramp mem)))
   (equal (write-mem a mem val) mem))
  ((enable write-mem)))

(prove-lemma update-state-with-inputs-nlistp (rewrite)
  (implies
   (and (not (listp (cadr s))) (not (ramp (cadr s))))
   (equal (update-state-with-inputs i s)
          (if (listp i) (list (car s) (cadr s)) s))))

(prove-lemma read-mem-cadr-update-state-with-inputs-not (rewrite)
  (implies
   (and
    (not (member-v-iff a1 (list-nat-to-v (list-of-cars b) 32)))
    (equal (length a1) 32))
   (equal
    (read-mem a1 (cadr (update-state-with-inputs (extract-from-assignment b assignment) s)))
    (read-mem a1 (cadr s)))))

(defn variable-not-overflowed-in-historyp (v h)
  (if (listp h)
      (and
       (lessp (cadr (assoc v (car h))) (exp 2 32))
       (numberp (cadr (assoc v (car h))))
       (variable-not-overflowed-in-historyp v (cdr h)))
    t))

(prove-lemma list-of-cars-extract-from-assignment (rewrite)
  (equal
   (list-of-cars (extract-from-assignment x a))
   (list-of-cars x)))

(prove-lemma ramp-mem1-write-mem1 (rewrite)
  (equal
   (ramp-mem1 a1 (write-mem1 a2 mem v))
   (ramp-mem1 a1 mem))
  ((enable ramp-mem1 write-mem1)))

(prove-lemma ramp-mem-write-mem (rewrite)
  (equal
   (ramp-mem a (write-mem a2 mem v))
   (ramp-mem a mem))
  ((enable ramp-mem write-mem)))
```

```
(prove-lemma ramp-mem-update-state-with-inputs (rewrite)
  (equal
   (ramp-mem a (cadr (update-state-with-inputs i s)))
   (ramp-mem a (cadr s))))

(prove-lemma extension-historyp-repeat-with-inputs-copy-unnecc (rewrite)
  (implies
   (and
    (not (member-v-iff (nat-to-v (car a) 32) (list-nat-to-v (list-of-cars b) 32)))
    (variable-not-overflowed-in-historyp (cadr a) h)
    (ramp-mem (nat-to-v (car a) 32) (cadr s))
    (nlistp (car a)))
  (equal
   (extension-historyp
    (repeat-with-inputs s (extract-from-history (cons a b) h) (cons a c))
    h)
   (extension-historyp
    (repeat-with-inputs s (extract-from-history (cons a b) h) c)
    h)))
  ((induct (length h))))

(prove-lemma nthcdr-extract-from-history (rewrite)
  (implies
   (not (lessp (length h) n))
   (equal
    (nthcdr n (extract-from-history out h))
    (extract-from-history out (nthcdr n h))))
  ((enable nthcdr)
   (induct (nthcdr n h))))

(prove-lemma cdr-extract-from-history (rewrite)
  (equal
   (cdr (extract-from-history out h))
   (if (listp h) (extract-from-history out (cdr h)) 0)))

(prove-lemma variable-not-overflowed-in-nthcdr-history (rewrite)
  (implies
   (variable-not-overflowed-in-historyp x h)
   (variable-not-overflowed-in-historyp x (nthcdr n h)))
  ((enable nthcdr variable-not-overflowed-in-historyp)
   (disable-theory t)
   (enable-theory ground-zero)))

(prove-lemma variable-not-overflowed-in-cdr-history (rewrite)
  (implies
   (and
    (variable-not-overflowed-in-historyp x h1)
    (equal h h1))
   (variable-not-overflowed-in-historyp x (cdr h))))

(prove-lemma ramp-mem1-fm9001-step (rewrite)
  (equal
   (ramp-mem1 a (cadr (fm9001-step s pc)))
   (ramp-mem1 a (cadr s)))
  ((enable fm9001-step ramp-mem1 fm9001-fetch fm9001-operand-a fm9001-operand-b
          fm9001-alu-operation ramp-mem1-write-mem1 write-mem)
   (disable-theory t)
   (enable-theory ground-zero)))

(prove-lemma ramp-mem-fm9001-step (rewrite)
  (equal
   (ramp-mem a (cadr (fm9001-step s pc)))
```

```
   (ramp-mem a (cadr s)))
  ((enable ramp-mem)))

(prove-lemma extension-historyp-repeat-with-inputs-cons-simple (rewrite)
  (implies
   (and
    (extension-historyp (repeat-with-inputs s i x) h)
    (subset y x))
   (extension-historyp (repeat-with-inputs s i y) h)))

(prove-lemma firstn-repeat-with-inputs (rewrite)
  (equal
   (firstn n (repeat-with-inputs s i o))
   (repeat-with-inputs s (firstn n i) o))
  ((enable firstn)))

(prove-lemma firstn-extract-from-history (rewrite)
  (equal
   (firstn n (extract-from-history o h))
   (extract-from-history o (firstn n h)))
  ((enable firstn)))

(prove-lemma variable-not-overflowed-in-historyp-firstn (rewrite)
  (implies
   (variable-not-overflowed-in-historyp v h)
   (variable-not-overflowed-in-historyp v (firstn n h)))
  ((enable firstn)))

(prove-lemma cdr-firstn (rewrite)
  (equal (cdr (firstn n l))
         (if (and (listp l) (not (zerop n)))
             (firstn (sub1 n) (cdr l))
           0))
  ((enable firstn)))

(prove-lemma plus-difference-sub1-hack (rewrite)
  (equal (plus (difference x (sub1 y)) y)
         (if (lessp x (sub1 y))
             (fix y)
           (if (zerop y)
               (fix x)
             (add1 x)))))

(prove-lemma extension-history-fm9001-repeat-helper-copy-unnecc (rewrite)
  (implies
   (and
    (not (member-v-iff (nat-to-v (car a) 32) (list-nat-to-v (list-of-cars b) 32)))
    (variable-not-overflowed-in-historyp (cadr a) h)
    (ramp-mem (nat-to-v (car a) 32) (cadr s))
    (nlistp (car a))
    (equal i (extract-from-history (cons a b) h)))
   (equal
    (extension-historyp
     (fm9001-rt-history-repeat-helper s m i (cons a c))
     h)
    (extension-historyp
     (fm9001-rt-history-repeat-helper s m i c)
     h)))
  ((induct (fm9001-rt-history-repeat-helper-and-h-induct s m i h))
   (expand
    (fm9001-rt-history-repeat-helper
     s m (extract-from-history (cons a b) h) (cons a c))
```

```
    (fm9001-rt-history-repeat-helper
     s m (extract-from-history (cons a b) h) c))
   (disable read-write-statep fm9001-rt-history-repeat-helper repeat-with-inputs
            extract-from-history)))


(prove-lemma extension-history-fm9001-copy-unnecc (rewrite)
  (implies
   (and
    (not (member-v-iff (nat-to-v (car a) 32) (list-nat-to-v (list-of-cars b) 32)))
    (variable-not-overflowed-in-historyp (cadr a) h)
    (ramp-mem (nat-to-v (car a) 32) (cadr s))
    (nlistp (car a)))
   (equal
    (extension-historyp
     (fm9001-rt-history s m (extract-from-history (cons a b) h) (cons a c))
     h)
    (extension-historyp
     (fm9001-rt-history s m (extract-from-history (cons a b) h) c)
     h))))

(prove-lemma variable-not-overflowed-in-historyp-zipper-lists (rewrite)
  (implies
   (variable-not-overflowed-in-historyp v x)
   (equal
    (variable-not-overflowed-in-historyp v (zipper-lists y x))
    (variable-not-overflowed-in-historyp v (firstn (length x) y)))))

(prove-lemma extract-locs-base (rewrite)
  (implies
   (not (member v (list-of-cadrs l)))
   (equal (cadr (assoc v (extract-locs l s))) 0)))

(prove-lemma variable-not-overflowed-in-repeat-with-inputs-simple (rewrite)
  (implies
   (not (member v (list-of-cadrs l)))
   (variable-not-overflowed-in-historyp v (repeat-with-inputs s i l)))
  ((induct (repeat-with-inputs s i l))))

(prove-lemma variable-not-overflowed-in-historyp-append (rewrite)
  (equal
   (variable-not-overflowed-in-historyp v (append x y))
   (and
    (variable-not-overflowed-in-historyp v x)
    (variable-not-overflowed-in-historyp v y))))

(prove-lemma variable-not-overflowed-in-historyp-fm9001-rt-history-simple (rewrite)
  (implies
   (not (member v (list-of-cadrs l)))
   (variable-not-overflowed-in-historyp v (fm9001-rt-history s m i l)))
  ((disable read-write-statep)))

(prove-lemma location-ramp-1 (rewrite)
  (ramp-mem (nat-to-v 3221225472 32)
            (cadr (mirror-initial-state))))


;;;;;;
;; mirror-prog meets program spec
;;;;;;
```

```
;; initial time of an instruction in the mirror program (really this
;; is just a complicated calculation of a constant)
(defn i-m-helper (s loc mat)
  (if (zerop loc)
      (init-time s mat)
    (i-m-helper (fm9001-step s (nat-to-v 15 (reg-size))) (sub1 loc) mat)))

(defn i-m (loc mat)
  (i-m-helper (mirror-initial-state) loc mat))

;; total time of an instruction in the mirror program
(defn t-m-helper (s loc mat)
  (if (zerop loc)
      (microcycles-constant s mat)
    (t-m-helper (fm9001-step s (nat-to-v 15 (reg-size))) (sub1 loc) mat)))

(defn t-m (loc mat)
  (t-m-helper (mirror-initial-state) loc mat))

;; conclusion time of an instruction
(defn c-m (loc mat)
  (difference (t-m loc mat) (i-m loc mat)))

(defn mirror-fm9001-state-invariants (s)
  (let
      ((mem (cadr s))
       (regs (regs (car s))))
    (let
        ((pc (v-to-nat (read-mem (nat-to-v 15 4) regs)))
         (r1 (v-to-nat (read-mem (nat-to-v 1 4) regs))))

      (and
       (equal (read-mem (nat-to-v #x40000000 32) mem) (nth 0 (mirror-prog)))
       (equal (read-mem (nat-to-v #x40000001 32) mem) (nth 1 (mirror-prog)))
       (equal (read-mem (nat-to-v #x40000002 32) mem) (nth 2 (mirror-prog)))
       (equal (read-mem (nat-to-v #x40000003 32) mem) (nth 3 (mirror-prog)))
       (equal (read-mem (nat-to-v #x40000004 32) mem) (nth 4 (mirror-prog)))
       (equal (read-mem (nat-to-v #x40000005 32) mem) (nth 5 (mirror-prog)))
       (equal (read-mem (nat-to-v #x40000006 32) mem) (nth 6 (mirror-prog)))
       (equal (read-mem (nat-to-v #x40000007 32) mem) (nth 7 (mirror-prog)))

       (equal (length (read-mem (nat-to-v #xc0000000 32) mem)) 32)
       (ramp-mem (nat-to-v #xc0000000 32) mem)

       (ramp-mem (nat-to-v #xc0000001 32) mem)

       (ramp-mem (nat-to-v 0 4) regs)
       (ramp-mem (nat-to-v 1 4) regs)
       (ramp-mem (nat-to-v 15 4) regs)

       (equal (length (read-mem (nat-to-v 15 4) regs)) 32)

       (equal (length (read-mem (nat-to-v 1 4) regs)) 32)

       (equal (length (read-mem (nat-to-v 0 4) regs)) 32)


       (or (equal pc #x40000000) (equal pc #x40000002) (equal pc #x40000003)
           (equal pc #x40000005) (equal pc #x40000006))))))

(defn mirror-prog-invariant-one-place (pc phase step r1 mat c7 c8)
  (and
```

```
   (numberp c7)

   (numberp c8)

   (or (equal phase 'a) (equal phase 'b))

   (or (equal pc #x40000000)
       (equal pc #x40000002)
       (equal pc #x40000003)
       (equal pc #x40000005)
       (equal pc #x40000006))

   (implies (and (equal pc #x40000000) (equal phase 'a))
            (not (lessp (plus (c-m 3 mat) (t-m 4 mat) (i-m 0 mat))
                        (plus c7 step))))

   (implies (and (equal pc #x40000002) (equal phase 'b))
            (and (not (lessp (plus (c-m 3 mat) (t-m 4 mat) (t-m 0 mat))
                             (plus c7 step)))
                 (equal r1 #xc0000000)))

   (implies (and (equal pc #x40000002) (equal phase 'a))
            (and
            (not (lessp (plus (c-m 3 mat) (t-m 4 mat) (t-m 0 mat) (i-m 1 mat))
                        (plus c7 step)))
            (equal r1 #xc0000000)))


   (implies (and (equal pc #x40000003) (equal phase 'b))
            (not (lessp (c-m 1 mat)
                        (plus c8 step))))

   (implies (and (equal pc #x40000003) (equal phase 'a))
            (not (lessp (plus (c-m 1 mat) (i-m 2 mat))
                        (plus c8 step))))

   (implies (and (equal pc #x40000005) (equal phase 'b))
            (and
             (not (lessp (plus (c-m 1 mat) (t-m 2 mat))
                         (plus c8 step)))
             (equal r1 #xc0000001)))

   (implies (and (equal pc #x40000005) (equal phase 'a))
            (and
             (not (lessp (plus (c-m 1 mat) (t-m 2 mat) (i-m 3 mat))
                         (plus c8 step)))
             (equal r1 #xc0000001)))

   (implies (and (equal pc #x40000006) (equal phase 'b))
            (not (lessp (c-m 3 mat) (plus step c7))))

   (implies (and (equal pc #x40000006) (equal phase 'a))
            (not (lessp (plus (c-m 3 mat) (i-m 4 mat)) (plus step c7))))

   (implies (and (equal pc #x40000000) (equal phase 'b))
            (not (lessp (plus (c-m 3 mat) (t-m 4 mat)) (plus step c7))))))


(defn mirror-prog-invariant-two-place
  (state-name1 state-name2 newsig1 newsig2 output1 output2 signal1 c71 c72 c81 c82)
  (if (equal state-name1 'a0)
```

```
      (if (equal state-name2 'a0)
          (and
           (equal c72 (add1 c71))
           (equal c82 (add1 c81))
           (equal newsig1 newsig2)
           (equal output1 output2))
        (and
         (equal c72 (add1 c71))
         (equal c82 0)
         (equal output1 output2)
         (equal newsig2 signal1)))
    (if (equal state-name2 'a0)
        (and
         (equal c72 0)
         (equal c82 (add1 c81))
         (equal output2 newsig1)
         (equal newsig1 newsig2))
      (and
       (equal c72 (add1 c71))
       (equal c82 (add1 c81))
       (equal output2 output1)
       (equal newsig1 newsig2)))))

(disable t-m)
(disable i-m)


(prove-lemma i-m-open (rewrite)
  (and
   (equal (i-m 0 3) 11)
   (equal (i-m 1 3) 11)
   (equal (i-m 2 3) 11)
   (equal (i-m 3 3) 13)
   (equal (i-m 4 3) 11)))

(prove-lemma t-m-open (rewrite)
  (and
   (equal (t-m 0 3) 16)
   (equal (t-m 1 3) 16)
   (equal (t-m 2 3) 16)
   (equal (t-m 3 3) 16)
   (equal (t-m 4 3) 16)))

(prove-lemma c-m-open (rewrite)
  (and
   (equal (c-m 0 3) 5)
   (equal (c-m 1 3) 5)
   (equal (c-m 2 3) 5)
   (equal (c-m 3 3) 3)
   (equal (c-m 4 3) 5)))

(defn mirror-program-map (pc)
  (if (or (equal pc #x40000003) (equal pc #x40000005))
      'a1
    'a0))

(defn mirror-program-map-path (h)
  (if (listp h)
      (cons (mirror-program-map (cadr (assoc 'pc (car h))))
            (mirror-program-map-path (cdr h)))
    nil))
```

```
(defn mirror-prog-invariants-hold (h c7 c8 mat)
  (let
      ((pc1 (cadr (assoc 'pc (car h))))
       (phase1 (cadr (assoc 'phase (car h))))
       (step1 (cadr (assoc 'step (car h))))
       (address1 (cadr (assoc 'address (car h))))
       (output1 (cadr (assoc 'output (car h))))
       (newsig1 (cadr (assoc 'newsig (car h))))
       (signal1 (cadr (assoc 'signal (car h))))
       (pc2 (cadr (assoc 'pc (cadr h))))
       (phase2 (cadr (assoc 'phase (cadr h))))
       (step2 (cadr (assoc 'step (cadr h))))
       (address2 (cadr (assoc 'address (cadr h))))
       (newsig2 (cadr (assoc 'newsig (cadr h))))
       (output2 (cadr (assoc 'output (cadr h)))))
    (let ((state-name1 (mirror-program-map pc1))
          (state-name2 (mirror-program-map pc2)))
      (let ((newc7 (if (and (equal state-name1 'a1) (equal state-name2 'a0)) 0 (add1 c7)))
            (newc8 (if (and (equal state-name1 'a0) (equal state-name2 'a1)) 0 (add1 c8))))
        (if (listp h)
            (and
             (mirror-prog-invariant-one-place pc1 phase1 step1 address1 mat c7 c8)
             (if (listp (cdr h))
                 (and
                  (mirror-prog-invariant-two-place
                   state-name1 state-name2
                   newsig1 newsig2 output1 output2 signal1
                   c7 newc7 c8 newc8)
                  (mirror-prog-invariants-hold (cdr h) newc7 newc8 mat))
               t))
          t)))))


(prove-lemma assoc-update-counter-alist-casesplit (rewrite)
  (implies
   (not (equal a 0))
  (equal
   (assoc a (update-counter-alist c l))
   (if (assoc a c)
       (if (member a l)
           (list a 0)
         (list a (add1 (cadr (assoc a c)))))
     f)))
  ((induct (assoc a c))))


(prove-lemma mirror-program-map-normalize (rewrite)
  (equal
   (equal (mirror-program-map x) 'a0)
   (not (equal (mirror-program-map x) 'a1))))

(prove-lemma satisfiesp-with-path-helper-cons (rewrite)
  (equal
   (satisfiesp-with-path-helper pred trans (cons x y) counters b-states path)
   (if (listp y)
       (if (listp trans)
           (and
            (assoc (car path) b-states)
            (my-eval pred (append counters x) nil)
            (let
                ((new-state (assoc (trans-to (car trans)) b-states))
                 (new-counters (update-counter-alist
```

```
                              counters (trans-resets (car trans)))))
              (if (equal (trans-to (car trans)) (cadr path))
                  (and
                   (my-eval
                    (trans-pred (car trans))
                    (append new-counters (car y))
                       (append counters x))
                   (satisfiesp-with-path-helper
                    (state-pred new-state) (state-trans-list new-state)
                    y new-counters b-states (cdr path)))
                (satisfiesp-with-path-helper pred (cdr trans) (cons x y) counters
                                             b-states path))))
         f)
     (and
      (assoc (car path) b-states)
      (my-eval pred (append counters x) nil)))))


(prove-lemma lessp-plus-hack-rewrite (rewrite)
  (implies
   (and
    (not (lessp n1 (plus x y)))
    (lessp n1 n2))
   (equal (lessp x n2) t)))

;; repeat so that it is applied before casesplit version when propositional
(prove-lemma assoc-update-counter-alist-iff-repeat (rewrite)
  (implies
   (not (equal k 0))
  (iff
   (assoc k (update-counter-alist c l))
   (assoc k c))))

(prove-lemma mirror-prog-invariants-hold-means-satisfies-with-path-helper-prog (rewrite)
  (implies
   (and
    (assoc '(c 7) c)
    (assoc '(c 8) c)
    (not (assoc 'output c))
    (not (assoc 'pc c))
    (not (assoc 'step c))
    (not (assoc 'newsig c))
    (not (assoc 'signal c))
    (not (assoc 'address c))
    (not (assoc 'phase c))
    (equal state-pred (state-pred (assoc (mirror-program-map (cadr (assoc 'pc (car h))))
                                         (cadr (mirror-program-spec)))))
    (or (nlistp (cdr h))
        (equal (assoc (mirror-program-map (cadr (assoc 'pc (cadr h)))) state-trans-list)
               (assoc (mirror-program-map (cadr (assoc 'pc (cadr h))))
                      (state-trans-list (assoc (mirror-program-map (cadr (assoc 'pc (car h))))
                                               (cadr (mirror-program-spec)))))))
    (equal b (cadr (mirror-program-spec)))
    (equal p (mirror-program-map-path h))
   (mirror-prog-invariants-hold h (cadr (assoc '(c 7) c)) (cadr (assoc '(c 8) c)) 3))
   (satisfiesp-with-path-helper state-pred state-trans-list h c b p))
   ((induct (satisfiesp-with-path-helper state-pred state-trans-list h c b p))
    (enable-theory ground-zero)
    (disable-theory t)
    (expand  (mirror-prog-invariants-hold h (cadr (assoc '(c 7) c)) (cadr (assoc '(c 8) c)) 3)
             (satisfiesp-with-path-helper state-pred state-trans-list h c b p))
    (enable state-pred state-trans-list mirror-program-spec assoc-cons
```

```
           mirror-program-map-path trans-to trans-pred trans-resets
           assoc-update-counter-alist-iff-repeat assoc-update-counter-alist-casesplit
           my-eval-open mirror-prog-invariant-one-place mirror-prog-invariant-two-place
           c-m-open i-m-open t-m-open satisfiesp-with-path-helper lessp-plus-hack-rewrite
           *1*mirror-program-map mirror-program-map-normalize assoc-append *1*member
           *1*assoc)))

(prove-lemma mirror-prog-invariants-hold-means-satisfies-helper-prog (rewrite)
  (implies
   (and
    (assoc '(c 7) c)
    (assoc '(c 8) c)
    (equal (cadr (assoc 'pc (car h))) #x40000000)
    (not (assoc 'output c))
    (not (assoc 'pc c))
    (not (assoc 'step c))
    (not (assoc 'newsig c))
    (not (assoc 'signal c))
    (not (assoc 'address c))
    (not (assoc 'phase c))
    (mirror-prog-invariants-hold h (cadr (assoc '(c 7) c)) (cadr (assoc '(c 8) c)) 3)
    (equal b (cadr (mirror-program-spec)))
    (equal p (state-pred (assoc 'a0 (cadr (mirror-program-spec)))))
    (equal tl (state-trans-list (assoc 'a0 (cadr (mirror-program-spec))))))
   (satisfiesp-helper 'a0 p tl h c b))
  ((use (mirror-prog-invariants-hold-means-satisfies-with-path-helper-prog
         (state-pred (state-pred (assoc 'a0 (cadr (mirror-program-spec)))))
         (b (cadr (mirror-program-spec)))
         (p (mirror-program-map-path h))
         (state-trans-list (state-trans-list (assoc 'a0 (cadr (mirror-program-spec)))))))
   (enable mirror-program-map mirror-program-map-path
           satisfiesp-with-path-helper-means-satisfiesp-helper-rewrite
           satisfiesp-eff-helper-equiv)
   (disable-theory t)
   (enable-theory ground-zero)))

(defn cdr-add1-add1 (l n2 n3)
  (if (listp l)
      (cdr-add1-add1 (cdr l) (add1 n2) (add1 n3))
    nil))

(defn assoc-cadr (x l)
      (if (nlistp l)
          f
          (if (equal x (cadar l))
              (car l)
              (assoc-cadr x (cdr l)))))


(prove-lemma assoc-extract-locs (rewrite)
  (implies
   (not (equal x 0))
   (equal
    (cadr (assoc x (extract-locs l s)))
    (if (assoc-cadr x l)
        (if (listp (car (assoc-cadr x l)))
            (v-to-nat (read-mem (nat-to-v (caar (assoc-cadr x l)) (reg-size))
                                (regs (car s))))
          (v-to-nat (read-mem (nat-to-v (car (assoc-cadr x l)) 32)
                              (cadr s))))
      0)))
  ((induct (extract-locs l s))))
```

```
(prove-lemma assoc-extract-locs-iff (rewrite)
  (implies
   (not (equal x 0))
   (iff
    (assoc x (extract-locs l1 s))
    (assoc-cadr x l1))))


(prove-lemma car-update-state-with-inputs (rewrite)
  (equal (car (update-state-with-inputs i s)) (car s)))


(prove-lemma nth-small (rewrite)
  (and
   (equal (nth 0 l) (car l))
   (equal (nth 1 l) (cadr l)))
  ((enable nth)))


(prove-lemma listp-repeat-with-inputs-step (rewrite)
  (equal (listp (repeat-with-inputs-step phase state i o)) (listp i)))

(prove-lemma car-repeat-with-inputs-step (rewrite)
  (equal
   (car (repeat-with-inputs-step phase s i o))
   (if (listp i)
       (cons
        (list 'phase phase)
        (cons
         (list 'step (length i))
         (extract-locs o (update-state-with-inputs (car i) s))))
     0)))

(defn all-cars-numbers (l)
  (if (listp l)
      (and
       (numberp (caar l))
       (all-cars-numbers (cdr l)))
    t))

(prove-lemma remainder-plus-a-a-2 (rewrite)
  (and
   (equal (remainder (plus a a) 2) 0)
   (equal (remainder (add1 (plus a a)) 2) 1))
  ((enable plus remainder)))

(prove-lemma quotient-plus-a-a-2 (rewrite)
  (equal (quotient (plus a a) 2) (fix a))
  ((enable quotient plus)))

(prove-lemma nat-to-v-v-to-nat (rewrite)
  (implies
   (equal (length v) (fix length))
   (equal
    (nat-to-v (v-to-nat v) length)
    (v-buf v)))
  ((enable nat-to-v v-to-nat v-buf)))

(prove-lemma read-mem1-fix-v (rewrite)
  (equal (read-mem1 (v-buf a) mem)
         (read-mem1 a mem))
```

```
  ((enable read-mem1 v-buf)))

(prove-lemma rev1-fix-v-help nil
  (equal (rev1 (v-buf x) (v-buf acc))
         (v-buf (rev1 x acc)))
  ((enable rev1 v-buf)))

(prove-lemma rev1-v-buf (rewrite)
  (equal (rev1 (v-buf x) nil)
         (v-buf (rev1 x nil)))
  ((use (rev1-fix-v-help (acc nil)))))

(prove-lemma reverse-v-buf (rewrite)
  (equal (reverse (v-buf x)) (v-buf (reverse x)))
  ((enable reverse v-buf)))

(prove-lemma read-mem-fix-v (rewrite)
  (equal (read-mem (v-buf a) mem)
         (read-mem a mem))
  ((enable read-mem)))

(prove-lemma v-iff-means-same-ramp-mem1 nil
  (implies
   (and
    (equal (length x) (length y))
    (v-iff x y))
   (equal (ramp-mem1 y mem) (ramp-mem1 x mem)))
  ((enable ramp-mem1 v-iff)))

(prove-lemma v-iff-means-same-ramp-helper (rewrite)
  (implies
   (and
    (v-iff x y)
    (equal (length x) (length y)))
   (equal (ramp-mem y mem) (ramp-mem x mem)))
  ((use (v-iff-means-same-ramp-mem1 (x (reverse x)) (y (reverse y))))
   (enable ramp-mem)))

(prove-lemma not-member-list-of-cars-means-assoc (rewrite)
  (implies
   (not (member x (list-of-cars list)))
   (equal (assoc x list) f)))

(prove-lemma v-iff-v-to-nat (rewrite)
  (implies
   (and
    (v-iff y x)
    (equal (length x) (length y))
    (equal (v-to-nat y) z))
   (equal (equal (v-to-nat x) z) t))
  ((enable v-iff v-to-nat)))

(prove-lemma v-iff-v-buf-1 (rewrite)
  (equal (v-iff (v-buf a) b) (v-iff a b))
  ((enable v-iff v-buf)))

(prove-lemma v-iff-v-buf-2 (rewrite)
  (equal (v-iff a (v-buf b)) (v-iff a b))
  ((enable v-iff v-buf)))

(prove-lemma member-v-iff-v-buf (rewrite)
  (equal
```

```
   (member-v-iff (v-buf e) l)
   (member-v-iff e l))
  ((enable v-buf member-v-iff)))


(defn cdr-quotient-sub1-induct (x y n)
  (if (listp x)
      (cdr-quotient-sub1-induct (cdr x) (quotient y 2) (sub1 n))
    nil))

(prove-lemma car-nat-to-v (rewrite)
  (iff
   (car (nat-to-v x n))
   (or
    (zerop n)
    (equal (remainder x 2) 1)))
  ((expand (nat-to-v x n))))

(prove-lemma cdr-nat-to-v-hack (rewrite)
  (implies
   (lessp v 2)
   (equal
    (cdr (nat-to-v (plus v x x y y) (add1 z)))
    (nat-to-v (plus x y) z)))
  ((enable nat-to-v)
   (enable-theory naturals)))

(prove-lemma remainder-plus-hack (rewrite)
  (implies
   (equal (remainder a z) 0)
   (equal (remainder (plus d b c a) z)
          (remainder (plus d b c) z)))
  ((enable-theory naturals ground-zero)
   (disable-theory t)))

(prove-lemma remainder-add1-plus-hack (rewrite)
  (implies
   (equal (remainder a z) 0)
   (equal (remainder (add1 (plus d b c a)) z)
          (remainder (add1 (plus d b c)) z)))
  ((enable-theory naturals ground-zero)
   (use (remainder-plus-hack (b (add1 b))))
   (disable-theory t)))

(prove-lemma remainder-plus-factor-hack (rewrite)
  (implies
   (equal (remainder x y) 0)
   (equal (remainder (plus x x) (plus y y)) 0))
  ((enable-theory ground-zero naturals)))

(prove-lemma remainder-plus-x-x-plus-y-y-0 (rewrite)
  (equal
   (equal (remainder (plus x x) (plus y y)) 0)
   (equal (remainder x y) 0))
  ((enable-theory naturals)))

(prove-lemma lessp-2 (rewrite)
  (equal (lessp x 2) (or (equal x 1) (zerop x))))

(disable lessp-2)

(prove-lemma remainder-plus-plus-x-x-2 (rewrite)
```

```
  (equal (remainder (plus x x y) 2)
         (remainder y 2))
  ((enable-theory ground-zero)))

(prove-lemma remainder-add1-plus-plus-x-x-2 (rewrite)
  (equal (remainder (add1 (plus x x y)) 2)
         (remainder (add1 y) 2))
  ((enable-theory ground-zero)))

(prove-lemma listp-nat-to-v (rewrite)
  (equal (listp (nat-to-v nat n)) (not (zerop n)))
  ((enable nat-to-v)))

(prove-lemma nat-to-v-0 (rewrite)
  (equal (nat-to-v nat 0) nil)
  ((enable nat-to-v)))

(prove-lemma v-iff-nat-to-v (rewrite)
  (implies
   (equal (length x) n)
   (and
    (equal
     (v-iff x (nat-to-v y n))
     (equal (v-to-nat x) (remainder y (exp 2 n))))
    (equal
     (v-iff (nat-to-v y n) x)
     (equal (v-to-nat x) (remainder y (exp 2 n))))))
  ((enable v-to-nat v-iff length times-2 car-nat-to-v cdr-nat-to-v-hack remainder-plus-hack
           remainder-plus-x-x-plus-y-y-0 lessp-2 quotient-plus-a-a-2 v-iff-commutative
           listp-nat-to-v nat-to-v-0 v-iff remainder-add1-plus-plus-x-x-2
           remainder-plus-a-a-2 remainder-add1-plus-hack remainder-plus-plus-x-x-2)
   (induct (cdr-quotient-sub1-induct x y n))
   (expand (nat-to-v y (add1 (length z)))
           (nat-to-v (add1 (plus (v-to-nat z)
                                 (v-to-nat z)
                                 (times d (exp 2 (length z)))
                                 (times d (exp 2 (length z)))))
                     (add1 (length z)))
           (nat-to-v (add1 (plus w w
                                 (times d (exp 2 (length z)))
                                 (times d (exp 2 (length z)))))
                     (add1 (length z))))
   (disable-theory t)
   (enable-theory ground-zero naturals)))

;; all elements of list are < n
(defn list-of-non-overflowedp (list n)
  (if (listp list)
      (and
       (lessp (car list) n)
       (list-of-non-overflowedp (cdr list) n))
    t))

;; all elements of list are numbers
(defn list-of-numbers (list)
  (if (listp list)
      (and
       (numberp (car list))
       (list-of-numbers (cdr list)))
    t))

(prove-lemma member-v-iff-list-nat-to-v (rewrite)
```

```
  (implies
   (and
    (equal (length x) n)
    (list-of-non-overflowedp l (exp 2 n))
    (list-of-numbers l))
   (equal
    (member-v-iff x (list-nat-to-v l n))
    (member (v-to-nat x) l))))

(prove-lemma listp-v-buf (rewrite)
  (equal (listp (v-buf a)) (listp a))
  ((enable v-buf)))

(prove-lemma ramp-mem1-v-buf (rewrite)
  (equal (ramp-mem1 (v-buf a) mem)
         (ramp-mem1 a mem))
  ((enable ramp-mem1 v-buf)))

(prove-lemma ramp-mem-v-buf (rewrite)
  (equal (ramp-mem (v-buf a) mem)
         (ramp-mem a mem))
  ((enable ramp-mem)))

(prove-lemma read-mem-cadr-update-state-with-inputs (rewrite)
  (implies
   (and
    (equal (length a1) 32)
    (list-of-numbers (list-of-cars i))
    (list-of-non-overflowedp (list-of-cars i) (exp 2 32))
    (no-dup-cars i))
   (equal
    (read-mem a1 (cadr (update-state-with-inputs i s)))
    (if (and
         (member-v-iff a1 (list-nat-to-v (list-of-cars i) 32))
         (ramp-mem a1 (cadr s)))
        (nat-to-v (cadr (assoc (v-to-nat a1) i)) 32)
      (read-mem a1 (cadr s))))))


(prove-lemma mirror-prog-invariants-hold-repeat-with-inputs-0a (rewrite)
  (implies
    (let
        ((pc (v-to-nat (read-mem (list t t t t) (caar s))))
         (step (length h))
         (mat 3))
      (and

       (numberp c7)

       (numberp c8)

       (and (equal pc #x40000000) (equal phase 'a))

       (not (lessp (plus (c-m 3 mat) (t-m 4 mat) (i-m 0 mat))
                   (plus c7 step)))
       (mirror-fm9001-state-invariants s)))
    (mirror-prog-invariants-hold
     (repeat-with-inputs-step
      phase s
      (extract-from-history '((3221225472 signal)) h)
      '((3221225472 signal) (3221225473 output) ((15) pc) ((0) newsig) ((1) address)))
     c7 c8 3))
```

```
  ((induct (cdr-add1-add1 h c7 c8))
   (disable-theory t)
   (enable-theory ground-zero naturals)
   (enable mirror-prog-invariant-one-place equal-length-small mirror-prog-invariants-hold
           length *1*list-of-numbers list-of-cars no-dup-cars *1*list-of-non-overflowedp
           read-mem-cadr-update-state-with-inputs *1*all-cars-numbers *1*no-dup-cars
           v-iff-commutative v-iff-means-same-ramp-helper length-extract-from-history
           nth-small *1*mirror-prog *1*nth listp-repeat-with-inputs-step
           car-extract-from-history mirror-prog-invariant-two-place extract-from-assignment
           mirror-fm9001-state-invariants car-repeat-with-inputs-step
           mirror-program-map-normalize c-m-open i-m-open t-m-open extract-from-history
           reg-size regs *1*read-mem *1*v-to-nat *1*nat-to-v *1*mirror-program-map
           *1*member-v-iff listp-extract-from-history *1*list-nat-to-v *1*list-of-cars
           read-mem-cadr-update-state-with-inputs-not length-extract-from-history
           assoc-extract-locs *1*assoc-cadr assoc-extract-locs-iff
           car-update-state-with-inputs repeat-with-inputs-step)))

(prove-lemma mirror-prog-invariants-hold-repeat-with-inputs-2b (rewrite)
  (implies
    (let
        ((pc (v-to-nat (read-mem (list t t t t) (caar s))))
         (r1 (v-to-nat (read-mem (list t f f f) (caar s))))
         (step (length h))
         (mat 3))
      (and
       (numberp c7)
       (numberp c8)
       (equal pc #x40000002) (equal phase 'b)
       (not (lessp (plus (c-m 3 mat) (t-m 4 mat) (t-m 0 mat))
                   (plus c7 step)))
       (equal r1 #xc0000000)
       (mirror-fm9001-state-invariants s)))
    (mirror-prog-invariants-hold
     (repeat-with-inputs-step
      phase s
      (extract-from-history '((3221225472 signal)) h)
      '((3221225472 signal) (3221225473 output) ((15) pc) ((0) newsig) ((1) address)))
     c7 c8 3))
  ((induct (cdr-add1-add1 h c7 c8))
   (disable-theory t)
   (enable-theory ground-zero naturals)
   (enable mirror-prog-invariant-one-place equal-length-small mirror-prog-invariants-hold
           length *1*list-of-numbers list-of-cars no-dup-cars *1*list-of-non-overflowedp
           read-mem-cadr-update-state-with-inputs *1*all-cars-numbers *1*no-dup-cars
           v-iff-commutative v-iff-means-same-ramp-helper length-extract-from-history
           nth-small *1*mirror-prog *1*nth listp-repeat-with-inputs-step
           car-extract-from-history mirror-prog-invariant-two-place extract-from-assignment
           mirror-fm9001-state-invariants car-repeat-with-inputs-step
           mirror-program-map-normalize c-m-open i-m-open t-m-open extract-from-history
           reg-size regs *1*read-mem *1*v-to-nat *1*nat-to-v *1*mirror-program-map
           *1*member-v-iff listp-extract-from-history *1*list-nat-to-v *1*list-of-cars
           read-mem-cadr-update-state-with-inputs-not length-extract-from-history
           assoc-extract-locs *1*assoc-cadr assoc-extract-locs-iff
           car-update-state-with-inputs repeat-with-inputs-step)))

(prove-lemma mirror-prog-invariants-hold-repeat-with-inputs-2a (rewrite)
  (implies
    (let
        ((pc (v-to-nat (read-mem (list t t t t) (caar s))))
         (r1 (v-to-nat (read-mem (list t f f f) (caar s))))
         (step (length h))
         (mat 3))
```

```
      (and
       (numberp c7)
       (numberp c8)
       (equal pc #x40000002) (equal phase 'a)
       (not (lessp (plus (c-m 3 mat) (t-m 4 mat) (t-m 0 mat) (i-m 1 mat))
                   (plus c7 step)))
       (equal r1 #xc0000000)
       (mirror-fm9001-state-invariants s)))
    (mirror-prog-invariants-hold
     (repeat-with-inputs-step
      phase s
      (extract-from-history '((3221225472 signal)) h)
      '((3221225472 signal) (3221225473 output) ((15) pc) ((0) newsig) ((1) address)))
     c7 c8 3))
  ((induct (cdr-add1-add1 h c7 c8))
   (disable-theory t)
   (enable-theory ground-zero naturals)
   (enable mirror-prog-invariant-one-place equal-length-small mirror-prog-invariants-hold
           length *1*list-of-numbers list-of-cars no-dup-cars *1*list-of-non-overflowedp
           read-mem-cadr-update-state-with-inputs *1*all-cars-numbers *1*no-dup-cars
           v-iff-commutative v-iff-means-same-ramp-helper length-extract-from-history
           nth-small *1*mirror-prog *1*nth listp-repeat-with-inputs-step
           car-extract-from-history mirror-prog-invariant-two-place extract-from-assignment
           mirror-fm9001-state-invariants car-repeat-with-inputs-step
           mirror-program-map-normalize c-m-open i-m-open t-m-open extract-from-history
           reg-size regs *1*read-mem *1*v-to-nat *1*nat-to-v *1*mirror-program-map
           *1*member-v-iff listp-extract-from-history *1*list-nat-to-v *1*list-of-cars
           read-mem-cadr-update-state-with-inputs-not length-extract-from-history
           assoc-extract-locs *1*assoc-cadr assoc-extract-locs-iff
           car-update-state-with-inputs repeat-with-inputs-step)))

(prove-lemma mirror-prog-invariants-hold-repeat-with-inputs-3b (rewrite)
  (implies
    (let
        ((pc (v-to-nat (read-mem (list t t t t) (caar s))))
         (r1 (v-to-nat (read-mem (list t f f f) (caar s))))
         (step (length h))
         (mat 3))
      (and
       (numberp c7)
       (numberp c8)
       (equal pc #x40000003) (equal phase 'b)
       (not (lessp (c-m 1 mat)
                   (plus c8 step)))
       (mirror-fm9001-state-invariants s)))
    (mirror-prog-invariants-hold
     (repeat-with-inputs-step
      phase s
      (extract-from-history '((3221225472 signal)) h)
      '((3221225472 signal) (3221225473 output) ((15) pc) ((0) newsig) ((1) address)))
     c7 c8 3))
  ((induct (cdr-add1-add1 h c7 c8))
   (disable-theory t)
   (enable-theory ground-zero naturals)
   (enable mirror-prog-invariant-one-place equal-length-small mirror-prog-invariants-hold
           length *1*list-of-numbers list-of-cars no-dup-cars *1*list-of-non-overflowedp
           read-mem-cadr-update-state-with-inputs *1*all-cars-numbers *1*no-dup-cars
           v-iff-commutative v-iff-means-same-ramp-helper length-extract-from-history
           nth-small *1*mirror-prog *1*nth listp-repeat-with-inputs-step
           car-extract-from-history mirror-prog-invariant-two-place extract-from-assignment
           mirror-fm9001-state-invariants car-repeat-with-inputs-step
           mirror-program-map-normalize c-m-open i-m-open t-m-open extract-from-history
```

```
          reg-size regs *1*read-mem *1*v-to-nat *1*nat-to-v *1*mirror-program-map
          *1*member-v-iff listp-extract-from-history *1*list-nat-to-v *1*list-of-cars
          read-mem-cadr-update-state-with-inputs-not length-extract-from-history
          assoc-extract-locs *1*assoc-cadr assoc-extract-locs-iff
          car-update-state-with-inputs repeat-with-inputs-step)))

(prove-lemma mirror-prog-invariants-hold-repeat-with-inputs-3a (rewrite)
  (implies
    (let
        ((pc (v-to-nat (read-mem (list t t t t) (caar s))))
         (r1 (v-to-nat (read-mem (list t f f f) (caar s))))
         (step (length h))
         (mat 3))
      (and
       (numberp c7)
       (numberp c8)
       (equal pc #x40000003) (equal phase 'a)
       (not (lessp (plus (c-m 1 mat) (i-m 2 mat))
                   (plus c8 step)))
       (mirror-fm9001-state-invariants s)))
    (mirror-prog-invariants-hold
     (repeat-with-inputs-step
      phase s
      (extract-from-history '((3221225472 signal)) h)
      '((3221225472 signal) (3221225473 output) ((15) pc) ((0) newsig) ((1) address)))
     c7 c8 3))
  ((induct (cdr-add1-add1 h c7 c8))
   (disable-theory t)
   (enable-theory ground-zero naturals)
   (enable mirror-prog-invariant-one-place equal-length-small mirror-prog-invariants-hold
          length *1*list-of-numbers list-of-cars no-dup-cars *1*list-of-non-overflowedp
          read-mem-cadr-update-state-with-inputs *1*all-cars-numbers *1*no-dup-cars
          v-iff-commutative v-iff-means-same-ramp-helper length-extract-from-history
          nth-small *1*mirror-prog *1*nth listp-repeat-with-inputs-step
          car-extract-from-history mirror-prog-invariant-two-place extract-from-assignment
          mirror-fm9001-state-invariants car-repeat-with-inputs-step
          mirror-program-map-normalize c-m-open i-m-open t-m-open extract-from-history
          reg-size regs *1*read-mem *1*v-to-nat *1*nat-to-v *1*mirror-program-map
          *1*member-v-iff listp-extract-from-history *1*list-nat-to-v *1*list-of-cars
          read-mem-cadr-update-state-with-inputs-not length-extract-from-history
          assoc-extract-locs *1*assoc-cadr assoc-extract-locs-iff
          car-update-state-with-inputs repeat-with-inputs-step)))

(prove-lemma mirror-prog-invariants-hold-repeat-with-inputs-5b (rewrite)
  (implies
    (let
        ((pc (v-to-nat (read-mem (list t t t t) (caar s))))
         (r1 (v-to-nat (read-mem (list t f f f) (caar s))))
         (step (length h))
         (mat 3))
      (and
       (numberp c7)
       (numberp c8)
       (equal pc #x40000005) (equal phase 'b)
       (not (lessp (plus (c-m 1 mat) (t-m 2 mat))
                   (plus c8 step)))
       (equal r1 #xc0000001)
       (mirror-fm9001-state-invariants s)))
    (mirror-prog-invariants-hold
     (repeat-with-inputs-step
      phase s
      (extract-from-history '((3221225472 signal)) h)
```

```
      '((3221225472 signal) (3221225473 output) ((15) pc) ((0) newsig) ((1) address)))
     c7 c8 3))
  ((induct (cdr-add1-add1 h c7 c8))
   (disable-theory t)
   (enable-theory ground-zero naturals)
   (enable mirror-prog-invariant-one-place equal-length-small mirror-prog-invariants-hold
           length *1*list-of-numbers list-of-cars no-dup-cars *1*list-of-non-overflowedp
           read-mem-cadr-update-state-with-inputs *1*all-cars-numbers *1*no-dup-cars
           v-iff-commutative v-iff-means-same-ramp-helper length-extract-from-history
           nth-small *1*mirror-prog *1*nth listp-repeat-with-inputs-step
           car-extract-from-history mirror-prog-invariant-two-place extract-from-assignment
           mirror-fm9001-state-invariants car-repeat-with-inputs-step
           mirror-program-map-normalize c-m-open i-m-open t-m-open extract-from-history
           reg-size regs *1*read-mem *1*v-to-nat *1*nat-to-v *1*mirror-program-map
           *1*member-v-iff listp-extract-from-history *1*list-nat-to-v *1*list-of-cars
           read-mem-cadr-update-state-with-inputs-not length-extract-from-history
           assoc-extract-locs *1*assoc-cadr assoc-extract-locs-iff
           car-update-state-with-inputs repeat-with-inputs-step)))

(prove-lemma mirror-prog-invariants-hold-repeat-with-inputs-5a (rewrite)
  (implies
    (let
        ((pc (v-to-nat (read-mem (list t t t t) (caar s))))
         (r1 (v-to-nat (read-mem (list t f f f) (caar s))))
         (step (length h))
         (mat 3))
      (and
       (numberp c7)
       (numberp c8)
       (equal pc #x40000005) (equal phase 'a)
       (not (lessp (plus (c-m 1 mat) (t-m 2 mat) (i-m 3 mat))
                   (plus c8 step)))
       (equal r1 #xc0000001)
       (mirror-fm9001-state-invariants s)))
    (mirror-prog-invariants-hold
     (repeat-with-inputs-step
      phase s
      (extract-from-history '((3221225472 signal)) h)
      '((3221225472 signal) (3221225473 output) ((15) pc) ((0) newsig) ((1) address)))
     c7 c8 3))
  ((induct (cdr-add1-add1 h c7 c8))
   (disable-theory t)
   (enable-theory ground-zero naturals)
   (enable mirror-prog-invariant-one-place equal-length-small mirror-prog-invariants-hold
           length *1*list-of-numbers list-of-cars no-dup-cars *1*list-of-non-overflowedp
           read-mem-cadr-update-state-with-inputs *1*all-cars-numbers *1*no-dup-cars
           v-iff-commutative v-iff-means-same-ramp-helper length-extract-from-history
           nth-small *1*mirror-prog *1*nth listp-repeat-with-inputs-step
           car-extract-from-history mirror-prog-invariant-two-place extract-from-assignment
           mirror-fm9001-state-invariants car-repeat-with-inputs-step
           mirror-program-map-normalize c-m-open i-m-open t-m-open extract-from-history
           reg-size regs *1*read-mem *1*v-to-nat *1*nat-to-v *1*mirror-program-map
           *1*member-v-iff listp-extract-from-history *1*list-nat-to-v *1*list-of-cars
           read-mem-cadr-update-state-with-inputs-not length-extract-from-history
           assoc-extract-locs *1*assoc-cadr assoc-extract-locs-iff
           car-update-state-with-inputs repeat-with-inputs-step)))


(prove-lemma mirror-prog-invariants-hold-repeat-with-inputs-6b (rewrite)
```

```
  (implies
    (let
        ((pc (v-to-nat (read-mem (list t t t t) (caar s))))
         (r1 (v-to-nat (read-mem (list t f f f) (caar s))))
         (step (length h))
         (mat 3))
      (and
       (numberp c7)
       (numberp c8)
       (and (equal pc #x40000006) (equal phase 'b))
       (not (lessp (c-m 3 mat) (plus step c7)))
       (mirror-fm9001-state-invariants s)))
    (mirror-prog-invariants-hold
     (repeat-with-inputs-step
      phase s
      (extract-from-history '((3221225472 signal)) h)
      '((3221225472 signal) (3221225473 output) ((15) pc) ((0) newsig) ((1) address)))
     c7 c8 3))
  ((induct (cdr-add1-add1 h c7 c8))
   (disable-theory t)
   (enable-theory ground-zero naturals)
   (enable mirror-prog-invariant-one-place equal-length-small mirror-prog-invariants-hold
           length *1*list-of-numbers list-of-cars no-dup-cars *1*list-of-non-overflowedp
           read-mem-cadr-update-state-with-inputs *1*all-cars-numbers *1*no-dup-cars
           v-iff-commutative v-iff-means-same-ramp-helper length-extract-from-history
           nth-small *1*mirror-prog *1*nth listp-repeat-with-inputs-step
           car-extract-from-history mirror-prog-invariant-two-place extract-from-assignment
           mirror-fm9001-state-invariants car-repeat-with-inputs-step
           mirror-program-map-normalize c-m-open i-m-open t-m-open extract-from-history
           reg-size regs *1*read-mem *1*v-to-nat *1*nat-to-v *1*mirror-program-map
           *1*member-v-iff listp-extract-from-history *1*list-nat-to-v *1*list-of-cars
           read-mem-cadr-update-state-with-inputs-not length-extract-from-history
           assoc-extract-locs *1*assoc-cadr assoc-extract-locs-iff
           car-update-state-with-inputs repeat-with-inputs-step)))

(prove-lemma mirror-prog-invariants-hold-repeat-with-inputs-6a (rewrite)
  (implies
    (let
        ((pc (v-to-nat (read-mem (list t t t t) (caar s))))
         (r1 (v-to-nat (read-mem (list t f f f) (caar s))))
         (step (length h))
         (mat 3))
      (and
       (numberp c7)
       (numberp c8)
       (equal pc #x40000006) (equal phase 'a)
       (not (lessp (plus (c-m 3 mat) (i-m 4 mat)) (plus step c7)))
       (mirror-fm9001-state-invariants s)))
    (mirror-prog-invariants-hold
     (repeat-with-inputs-step
      phase s
      (extract-from-history '((3221225472 signal)) h)
      '((3221225472 signal) (3221225473 output) ((15) pc) ((0) newsig) ((1) address)))
     c7 c8 3))
  ((induct (cdr-add1-add1 h c7 c8))
   (disable-theory t)
   (enable-theory ground-zero naturals)
   (enable mirror-prog-invariant-one-place equal-length-small mirror-prog-invariants-hold
           length *1*list-of-numbers list-of-cars no-dup-cars *1*list-of-non-overflowedp
           read-mem-cadr-update-state-with-inputs *1*all-cars-numbers *1*no-dup-cars
           v-iff-commutative v-iff-means-same-ramp-helper length-extract-from-history
           nth-small *1*mirror-prog *1*nth listp-repeat-with-inputs-step
```

```
          car-extract-from-history mirror-prog-invariant-two-place extract-from-assignment
          mirror-fm9001-state-invariants car-repeat-with-inputs-step
          mirror-program-map-normalize c-m-open i-m-open t-m-open extract-from-history
          reg-size regs *1*read-mem *1*v-to-nat *1*nat-to-v *1*mirror-program-map
          *1*member-v-iff listp-extract-from-history *1*list-nat-to-v *1*list-of-cars
          read-mem-cadr-update-state-with-inputs-not length-extract-from-history
          assoc-extract-locs *1*assoc-cadr assoc-extract-locs-iff
          car-update-state-with-inputs repeat-with-inputs-step)))

(prove-lemma mirror-prog-invariants-hold-repeat-with-inputs-0b (rewrite)
  (implies
    (let
        ((pc (v-to-nat (read-mem (list t t t t) (caar s))))
         (r1 (v-to-nat (read-mem (list t f f f) (caar s))))
         (step (length h))
         (mat 3))
      (and
       (numberp c7)
       (numberp c8)
       (equal pc #x40000000) (equal phase 'b)
       (not (lessp (plus (c-m 3 mat) (t-m 4 mat)) (plus step c7)))
       (mirror-fm9001-state-invariants s)))
    (mirror-prog-invariants-hold
     (repeat-with-inputs-step
      phase s
      (extract-from-history '((3221225472 signal)) h)
      '((3221225472 signal) (3221225473 output) ((15) pc) ((0) newsig) ((1) address)))
     c7 c8 3))
  ((induct (cdr-add1-add1 h c7 c8))
   (disable-theory t)
   (enable-theory ground-zero naturals)
   (enable mirror-prog-invariant-one-place equal-length-small mirror-prog-invariants-hold
          length *1*list-of-numbers list-of-cars no-dup-cars *1*list-of-non-overflowedp
          read-mem-cadr-update-state-with-inputs *1*all-cars-numbers *1*no-dup-cars
          v-iff-commutative v-iff-means-same-ramp-helper length-extract-from-history
          nth-small *1*mirror-prog *1*nth listp-repeat-with-inputs-step
          car-extract-from-history mirror-prog-invariant-two-place extract-from-assignment
          mirror-fm9001-state-invariants car-repeat-with-inputs-step
          mirror-program-map-normalize c-m-open i-m-open t-m-open extract-from-history
          reg-size regs *1*read-mem *1*v-to-nat *1*nat-to-v *1*mirror-program-map
          *1*member-v-iff listp-extract-from-history *1*list-nat-to-v *1*list-of-cars
          read-mem-cadr-update-state-with-inputs-not length-extract-from-history
          assoc-extract-locs *1*assoc-cadr assoc-extract-locs-iff
          car-update-state-with-inputs repeat-with-inputs-step)))

;; calculate the value of c7 given an initial value and a history
;; assuming use of mirror-prog-specification
(defn calculate-c7-from-history (c7 h)
  (if (listp h)
      (if (listp (cdr h))
          (if (and (equal (mirror-program-map (cadr (assoc 'pc (car h)))) 'a1)
                   (equal (mirror-program-map (cadr (assoc 'pc (cadr h)))) 'a0))
              (calculate-c7-from-history 0 (cdr h))
            (calculate-c7-from-history (add1 c7) (cdr h)))
        c7)
    c7))

(defn calculate-c8-from-history (c8 h)
  (if (listp h)
      (if (listp (cdr h))
          (if (and (equal (mirror-program-map (cadr (assoc 'pc (car h)))) 'a0)
                   (equal (mirror-program-map (cadr (assoc 'pc (cadr h)))) 'a1))
```

```
            (calculate-c8-from-history 0 (cdr h))
          (calculate-c8-from-history (add1 c8) (cdr h)))
      c8)
  c8))

(prove-lemma mirror-prog-invariants-hold-append (rewrite)
 (equal
  (mirror-prog-invariants-hold (append x y) c7 c8 mat)
  (if (listp x)
      (if (listp y)
          (and

           ;; first part satisfied
           (mirror-prog-invariants-hold x c7 c8 mat)

           ;; transition satisfied
           (let
               ((pc1 (cadr (assoc 'pc (last x))))
                (phase1 (cadr (assoc 'phase (last x))))
                (step1 (cadr (assoc 'step (last x))))
                (address1 (cadr (assoc 'address (last x))))
                (output1 (cadr (assoc 'output (last x))))
                (newsig1 (cadr (assoc 'newsig (last x))))
                (signal1 (cadr (assoc 'signal (last x))))
                (pc2 (cadr (assoc 'pc (car y))))
                (phase2 (cadr (assoc 'phase (car y))))
                (step2 (cadr (assoc 'step (car y))))
                (address2 (cadr (assoc 'address (car y))))
                (newsig2 (cadr (assoc 'newsig (car y))))
                (output2 (cadr (assoc 'output (car y))))

                (c71 (calculate-c7-from-history c7 x))
                (c81 (calculate-c8-from-history c8 x)))
             (let ((state-name1 (mirror-program-map pc1))
                   (state-name2 (mirror-program-map pc2)))
               (let ((c72 (if (and (equal state-name1 'a1) (equal state-name2 'a0))
                              0 (add1 c71)))
                     (c82 (if (and (equal state-name1 'a0) (equal state-name2 'a1))
                              0 (add1 c81))))

                 (and
                  (mirror-prog-invariant-two-place
                   state-name1 state-name2 newsig1 newsig2 output1 output2 signal1
                   c71 c72 c81 c82)

                  ;; second part satisfied

                  (mirror-prog-invariants-hold y c72 c82 mat))))))
        (mirror-prog-invariants-hold x c7 c8 mat))
    (mirror-prog-invariants-hold y c7 c8 mat)))
 ((induct (mirror-prog-invariants-hold x c7 c8 mat))
  (expand (mirror-prog-invariants-hold (append x y) c7 c8 mat))
  (enable mirror-program-map-normalize *1*mirror-program-map mirror-prog-invariants-hold
          last calculate-c7-from-history calculate-c8-from-history car-append listp-append)
  (enable-theory ground-zero)
  (disable-theory t)))

(prove-lemma last-repeat-with-inputs-step (rewrite)
 (equal
  (last (repeat-with-inputs-step phase s i o))
  (if (listp i)
      (cons
```

```
        (list 'phase phase)
        (cons
         (list 'step 1)
         (extract-locs o (update-state-with-inputs (last i) s))))
     nil))
  ((induct (repeat-with-inputs-step phase s i o))))

(prove-lemma last-extract-from-history (rewrite)
  (equal
   (last (extract-from-history o h))
   (if (listp h)
       (extract-from-assignment o (last h))
     nil)))

(prove-lemma nth-extract-from-history (rewrite)
  (equal
   (nth n (extract-from-history o h))
   (if (lessp n (length h))
       (extract-from-assignment o (nth n h))
     0))
  ((enable nth)
   (induct (nth n h))))

(defn cdr-add1-induct (l n)
  (if (listp l)
      (cdr-add1-induct (cdr l) (add1 n))
    nil))

(prove-lemma calculate-c7-from-history-repeat-with-inputs-step (rewrite)
  (implies
   (listp (car (assoc-cadr 'pc o)))
  (equal
   (calculate-c7-from-history c7 (repeat-with-inputs-step phase s i o))
   (if (listp (cdr i))
       (plus (length (cdr i)) c7)
     c7)))
  ((induct (cdr-add1-induct i c7))))

(prove-lemma calculate-c8-from-history-repeat-with-inputs-step (rewrite)
  (implies
   (listp (car (assoc-cadr 'pc o)))
  (equal
   (calculate-c8-from-history c8 (repeat-with-inputs-step phase s i o))
   (if (listp (cdr i))
       (plus (length (cdr i)) c8)
     c8)))
  ((induct (cdr-add1-induct i c8))))


(prove-lemma readmem-when-length-and-value-known (rewrite)
  (implies
   (and
    (equal (length v) n)
    (equal (v-to-nat v) n2))
   (equal (read-mem v mem) (read-mem (nat-to-v n2 n) mem)))
  ((enable read-mem nat-to-v v-to-nat)))

(prove-lemma ramp-mem-when-length-and-value-known (rewrite)
  (implies
   (and
    (equal (length v) n)
    (equal (v-to-nat v) n2))
```

```
   (equal (ramp-mem v mem) (ramp-mem (nat-to-v n2 n) mem)))
  ((enable ramp-mem nat-to-v v-to-nat)))


(prove-lemma v-adder-when-length-and-value-known (rewrite)
  (implies
   (and
    (equal (length v) n)
    (equal (v-to-nat v) n2))
   (equal (v-adder-output c v v2) (v-adder-output c (nat-to-v n2 n) v2)))
  ((enable v-adder v-adder-output nat-to-v v-to-nat firstn v-buf)))

(prove-lemma mirror-fm9001-state-invariants-update (rewrite)
  (implies
   (and
    (mirror-fm9001-state-invariants s)
    (not (member #x40000000 (list-of-cars i)))
    (not (member #x40000001 (list-of-cars i)))
    (not (member #x40000002 (list-of-cars i)))
    (not (member #x40000003 (list-of-cars i)))
    (not (member #x40000004 (list-of-cars i)))
    (not (member #x40000005 (list-of-cars i)))
    (not (member #x40000006 (list-of-cars i)))
    (not (member #x40000007 (list-of-cars i)))
    (not (member #xc0000001 (list-of-cars i)))
    (list-of-non-overflowedp (list-of-cars i) (exp 2 32))
    (no-dup-cars i)
    (list-of-numbers (list-of-cars i)))
   (mirror-fm9001-state-invariants (update-state-with-inputs i s)))
  ((enable regs)))


(prove-lemma mirror-next-pc (rewrite)
  (implies
   (mirror-fm9001-state-invariants s)
   (equal
    (v-to-nat
     (read-mem
      (list t t t t)
      (caar (fm9001-step s (list t t t t)))))
    (if (equal (v-to-nat (read-mem (list t t t t) (caar s))) #x40000000)
        #x40000002
      (if (equal (v-to-nat (read-mem (list t t t t) (caar s))) #x40000002)
          #x40000003
        (if (equal (v-to-nat (read-mem (list t t t t) (caar s))) #x40000003)
            #x40000005
          (if (equal (v-to-nat (read-mem (list t t t t) (caar s))) #x40000005)
              #x40000006
            #x40000000))))))
  ((enable fm9001-step regs mirror-fm9001-state-invariants
           read-mem-cadr-update-state-with-inputs readmem-when-length-and-value-known
           mirror-prog *1*nth member-v-iff-list-nat-to-v *1*asm *1*nat-to-v *1*v-to-nat
           *1*op-code *1*v-buf *1*cvzbv *1*bv v-alu store-resultp *1*store-cc
           v-adder-when-length-and-value-known *1*v-adder-output nth-small *1*a-immediate-p
           *1*mode-a *1*pre-dec-p *1*reg-direct-p *1*rn-a *1*rn-b *1*post-inc-p *1*v-iff
           *1*pre-dec-p *1*mode-b flags fm9001-alu-operation read-mem-write-mem-rewrite
           write-mem-write-mem-same write-mem-write-mem-different fm9001-operand-a
           fm9001-operand-b v-inc *1*length car-update-state-with-inputs fm9001-fetch)))

(prove-lemma no-dup-cars-extract-from-assignment (rewrite)
  (equal
   (no-dup-cars (extract-from-assignment o a))
```

```
   (no-dup-cars o)))

(prove-lemma repeat-with-inputs-step-nil (rewrite)
  (equal (repeat-with-inputs-step p s nil o) nil))

(prove-lemma extract-from-assignment-open (rewrite)
  (equal
   (extract-from-assignment (cons a b) assignment)
   (cons (list (car a) (cadr (assoc (cadr a) assignment)))
         (extract-from-assignment b assignment))))

(disable extract-from-assignment-open)


(prove-lemma update-state-with-inputs-open (rewrite)
  (equal
   (update-state-with-inputs (cons a b) state)
   (update-state-with-inputs
    b
    (list (car state)
          (write-mem (nat-to-v (car a) 32) (cadr state)
                     (nat-to-v (cadr a) 32))))))

(disable update-state-with-inputs-open)

(prove-lemma extract-from-assignment-nil (rewrite)
  (equal (extract-from-assignment nil a) nil))

(prove-lemma update-state-with-inputs-nil (rewrite)
  (equal (update-state-with-inputs nil s) s))


(defn fm9001-rt-history-repeat-step-helper-induct (state mat inputs-list h loclist c7 c8)
  (let
      ((init-time (init-time state mat))
       (total-time (microcycles-constant state mat)))
    (if (read-write-statep state) nil
      (if (not (lessp init-time (length inputs-list)))
          nil
        (let ((new-state
               (fm9001-step
                (update-state-with-inputs (nth (sub1 init-time) inputs-list) state)
                (nat-to-v 15 (reg-size)))))
          (if (not (lessp total-time (length inputs-list)))
              nil
            (let
                ((pc1 (v-to-nat (read-mem (list t t t t) (caar state))))
                 (pc2 (v-to-nat (read-mem (list t t t t) (caar new-state)))))
              (fm9001-rt-history-repeat-step-helper-induct
               new-state mat
               (nthcdr total-time inputs-list)
               (nthcdr total-time h) loclist
               (if (and (equal (mirror-program-map pc1) 'a1)
                        (equal (mirror-program-map pc2) 'a0))
                   (difference total-time init-time)
                 (plus c7 total-time))
               (if (and (equal (mirror-program-map pc1) 'a0)
                        (equal (mirror-program-map pc2) 'a1))
                   (difference total-time init-time)
                 (plus c8 total-time)))))))))
  ((lessp (length inputs-list))))
```

```
(prove-lemma read-mem-light-fm9001-step (rewrite)
  (implies
   (mirror-fm9001-state-invariants s)
   (equal
    (read-mem
     (list t f f f f f f f f f f f f f f f f f f f f f f f f f f f f f t t)
     (cadr (fm9001-step s (list t t t t))))
    (if (and
         (equal (v-to-nat (read-mem (list t t t t) (caar s))) #x40000005)
         (v-iff (read-mem (list t f f f) (caar s))
                (list t f f f f f f f f f f f f f f f f f f f f f f f f f f f f f t t)))
        (v-buf (read-mem (list f f f f) (caar s)))
      (read-mem
       (list t f f f f f f f f f f f f f f f f f f f f f f f f f f f f f t t)
       (cadr s)))))
  ((enable fm9001-step regs mirror-fm9001-state-invariants
           read-mem-cadr-update-state-with-inputs readmem-when-length-and-value-known
           mirror-prog *1*nth member-v-iff-list-nat-to-v *1*asm *1*nat-to-v *1*v-to-nat bv
           *1*op-code *1*v-buf *1*cvzbv *1*bv v-alu store-resultp *1*store-cc
           v-adder-when-length-and-value-known *1*v-adder-output nth-small *1*a-immediate-p
           *1*mode-a *1*pre-dec-p *1*reg-direct-p *1*rn-a *1*rn-b *1*post-inc-p *1*v-iff
           *1*pre-dec-p *1*mode-b flags fm9001-alu-operation read-mem-write-mem-rewrite
           write-mem-write-mem-same write-mem-write-mem-different fm9001-operand-a
           fm9001-operand-b v-inc *1*length car-update-state-with-inputs fm9001-fetch)))

(prove-lemma read-mem-r0-fm9001-step (rewrite)
  (implies
   (mirror-fm9001-state-invariants s)
   (equal
    (read-mem
     (list f f f f)
     (caar (fm9001-step s (list t t t t))))
    (if (equal (v-to-nat (read-mem (list t t t t) (caar s))) #x40000002)
        (v-buf (read-mem (read-mem (list t f f f) (caar s)) (cadr s)))
      (read-mem (list f f f f) (caar s)))))
  ((enable fm9001-step regs mirror-fm9001-state-invariants
           read-mem-cadr-update-state-with-inputs readmem-when-length-and-value-known
           mirror-prog *1*nth member-v-iff-list-nat-to-v *1*asm *1*nat-to-v *1*v-to-nat bv
           *1*op-code *1*v-buf *1*cvzbv *1*bv v-alu store-resultp *1*store-cc
           v-adder-when-length-and-value-known *1*v-adder-output nth-small *1*a-immediate-p
           *1*mode-a *1*pre-dec-p *1*reg-direct-p *1*rn-a *1*rn-b *1*post-inc-p *1*v-iff
           *1*pre-dec-p *1*mode-b flags fm9001-alu-operation read-mem-write-mem-rewrite
           write-mem-write-mem-same write-mem-write-mem-different fm9001-operand-a
           fm9001-operand-b v-inc *1*length car-update-state-with-inputs fm9001-fetch)))

(prove-lemma read-mem-r1-fm9001-step (rewrite)
  (implies
   (mirror-fm9001-state-invariants s)
   (equal
    (read-mem
     (list t f f f)
     (caar (fm9001-step s (list t t t t))))
    (if (equal (v-to-nat (read-mem (list t t t t) (caar s))) #x40000003)
        (nat-to-v #xc0000001 32)
      (if (equal (v-to-nat (read-mem (list t t t t) (caar s))) #x40000000)
          (nat-to-v #xc0000000 32)
        (read-mem (list t f f f) (caar s))))))
  ((enable fm9001-step regs mirror-fm9001-state-invariants
           read-mem-cadr-update-state-with-inputs readmem-when-length-and-value-known
           mirror-prog *1*nth member-v-iff-list-nat-to-v *1*asm *1*nat-to-v *1*v-to-nat bv
           *1*op-code *1*v-buf *1*cvzbv *1*bv v-alu store-resultp *1*store-cc
           v-adder-when-length-and-value-known *1*v-adder-output nth-small *1*a-immediate-p
```

```
         *1*mode-a *1*pre-dec-p *1*reg-direct-p *1*rn-a *1*rn-b *1*post-inc-p *1*v-iff
         *1*pre-dec-p *1*mode-b flags fm9001-alu-operation read-mem-write-mem-rewrite
         write-mem-write-mem-same write-mem-write-mem-different fm9001-operand-a
         fm9001-operand-b v-inc *1*length car-update-state-with-inputs fm9001-fetch)))

(prove-lemma v-iff-when-length-and-value-known (rewrite)
  (implies
   (and
    (equal (length v) n)
    (equal (v-to-nat v) n2))
   (equal (v-iff x v) (v-iff x (nat-to-v n2 n))))
  ((enable v-iff nat-to-v v-to-nat)))

(prove-lemma program-unchanged-fm9001-step (rewrite)
  (implies
   (and
    (mirror-fm9001-state-invariants s)
    (or
     (equal x (list f f f f f f f f f f f f f f f f f f f f f f f f f f f f f f t f))
     (equal x (list t f f f f f f f f f f f f f f f f f f f f f f f f f f f f f t f))
     (equal x (list f t f f f f f f f f f f f f f f f f f f f f f f f f f f f f t f))
     (equal x (list t t f f f f f f f f f f f f f f f f f f f f f f f f f f f f t f))
     (equal x (list f f t f f f f f f f f f f f f f f f f f f f f f f f f f f f t f))
     (equal x (list t f t f f f f f f f f f f f f f f f f f f f f f f f f f f f t f))
     (equal x (list f t t f f f f f f f f f f f f f f f f f f f f f f f f f f f t f))
     (equal x (list t t t f f f f f f f f f f f f f f f f f f f f f f f f f f f t f)))
    (or (not (equal (v-to-nat (read-mem (list t t t t) (caar s))) #x40000005))
        (equal (v-to-nat (read-mem (list t f f f) (caar s))) #xc0000001)))
   (equal
    (read-mem x (cadr (fm9001-step s (list t t t t))))
    (read-mem x (cadr s))))
  ((enable fm9001-step regs mirror-fm9001-state-invariants
         read-mem-cadr-update-state-with-inputs readmem-when-length-and-value-known
         mirror-prog *1*nth member-v-iff-list-nat-to-v v-iff-when-length-and-value-known
         *1*asm *1*nat-to-v *1*v-to-nat bv *1*op-code *1*v-buf *1*cvzbv *1*bv v-alu
         store-resultp *1*store-cc v-adder-when-length-and-value-known *1*v-adder-output
         nth-small *1*a-immediate-p *1*mode-a *1*pre-dec-p *1*reg-direct-p *1*rn-a *1*rn-b
         *1*post-inc-p *1*v-iff *1*pre-dec-p *1*mode-b flags fm9001-alu-operation
         read-mem-write-mem-rewrite write-mem-write-mem-same write-mem-write-mem-different
         fm9001-operand-a fm9001-operand-b v-inc *1*length car-update-state-with-inputs
         fm9001-fetch)))

(prove-lemma length-pc-fm9001-step (rewrite)
  (implies
   (mirror-fm9001-state-invariants s)
   (equal
    (length (read-mem (list t t t t) (caar (fm9001-step s (list t t t t)))))
    32))
  ((enable fm9001-step regs mirror-fm9001-state-invariants
         read-mem-cadr-update-state-with-inputs readmem-when-length-and-value-known
         mirror-prog *1*nth member-v-iff-list-nat-to-v v-iff-when-length-and-value-known
         *1*asm *1*nat-to-v *1*v-to-nat bv *1*op-code *1*v-buf *1*cvzbv *1*bv v-alu
         store-resultp *1*store-cc v-adder-when-length-and-value-known *1*v-adder-output
         nth-small *1*a-immediate-p *1*mode-a *1*pre-dec-p *1*reg-direct-p *1*rn-a *1*rn-b
         *1*post-inc-p *1*v-iff *1*pre-dec-p *1*mode-b flags fm9001-alu-operation
         read-mem-write-mem-rewrite write-mem-write-mem-same write-mem-write-mem-different
         fm9001-operand-a fm9001-operand-b v-inc *1*length car-update-state-with-inputs
         fm9001-fetch)))

(prove-lemma length-signal-fm9001-step (rewrite)
  (implies
   (mirror-fm9001-state-invariants s)
```

```
  (equal
   (length (read-mem (list f f f f f f f f f f f f f f f f f f f f f f f f f f f f f f t t)
                     (cadr (fm9001-step s (list t t t t)))))
   32))
 ((enable fm9001-step regs mirror-fm9001-state-invariants
          read-mem-cadr-update-state-with-inputs readmem-when-length-and-value-known
          mirror-prog *1*nth member-v-iff-list-nat-to-v v-iff-when-length-and-value-known
          *1*asm *1*nat-to-v *1*v-to-nat bv *1*op-code *1*v-buf *1*cvzbv *1*bv v-alu
          store-resultp *1*store-cc v-adder-when-length-and-value-known *1*v-adder-output
          nth-small *1*a-immediate-p *1*mode-a *1*pre-dec-p *1*reg-direct-p *1*rn-a *1*rn-b
          *1*post-inc-p *1*v-iff *1*pre-dec-p *1*mode-b flags fm9001-alu-operation
          read-mem-write-mem-rewrite write-mem-write-mem-same write-mem-write-mem-different
          fm9001-operand-a fm9001-operand-b v-inc *1*length car-update-state-with-inputs
          fm9001-fetch)))

(prove-lemma ramp-unchanged-fm9001-step (rewrite)
 (implies
  (and
  (mirror-fm9001-state-invariants s)
   (or
    (equal x (list f f f f))
    (equal x (list t f f f))
    (equal x (list t t t t))))
  (ramp-mem x (caar (fm9001-step s (list t t t t)))))
 ((enable fm9001-step regs mirror-fm9001-state-invariants
          read-mem-cadr-update-state-with-inputs readmem-when-length-and-value-known
          mirror-prog *1*nth member-v-iff-list-nat-to-v v-iff-when-length-and-value-known
          *1*asm *1*nat-to-v *1*v-to-nat bv *1*op-code *1*v-buf *1*cvzbv *1*bv v-alu
          store-resultp *1*store-cc v-adder-when-length-and-value-known *1*v-adder-output
          nth-small *1*a-immediate-p *1*mode-a *1*pre-dec-p *1*reg-direct-p *1*rn-a *1*rn-b
          *1*post-inc-p *1*v-iff *1*pre-dec-p *1*mode-b flags fm9001-alu-operation
          read-mem-write-mem-rewrite write-mem-write-mem-same write-mem-write-mem-different
          fm9001-operand-a fm9001-operand-b v-inc *1*length car-update-state-with-inputs
          fm9001-fetch)))


(prove-lemma mirror-next-pc-bitv (rewrite)
 (implies
  (mirror-fm9001-state-invariants s)
  (equal
    (read-mem
     (list t t t t)
     (caar (fm9001-step s (list t t t t))))
   (if (equal (v-to-nat (read-mem (list t t t t) (caar s))) #x40000000)
       (nat-to-v #x40000002 32)
     (if (equal (v-to-nat (read-mem (list t t t t) (caar s))) #x40000002)
         (nat-to-v #x40000003 32)
       (if (equal (v-to-nat (read-mem (list t t t t) (caar s))) #x40000003)
           (nat-to-v #x40000005 32)
         (if (equal (v-to-nat (read-mem (list t t t t) (caar s))) #x40000005)
             (nat-to-v #x40000006 32)
           (nat-to-v #x40000000 32)))))))
 ((enable fm9001-step regs mirror-fm9001-state-invariants
          read-mem-cadr-update-state-with-inputs readmem-when-length-and-value-known
          mirror-prog *1*nth member-v-iff-list-nat-to-v *1*asm *1*nat-to-v *1*v-to-nat
          *1*op-code *1*v-buf *1*cvzbv *1*bv v-alu store-resultp *1*store-cc
          v-adder-when-length-and-value-known *1*v-adder-output nth-small *1*a-immediate-p
          *1*mode-a *1*pre-dec-p *1*reg-direct-p *1*rn-a *1*rn-b *1*post-inc-p *1*v-iff
          *1*pre-dec-p *1*mode-b flags fm9001-alu-operation read-mem-write-mem-rewrite
          write-mem-write-mem-same write-mem-write-mem-different fm9001-operand-a
          fm9001-operand-b v-inc *1*length car-update-state-with-inputs fm9001-fetch))
```

```
(prove-lemma read-write-statep-fm9001-step (rewrite)
  (implies
   (and
    (mirror-fm9001-state-invariants s)
    (or (not (equal (v-to-nat (read-mem (list t t t t) (caar s))) #x40000005))
        (equal (v-to-nat (read-mem (list t f f f) (caar s))) #xc0000001)))
   (not (read-write-statep (fm9001-step s (list t t t t)))))
  ((enable regs mirror-next-pc-bitv mirror-fm9001-state-invariants
           read-mem-cadr-update-state-with-inputs readmem-when-length-and-value-known
           mirror-prog *1*nth member-v-iff-list-nat-to-v v-iff-when-length-and-value-known
           *1*asm *1*nat-to-v *1*v-to-nat bv *1*op-code *1*v-buf *1*cvzbv *1*bv v-alu
           store-resultp *1*store-cc v-adder-when-length-and-value-known *1*v-adder-output
           nth-small *1*a-immediate-p *1*mode-a *1*pre-dec-p *1*reg-direct-p *1*rn-a *1*rn-b
           *1*post-inc-p *1*v-iff *1*pre-dec-p *1*mode-b flags fm9001-alu-operation
           read-mem-write-mem-rewrite write-mem-write-mem-same write-mem-write-mem-different
           fm9001-operand-a fm9001-operand-b v-inc *1*length car-update-state-with-inputs
           fm9001-fetch)))

(prove-lemma init-time-is-i-m (rewrite)
  (implies
   (mirror-fm9001-state-invariants s)
   (equal (init-time s 3)
          (if (equal (v-to-nat (read-mem (list t t t t) (caar s))) #x40000000)
              (i-m 0 3)
            (if (equal (v-to-nat (read-mem (list t t t t) (caar s))) #x40000002)
                (i-m 1 3)
              (if (equal (v-to-nat (read-mem (list t t t t) (caar s))) #x40000003)
                  (i-m 2 3)
                (if (equal (v-to-nat (read-mem (list t t t t) (caar s))) #x40000005)
                    (i-m 3 3)
                  (i-m 4 3)))))))
  ((enable init-time regs store-resultp)))

(prove-lemma microcycles-constant-is-t-m (rewrite)
  (implies
   (mirror-fm9001-state-invariants s)
   (equal (microcycles-constant s 3)
          (if (equal (v-to-nat (read-mem (list t t t t) (caar s))) #x40000000)
              (t-m 0 3)
            (if (equal (v-to-nat (read-mem (list t t t t) (caar s))) #x40000002)
                (t-m 1 3)
              (if (equal (v-to-nat (read-mem (list t t t t) (caar s))) #x40000003)
                  (t-m 2 3)
                (if (equal (v-to-nat (read-mem (list t t t t) (caar s))) #x40000005)
                    (t-m 3 3)
                  (t-m 4 3)))))))
  ((enable microcycles-constant regs store-resultp)))


(prove-lemma mirror-fm9001-state-invariants-open (rewrite)
  (equal (mirror-fm9001-state-invariants s)
  (let
      ((mem (cadr s))
       (regs (caar s)))
    (let
        ((pc (v-to-nat (read-mem (nat-to-v 15 4) regs)))
         (r1 (v-to-nat (read-mem (nat-to-v 1 4) regs))))

      (and
       (equal (read-mem (nat-to-v #x40000000 32) mem)
              (list t t t t t t f f f f t f f f f f f f f f f f t t t f f f f f f f))
       (equal (read-mem (nat-to-v #x40000001 32) mem)
```

```
             (list f f f f f f f f f f f f f f f f f f f f f f f f f f f f f f t t))
      (equal (read-mem (nat-to-v #x40000002 32) mem)
             (list t f f f t f f f f f f f f f f f f f f f f t t t f f f f f f f f))
      (equal (read-mem (nat-to-v #x40000003 32) mem)
             (list t t t t t t f f f f t f f f f f f f f f f t t t f f f f f f f f))
      (equal (read-mem (nat-to-v #x40000004 32) mem)
             (list t f f f f f f f f f f f f f f f f f f f f f f f f f f f f f t t))
      (equal (read-mem (nat-to-v #x40000005 32) mem)
             (list f f f f f f f f f f t f f f t f f f f f f t t t f f f f f f f f))
      (equal (read-mem (nat-to-v #x40000006 32) mem)
             (list t t t t t t f f f f t t t t f f f f f f f t t t f f f f f f f f))
      (equal (read-mem (nat-to-v #x40000007 32) mem)
             (list f f f f f f f f f f f f f f f f f f f f f f f f f f f f f f t f))

      (equal (length (read-mem (nat-to-v #xc0000000 32) mem)) 32)
      (ramp-mem (nat-to-v #xc0000000 32) mem)

      (ramp-mem (nat-to-v #xc0000001 32) mem)

      (ramp-mem (nat-to-v 0 4) regs)
      (ramp-mem (nat-to-v 1 4) regs)
      (ramp-mem (nat-to-v 15 4) regs)

      (equal (length (read-mem (nat-to-v 15 4) regs)) 32)

      (equal (length (read-mem (nat-to-v 1 4) regs)) 32)

      (equal (length (read-mem (nat-to-v 0 4) regs)) 32)


      (or (equal pc #x40000000) (equal pc #x40000002) (equal pc #x40000003)
          (equal pc #x40000005) (equal pc #x40000006))))))
 ((enable regs)))



(prove-lemma mirror-prog-invariant-two-place-open (rewrite)
  (equal
   (mirror-prog-invariant-two-place
    state-name1 state-name2 newsig1 newsig2 output1 output2 signal1 c71 c72 c81 c82)
  (if (equal state-name1 'a0)
      (if (equal state-name2 'a0)
          (and
           (equal c72 (add1 c71))
           (equal c82 (add1 c81))
           (equal newsig1 newsig2)
           (equal output1 output2))
        (and
         (equal c72 (add1 c71))
         (equal c82 0)
         (equal output1 output2)
         (equal newsig2 signal1)))
    (if (equal state-name2 'a0)
        (and
         (equal c72 0)
         (equal c82 (add1 c81))
         (equal output2 newsig1)
         (equal newsig1 newsig2))
      (and
       (equal c72 (add1 c71))
       (equal c82 (add1 c81))
       (equal output2 output1)
```

```
      (equal newsig1 newsig2))))))


(prove-lemma mirror-prog-invariant-one-place-open (rewrite)
  (equal (mirror-prog-invariant-one-place pc phase step r1 mat c7 c8)
  (and

   (numberp c7)

   (numberp c8)

   (or (equal phase 'a) (equal phase 'b))

   (or (equal pc #x40000000)
       (equal pc #x40000002)
       (equal pc #x40000003)
       (equal pc #x40000005)
       (equal pc #x40000006))

   (implies (and (equal pc #x40000000) (equal phase 'a))
            (not (lessp (plus (c-m 3 mat) (t-m 4 mat) (i-m 0 mat))
                        (plus c7 step))))

   (implies (and (equal pc #x40000002) (equal phase 'b))
            (and (not (lessp (plus (c-m 3 mat) (t-m 4 mat) (t-m 0 mat))
                             (plus c7 step)))
                 (equal r1 #xc0000000)))

   (implies (and (equal pc #x40000002) (equal phase 'a))
            (and
            (not (lessp (plus (c-m 3 mat) (t-m 4 mat) (t-m 0 mat) (i-m 1 mat))
                        (plus c7 step)))
            (equal r1 #xc0000000)))


   (implies (and (equal pc #x40000003) (equal phase 'b))
            (not (lessp (c-m 1 mat)
                        (plus c8 step))))

   (implies (and (equal pc #x40000003) (equal phase 'a))
            (not (lessp (plus (c-m 1 mat) (i-m 2 mat))
                        (plus c8 step))))

   (implies (and (equal pc #x40000005) (equal phase 'b))
            (and
             (not (lessp (plus (c-m 1 mat) (t-m 2 mat))
                         (plus c8 step)))
             (equal r1 #xc0000001)))

   (implies (and (equal pc #x40000005) (equal phase 'a))
            (and
             (not (lessp (plus (c-m 1 mat) (t-m 2 mat) (i-m 3 mat))
                         (plus c8 step)))
             (equal r1 #xc0000001)))

   (implies (and (equal pc #x40000006) (equal phase 'b))
            (not (lessp (c-m 3 mat) (plus step c7))))

   (implies (and (equal pc #x40000006) (equal phase 'a))
            (not (lessp (plus (c-m 3 mat) (i-m 4 mat)) (plus step c7))))

   (implies (and (equal pc #x40000000) (equal phase 'b))
```

```
           (not (lessp (plus (c-m 3 mat) (t-m 4 mat)) (plus step c7)))))))

(prove-lemma lessp-hack-from-free-variables (rewrite)
  (implies
   (and
    (lessp a b)
    (not (lessp x b))
    (not (lessp (add1 a) y)))
   (equal (lessp x y) f)))

(prove-lemma nthcdr-nil (rewrite)
  (equal (nthcdr n nil) (if (zerop n) nil 0)))


(disable lessp-hack-from-free-variables)

(prove-lemma v-to-nat-v-buf (rewrite)
  (equal (v-to-nat (v-buf v))  (v-to-nat v))
  ((enable v-to-nat v-buf)))

(prove-lemma add1-sub1-init-time (rewrite)
  (equal (add1 (sub1 (init-time s m)))
         (init-time s m)))

(prove-lemma car-fm9001-rt-history-repeat-step-helper (rewrite)
  (equal
   (car (fm9001-rt-history-repeat-step-helper state mat inputs-list loclist))
   (if (listp inputs-list)
       (if (read-write-statep state)
             0
         (car (repeat-with-inputs-step
               'a state (firstn (init-time state mat) inputs-list) loclist)))
     0))
  ((expand (fm9001-rt-history-repeat-step-helper state mat inputs-list loclist))
   (disable read-write-statep)))

(prove-lemma nthcdr-add1-cons (rewrite)
  (equal (nthcdr (add1 x) (cons a b))
         (nthcdr x b))
  ((enable nthcdr)))

(prove-lemma mirror-prog-invariants-hold-nil (rewrite)
  (mirror-prog-invariants-hold nil c7 c8 mat))

(prove-lemma listp-fm9001-rt-history-repeat-step-helper  (rewrite)
  (equal
   (listp (fm9001-rt-history-repeat-step-helper s m i l))
   (and (listp i) (not (read-write-statep s))))
  ((disable read-write-statep fm9001-rt-history-repeat-step-helper)
   (expand (fm9001-rt-history-repeat-step-helper s m i l))))

(prove-lemma lessp-add1-add1 (rewrite)
  (equal (lessp (add1 x) (add1 y)) (lessp x y)))

(prove-lemma lessp-add1-plus-add1 (rewrite)
  (equal (lessp (add1 x) (plus (add1 y) z))
         (lessp x (plus y z))))

(prove-lemma lessp-plus-add1-add1 (rewrite)
  (equal (lessp (plus (add1 y) z) (add1 x))
         (lessp (plus y z) x)))
```

```
(prove-lemma lessp-plus-add1-plus-add1 (rewrite)
  (equal (lessp (plus (add1 a) b) (plus (add1 x) y))
         (lessp (plus a b) (plus x y))))

(prove-lemma plus-zero (rewrite)
  (and
   (equal (plus 0 x) (fix x))
   (equal (plus x 0) (fix x))))

(prove-lemma lessp-difference-casesplit (rewrite)
  (equal (lessp (difference x y) z)
         (or
          (and
           (not (lessp y x))
           (not (zerop z)))
          (and
           (lessp y x)
           (lessp x (plus y z)))))
  ((induct (difference x y))))

(prove-lemma listp-cdr-nthcdr (rewrite)
  (equal (listp (cdr (nthcdr n l)))
         (lessp (add1 n) (length l)))
  ((enable nthcdr)))

(defn mirror-execution-step (s)
  (let ((pc (v-to-nat (read-mem (list t t t t) (caar s)))))
    (if (equal pc #x40000000)
        (list
         (list
          (write-mem (list t t t t)
                     (write-mem (list t f f f)
                                (caar s)
                                (nat-to-v #xc0000000 32))
                     (nat-to-v #x40000002 32))
          (v-buf (list (caadar s) (cadadar s) (caddadar s) (cadddadar s))))
         (cadr s))
      (if (equal pc #x40000002)
          (list
           (list
            (write-mem (list t t t t)
                       (write-mem (list f f f f)
                                  (caar s)
                                  (v-buf (read-mem
                                          (read-mem (list t f f f) (caar s))
                                          (cadr s))))
                       (nat-to-v #x40000003 32))
            (v-buf (list (caadar s) (cadadar s) (caddadar s) (cadddadar s))))
           (cadr s))
        (if (equal pc #x40000003)
            (list
             (list
              (write-mem (list t t t t)
                         (write-mem (list t f f f)
                                    (caar s)
                                    (nat-to-v #xc0000001 32))
                         (nat-to-v #x40000005 32))
              (v-buf (list (caadar s) (cadadar s) (caddadar s) (cadddadar s))))
             (cadr s))
          (if (equal pc #x40000005)
              (list
               (list
```

```
                (write-mem (list t t t t)
                           (caar s)
                           (nat-to-v #x40000006 32))
                (v-buf (list (caadar s) (cadadar s) (caddadar s) (cadddadar s))))
               (write-mem (read-mem (list t f f f) (caar s))
                          (cadr s)
                          (v-buf (read-mem (list f f f f) (caar s)))))
           (list
            (list (write-mem (list t t t t)
                             (caar s)
                             (nat-to-v #x40000000 32))
                  (v-buf (list (caadar s) (cadadar s) (caddadar s) (cadddadar s))))
            (cadr s))))))))

(prove-lemma fm9001-step-as-mirror-execution-step (rewrite)
  (implies
   (mirror-fm9001-state-invariants s)
   (equal
    (fm9001-step s (list t t t t))
    (mirror-execution-step s)))
  ((enable regs fm9001-step mirror-next-pc-bitv update-flags mirror-fm9001-state-invariants
           read-mem-cadr-update-state-with-inputs readmem-when-length-and-value-known
           mirror-prog *1*nth member-v-iff-list-nat-to-v v-iff-when-length-and-value-known
           *1*asm *1*nat-to-v *1*v-to-nat firstn v-buf b-buf bv c-flag v-flag n-flag z-flag
           *1*op-code *1*v-buf *1*cvzbv *1*bv v-alu store-resultp *1*store-cc
           v-adder-when-length-and-value-known *1*v-adder-output nth-small *1*a-immediate-p
           *1*mode-a *1*pre-dec-p *1*reg-direct-p *1*rn-a *1*rn-b *1*post-inc-p *1*v-iff
           *1*pre-dec-p *1*mode-b flags fm9001-alu-operation read-mem-write-mem-rewrite
           write-mem-write-mem-same write-mem-write-mem-different fm9001-operand-a
           fm9001-operand-b v-inc *1*length car-update-state-with-inputs fm9001-fetch))

(prove-lemma lessp-sub1-add1 (rewrite)
  (implies
   (lessp 0 x)
   (equal
    (lessp (sub1 x) (add1 y))
    (lessp x (add1 (add1 y))))))

(prove-lemma lessp-subsumed (rewrite)
  (implies
   (and
    (lessp a c)
    (not (lessp a b)))
   (equal (lessp b c) t)))

(prove-lemma mirror-prog-invariants-hold-of-fm9001 nil
  (let
      ((pc (v-to-nat (read-mem (list t t t t) (caar s))))
       (address (v-to-nat (read-mem (list t f f f) (caar s))))
       (newsig (v-to-nat (read-mem (list f f f f) (caar s))))
       (output (v-to-nat (read-mem (nat-to-v #xc0000001 32) (cadr s))))
       (signal (assoc #xc0000000 (car i))))
  (implies
   (and
    (mirror-fm9001-state-invariants s)
    (mirror-prog-invariant-one-place
     pc 'a (init-time s 3)
     address 3 c7 c8)
    (equal i (extract-from-history '((#xc0000000 signal)) h))
    (equal o '((#xc0000000 signal) (#xc0000001 output) ((15) pc) ((0) newsig) ((1) address)))
    (equal mat 3))
   (mirror-prog-invariants-hold
```

```
    (fm9001-rt-history-repeat-step-helper s mat i o)
    c7 c8 3)))
  ((induct (fm9001-rt-history-repeat-step-helper-induct s mat i h o c7 c8))
   (disable-theory t)
   (enable-theory ground-zero)
   (expand (fm9001-rt-history-repeat-step-helper s 3
                                                 (extract-from-history '((3221225472 signal))
                                                                       h)
                                                 '((3221225472 signal)
                                                   (3221225473 output)
                                                   ((15) pc)
                                                   ((0) newsig)
                                                   ((1) address))))
   (enable mirror-prog-invariants-hold-nil mirror-program-map-normalize mirror-execution-step
           fm9001-step-as-mirror-execution-step length-nlistp-cdr lessp-sub1-add1
           lessp-subsumed listp-cdr-nthcdr listp-fm9001-rt-history-repeat-step-helper
           cdr-firstn nthcdr-firstn commutativity-of-plus commutativity2-of-plus
           associativity-of-plus lessp-add1-add1 *1*plus plus-add1 plus-zero
           lessp-difference-casesplit lessp-hack-from-free-variables
           readmem-when-length-and-value-known ramp-mem-when-length-and-value-known
           v-iff-when-length-and-value-known v-iff-commutative car-append listp-append
           car-fm9001-rt-history-repeat-step-helper firstn-add1-cons nthcdr-add1-cons
           mirror-prog-invariant-one-place-open v-to-nat-v-buf car-extract-from-history
           mirror-prog-invariant-two-place-open mirror-fm9001-state-invariants-open
           mirror-next-pc read-mem-write-mem-rewrite extract-from-assignment-open
           extract-from-assignment-nil update-state-with-inputs-open
           update-state-with-inputs-nil lessp-init-time-1 equal-sub1-0 length-v-buf
           *1*mirror-prog *1*v-iff length-nat-to-v c-m-open t-m-open i-m-open length-nthcdr
           add1-sub1-init-time *1*v-to-nat *1*list-nat-to-v *1*member-v-iff
           read-mem-cadr-update-state-with-inputs
           mirror-prog-invariants-hold-repeat-with-inputs-0a
           mirror-prog-invariants-hold-repeat-with-inputs-2b
           mirror-prog-invariants-hold-repeat-with-inputs-2a
           mirror-prog-invariants-hold-repeat-with-inputs-3b
           mirror-prog-invariants-hold-repeat-with-inputs-3a
           mirror-prog-invariants-hold-repeat-with-inputs-5b
           mirror-prog-invariants-hold-repeat-with-inputs-5a
           mirror-prog-invariants-hold-repeat-with-inputs-6b
           mirror-prog-invariants-hold-repeat-with-inputs-6a
           mirror-prog-invariants-hold-repeat-with-inputs-0b
           no-dup-cars-extract-from-assignment list-of-cars-extract-from-assignment
           *1*list-of-non-overflowedp *1*list-of-cars *1*list-of-numbers *1*no-dup-cars
           *1*exp mirror-fm9001-state-invariants-update *1*member
           listp-repeat-with-inputs-step car-repeat-with-inputs-step assoc-extract-locs
           cdr-extract-from-history *1*length length-cdr-rewrite assoc-cadr
           calculate-c7-from-history-repeat-with-inputs-step
           calculate-c8-from-history-repeat-with-inputs-step *1*nth
           nthcdr-extract-from-history cdr-firstn equal-length-small
           repeat-with-inputs-step-nil nth-small mirror-program-map reg-size nthcdr-nil
           nth-extract-from-history *1*nat-to-v car-update-state-with-inputs car-nthcdr
           last-extract-from-history regs ramp-mem-fm9001-step ramp-mem-write-mem last-firstn
           microcycles-constant-is-t-m init-time-is-i-m listp-firstn length-cons
           last-repeat-with-inputs-step listp-nthcdr ramp-mem-update-state-with-inputs
           length-firstn firstn-extract-from-history equal-init-time-0
           lessp-init-time-total-time listp-extract-from-history length-extract-from-history
           mirror-prog-invariants-hold-append)))

(prove-lemma mirror-prog-invariants-hold-of-fm9001-rewrite (rewrite)
  (let
      ((pc (v-to-nat (read-mem (list t t t t) (caar s))))
       (address (v-to-nat (read-mem (list t f f f) (caar s))))
       (newsig (v-to-nat (read-mem (list f f f f) (caar s))))
```

```
      (output (v-to-nat (read-mem (nat-to-v #xc0000001 32) (cadr s))))
      (signal (assoc #xc0000000 (car i))))
  (implies
   (and
    (mirror-fm9001-state-invariants s)
    (mirror-prog-invariant-one-place
     pc 'a (init-time s 3)
     address 3 c7 c8)
    (equal o '((#xc0000000 signal) (#xc0000001 output) ((15) pc) ((0) newsig) ((1) address)))
    (equal mat 3))
   (mirror-prog-invariants-hold
    (fm9001-rt-history-repeat-step-helper
     s mat (extract-from-history '((#xc0000000 signal)) h) o)
    c7 c8 3)))
  ((use (mirror-prog-invariants-hold-of-fm9001
         (i (extract-from-history '((#xc0000000 signal)) h))))
   (disable-theory t)
   (enable-theory ground-zero)))


(prove-lemma mirror-program-meets-spec2-silly nil
  (implies
   (listp h)
  (satisfiesp
   (fm9001-rt-history-repeat-step-helper
    (mirror-initial-state)
    3
    (extract-from-history '((#xc0000000 signal)) h)
    '((#xc0000000 signal) (#xc0000001 output) ((15) pc) ((0) newsig) ((1) address)))
   (mirror-program-spec)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable mirror-prog-invariants-hold-means-satisfies-helper-prog
           mirror-prog-invariants-hold-of-fm9001-rewrite *1*mirror-fm9001-state-invariants
           *1*mirror-prog-invariant-one-place state-name assoc behavior-initial
           behavior-state-list *1*mirror-program-spec assoc *1*nth *1*read-mem *1*v-to-nat
           *1*assoc-cadr reg-size *1*nat-to-v regs car-update-state-with-inputs car-firstn
           car-extract-from-history assoc-extract-locs assoc-cons *1*init-time
           *1*mirror-initial-state *1*read-write-statep listp-firstn
           listp-extract-from-history car-repeat-with-inputs-step state-pred
           car-fm9001-rt-history-repeat-step-helper state-trans-list make-list-alist
           *1*counters-of-state-list satisfiesp)))

(prove-lemma mirror-program-meets-spec nil
  (satisfiesp
   (fm9001-rt-history-repeat-step-helper
    (mirror-initial-state)
    3
    (extract-from-history '((#xc0000000 signal)) h)
    '((#xc0000000 signal) (#xc0000001 output) ((15) pc) ((0) newsig) ((1) address)))
   (mirror-program-spec))
  ((use (mirror-program-meets-spec2-silly))
   (expand (fm9001-rt-history-repeat-step-helper
            (mirror-initial-state)
            3 nil
            '((3221225472 signal)
              (3221225473 output)
              ((15) pc)
              ((0) newsig)
              ((1) address))))
   (disable-theory t)
   (enable-theory ground-zero)
```

```
   (enable satisfiesp-nlistp extract-from-history fm9001-rt-history-repeat-step-helper length
           repeat-with-inputs-step-nil)))


;; Now we tie it all up.

(prove-lemma mirror-program-meets-spec-no-step-helper (rewrite)
  (satisfiesp
   (fm9001-rt-history-repeat-helper
    (mirror-initial-state)
    3
    (extract-from-history '((#xc0000000 signal)) h)
    '((#xc0000000 signal) (#xc0000001 output) ((15) pc) ((0) newsig) ((1) address)))
   (mirror-program-spec))
  ((use (satisfiesp-rembinding-rembinding-hack
         (v1 'step)
         (v2 'phase)
         (h (fm9001-rt-history-repeat-step-helper
             (mirror-initial-state)
              3
              (extract-from-history '((#xc0000000 signal)) h)
              '((#xc0000000 signal) (#xc0000001 output) ((15) pc) ((0) newsig)
                ((1) address))))
         (b (mirror-program-spec)))
        (mirror-program-meets-spec))
   (disable-theory t)
   (enable-theory ground-zero naturals)
   (enable rembinding-history-rt-history-repeat-step behavior-state-list variable-in-termp
           variable-in-translist trans-pred *1*mirror-program-spec variable-in-state-listp
           remove-if-cadr state-pred state-trans-list)))


(prove-lemma light-switch-works-helper nil
  (implies
   (and
    (listp h)
    (satisfiesp h (button-spec))
    (extension-historyp
     (fm9001-rt-history (mirror-initial-state) 3
                        (extract-from-history '((#xc0000000 signal)) h)
                        '((#xc0000000 signal) (#xc0000001 output) ((0) newsig)))
     h))
   (satisfiesp h (mirror-system-spec)))
  ((use (mirror-program-meets-spec-no-step-helper)
        (mirror-satisfiesp)
        (satisfiesp-rembinding-history
         (h (fm9001-rt-history-repeat-helper
             (mirror-initial-state)
             3
             (extract-from-history '((3221225472 signal))
                                   h)
             '((3221225472 signal)
               (3221225473 output)
               ((15) pc)
               ((0) newsig)
               ((1) address))))
         (b (mirror-program-spec))
         (v 'address))
        (satisfiesp-rembinding-history
         (h (fm9001-rt-history-repeat-helper
             (mirror-initial-state)
             3
```

```
            (extract-from-history '((3221225472 signal))
                                  h)
            '((3221225472 signal)
              (3221225473 output)
              ((15) pc)
              ((0) newsig))))
        (b (mirror-program-spec))
        (v 'pc)))
  (disable-theory t)
  (enable-theory ground-zero)
  (enable extension-assignmentp car-when-extension-hack
          rembinding-history-fm9001-rt-history-repeat-helper *1*remove-if-cadr
          light-switch-helper-helper *1*variable-in-state-listp
          behavior-state-list-mirror-program-spec extension-historyp-cons
          car-fm9001-rt-history extract-locs-silly-hack
          read-write-statep-mirror-initial-state listp-extract-from-history)))


(prove-lemma light-switch-works-ver2-helper nil
 (implies
  (and
   (listp h)
   (satisfiesp h (button-spec))
   (extension-historyp
               (fm9001-rt-history (mirror-initial-state)
                                  3
                                  (extract-from-history '((3221225472 signal))
                                                        h)
                                  '((3221225472 signal)
                                    (3221225473 output)))
               h))
  (satisfiesp h (mirror-system-spec)))
 ((disable-theory t)
  (use (light-switch-works-helper
        (h
         (zipper-lists
          (fm9001-rt-history (mirror-initial-state)
                             3
                             (extract-from-history '((3221225472 signal))
                                                   h)
                             '(((0) newsig)))
          h)))
        (permutation-means-extension-fm9001-rt-history-same
         (s (mirror-initial-state))
         (m 3)
         (i (extract-from-history '((3221225472 signal)) h))
         (x '((3221225472 signal) (3221225473 output) ((0) newsig)))
         (y (append '(((0) newsig)) '(((3221225472 signal) (3221225473 output)))))
         (z (zipper-lists
               (fm9001-rt-history (mirror-initial-state) 3
                                  (extract-from-history '((3221225472 signal)) h)
                                  '(((0) newsig)))
               h))))
  (enable listp-zipper-lists firstn-too-big member-append subset-cons
          not-lessp-length-from-extension-historyp-fm9001 state-trans-list behavior-initial
          behavior-state-list make-list-alist counters-of-state-list satisfiesp-nlistp
          *1*intersection extension-historyp-extract-from-history-ziplist
          intersection-single-element *1*subset equal-length-from-extension-historyp-fm9001
          satisfiesp-zipper-lists button-spec mirror-system-spec trans-pred trans-resets
          state-name all-variables-in-term counters-of-trans counters-of-term state-pred
          counters-of-translist counters-of-state set-union all-variables-in-translist
          *1*all-variables-in-state-list extension-historyp list-of-cadrs set-difference
```

```
         all-bound-in-history-fm9001-rt-history listp-extract-from-history
         read-write-statep-mirror-initial-state listp-fm9001-rt-history
         extract-from-history-zipper-lists extract-from-history-make-properp
         intersection-nil all-litatoms no-dup-cadrs)
  (disable append *1*append)
  (enable-theory ground-zero)))

(prove-lemma light-switch-works-ver2 nil
 (implies
  (and
   (satisfiesp h (button-spec))
   (extension-historyp
               (fm9001-rt-history (mirror-initial-state)
                                  3
                                  (extract-from-history '((3221225472 signal))
                                                        h)
                                  '((3221225472 signal)
                                    (3221225473 output)))
               h))
  (satisfiesp h (mirror-system-spec)))
 ((use (light-switch-works-ver2-helper))
  (disable-theory t)
  (enable-theory ground-zero)
  (enable satisfiesp satisfiesp-helper)))

(prove-lemma light-switch-works-helper2 nil
 (implies
  (and
   (listp h)
   (satisfiesp h (button-spec))
   (variable-not-overflowed-in-historyp 'signal h)
   (extension-historyp
               (fm9001-rt-history (mirror-initial-state)
                                  3
                                  (extract-from-history '((3221225472 signal))
                                                        h)
                                  '((3221225473 output)))
               h))
  (satisfiesp h (mirror-system-spec)))
 ((disable-theory t)
  (use (light-switch-works-helper
        (h
         (zipper-lists
          (fm9001-rt-history (mirror-initial-state)
                             3
                             (extract-from-history '((3221225472 signal))
                                                   h)
                             '(((0) newsig)))
          h)))
        (permutation-means-extension-fm9001-rt-history-same
         (s (mirror-initial-state))
         (m 3)
         (i (extract-from-history '((3221225472 signal)) h))
         (x '((3221225472 signal) (3221225473 output) ((0) newsig)))
         (y (append '(((0) newsig)) '(((3221225472 signal) (3221225473 output)))))
         (z (zipper-lists
               (fm9001-rt-history (mirror-initial-state) 3
                                  (extract-from-history '((3221225472 signal)) h)
                                  '(((0) newsig)))
               h))))
  (enable listp-zipper-lists firstn-too-big member-append subset-cons
         not-lessp-length-from-extension-historyp-fm9001 state-trans-list behavior-initial
```

```
          behavior-state-list make-list-alist counters-of-state-list satisfiesp-nlistp
          *1*intersection extension-historyp-extract-from-history-ziplist
          intersection-single-element *1*subset equal-length-from-extension-historyp-fm9001
          satisfiesp-zipper-lists location-ramp-1 button-spec mirror-system-spec trans-pred
          trans-resets state-name all-variables-in-term counters-of-trans counters-of-term
          state-pred counters-of-translist counters-of-state set-union
          all-variables-in-translist *1*all-variables-in-state-list extension-historyp
          list-of-cadrs set-difference all-bound-in-history-fm9001-rt-history
          listp-extract-from-history read-write-statep-mirror-initial-state
          listp-fm9001-rt-history extract-from-history-zipper-lists
          extract-from-history-make-properp intersection-nil all-litatoms no-dup-cadrs
          member-v-iff *1*list-nat-to-v *1*list-of-cars *1*ramp-mem *1*write-array-memory
          *1*repeat *1*empty-memory *1*asm variable-not-overflowed-in-historyp
          variable-not-overflowed-in-historyp-zipper-lists
          extension-history-fm9001-copy-unnecc variable-not-overflowed-in-historyp-firstn
          variable-not-overflowed-in-historyp-fm9001-rt-history-simple)
  (disable append *1*append)
  (enable-theory ground-zero)))

(prove-lemma light-switch-works nil
 (implies
  (and
   (satisfiesp h (button-spec))
   (variable-not-overflowed-in-historyp 'signal h)
   (extension-historyp
     (fm9001-rt-history (mirror-initial-state) 3
                        (extract-from-history '((#xc0000000 signal)) h)
                        '((#xc0000001 output)))
     h))
  (satisfiesp h (mirror-system-spec)))
 ((use (light-switch-works-helper2))
  (disable-theory t) (enable satisfiesp-nlistp) (enable-theory ground-zero)))

;We go on something of a diversion in order to prove something about
;the relationship between the programmer's and real-time models of the
;FM9001

(prove-lemma lessp-fix (rewrite)
  (and
   (equal (lessp (fix a) b) (lessp a b))
   (equal (lessp a (fix b)) (lessp a b))))

(defn number-fm9001-instructions
      (n s mat pc)
      (let ((init (init-time s mat))
            (total (microcycles-constant s mat)))
           (if (not (lessp total n))
               (if (lessp init n) 1 0)
               (add1 (number-fm9001-instructions (difference n total)
                                                 (fm9001-step s pc)
                                                 mat pc))))
      ((lessp (fix n))))

(defn read-write-encounteredp (s n pc)
  (if (read-write-statep s) t
    (if (zerop n) f
      (read-write-encounteredp (fm9001-step s pc) (sub1 n) pc)))
  ((lessp (fix n))))

(prove-lemma last-as-nth (rewrite)
  (equal (last l) (if (listp l) (nth (sub1 (length l)) l) nil)))
```

```
(disable last-as-nth)

(defn fm9001-rt-history-versus-induct (s n mat pc)
  (if (or (read-write-statep s)
          (lessp n (microcycles-constant s mat)))
      t
    (fm9001-rt-history-versus-induct
     (fm9001-step s pc) (difference n (microcycles-constant s mat))
     mat pc))
  ((lessp (fix n))))

(prove-lemma listp-fm9001-rt-history-helper (rewrite)
  (equal (listp (fm9001-rt-history-helper s mat inputs loclist 0))
         (and (not (read-write-statep s)) (listp inputs)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable length-nlistp firstn-1 fm9001-rt last length lessp-0-init-time
           *1*read-write-statep equal-length-small)
   (expand (fm9001-rt-history-helper s mat inputs loclist 0)
           (update-state-with-inputs x s)
           (fm9001-rt s (list x) mat))))

(prove-lemma nth-length-cons (rewrite)
  (equal (nth (length l) (cons a l)) (if (listp l) (nth (sub1 (length l)) l) a)))

(prove-lemma lessp-sub1-hack2 (rewrite)
  (implies
   (lessp 0 a)
  (equal
   (lessp (sub1 (plus a x)) (plus b x))
   (lessp (sub1 a) b))))

(prove-lemma listp-repeat-2 (rewrite)
  (equal (listp (repeat n e)) (lessp 0 n)))

(prove-lemma firstn-repeat-2 (rewrite)
  (equal (firstn n1 (repeat n2 e))
         (if (lessp n1 n2) (repeat n1 e) (repeat n2 e)))
  ((enable firstn)))

(prove-lemma nthcdr-repeat-2 (rewrite)
  (equal (nthcdr n1 (repeat n2 e))
         (if (lessp n2 n1) 0 (repeat (difference n2 n1) e)))
  ((enable nthcdr)))

(prove-lemma nth-repeat-2 (rewrite)
  (equal (nth n1 (repeat n2 e)) (if (lessp n1 n2) e 0))
  ((enable nth)))

(prove-lemma length-repeat-2 (rewrite)
  (equal (length (repeat n e)) (fix n)))

(disable listp-repeat-2)
(disable firstn-repeat-2)
(disable nthcdr-repeat-2)
(disable nth-repeat-2)
(disable length-repeat-2)

(prove-lemma fm9001-rt-history-versus-fm9001-interpreter nil
  (let
      ((numinstr (number-fm9001-instructions n s mat pc)))
    (implies
```

```
     (and
      (not (read-write-encounteredp s numinstr pc))
      (equal pc (nat-to-v 15 (reg-size)))
      (listp s)
      (lessp 0 mat)
      (lessp 0 n))
     (equal (last (fm9001-rt-history s mat (repeat n nil) loclist))
            (extract-locs loclist (fm9001-interpreter s pc numinstr)))))
  ((disable-theory t)
   (enable-theory ground-zero naturals)
   (enable last-as-nth fm9001-rt-history fm9001-interpreter lessp-sub1-hack2
           lessp-init-time-total-time equal-init-time-0 *1*fm9001-interpreter *1*repeat
           fm9001-rt nth nth-length-cons listp-fm9001-rt-history-repeat-helper
           equal-length-small fm9001-rt-repeat-helper-equiv listp-append
           listp-repeat-with-inputs read-write-encounteredp listp-repeat-2 firstn-repeat-2
           nth-repeat-2 nthcdr-repeat-2 length-repeat-2 length *1*nth length-nthcdr
           nth-repeat-with-inputs length-append nth-append-better length-repeat-with-inputs
           update-state-with-inputs *1*reg-size *1*nat-to-v
           nth-fm9001-rt-history-repeat-helper lessp-0-init-time number-fm9001-instructions
           listp-fm9001-rt-history-helper fm9001-rt-history-repeat-helper
           fm9001-rt-history-helper)
   (expand (number-fm9001-instructions n s mat (list t t t t))
           (fm9001-interpreter s (list t t t t) 1)
           (number-fm9001-instructions 1 0 mat
                                       (list t t t t))
           (fm9001-rt s '(nil) mat))
   (induct (fm9001-rt-history-versus-induct s n mat pc))))

))
```

E.2 "quiz-abstract.events"

```
(proveall "quiz-abstract"
'(
#|
 Copyright (C) 1995 by Matthew Wilding and Computational Logic, Inc.
 All Rights Reserved.

 This script is hereby placed in the public domain, and therefore unlimited
 editing and redistribution is permitted.

 NO WARRANTY

 Matthew Wilding and Computational Logic, Inc. PROVIDES ABSOLUTELY NO
 WARRANTY.  THE EVENT SCRIPT IS PROVIDED "AS IS" WITHOUT WARRANTY OF
 ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
 ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE.  THE ENTIRE RISK AS TO THE QUALITY AND
 PERFORMANCE OF THE SCRIPT IS WITH YOU.  SHOULD THE SCRIPT PROVE
 DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR
 CORRECTION.

 IN NO EVENT WILL Matthew Wilding and Computational Logic, Inc. BE
 LIABLE TO YOU FOR ANY DAMAGES, ANY LOST PROFITS, LOST MONIES, OR
 OTHER SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE
 USE OR INABILITY TO USE THIS SCRIPT (INCLUDING BUT NOT LIMITED TO
 LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY
 THIRD PARTIES), EVEN IF YOU HAVE ADVISED US OF THE POSSIBILITY OF
 SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.
```

```
This file contains the events that lead to the proof of the abstract
lemma for the quiz-show system.

|#

; We use the library resulting from the proof of the light-switch.
(note-lib "light-switch")

(compile-uncompiled-defns "temp")

;; We define the relevant behaviors
(defn button1-spec ()
  '(up
    ((up
      (and (equal signal1 0) (equal a1 'f))
      ((up () (true))
       (going-down ((c 1))
         (and (equal signal1 1)
              (and (equal a1 'depress) (lessp 40000 (c 2)))))))
     (going-down
      (and (or (equal signal1 0) (equal signal1 1))
           (lessp (c 1) 30000))
      ((going-down () (equal a1 'f)) (down () (true))))
     (down
      (and (equal signal1 1)  (equal a1 'f))
      ((down () (true))
       (going-up ((c 2))
         (and (equal signal1 0)
              (and (equal a1 'release) (lessp 40000 (c 1)))))))
     (going-up
      (and (or (equal signal1 0) (equal signal1 1))
           (lessp (c 2) 30000))
      ((going-up () (equal a1 'f)) (up () (true)))))))


(defn button2-spec ()
  '(up
    ((up
      (and (equal signal2 0) (equal a2 'f))
      ((up () (true))
       (going-down ((c 3))
         (and (equal signal2 1)
              (and (equal a2 'depress) (lessp 40000 (c 4)))))))
     (going-down
      (and (or (equal signal2 0) (equal signal2 1))
           (lessp (c 3) 30000))
      ((going-down () (equal a2 'f)) (down () (true))))
     (down
      (and (equal signal2 1)  (equal a2 'f))
      ((down () (true))
       (going-up ((c 4))
         (and (equal signal2 0)
              (and (equal a2 'release) (lessp 40000 (c 3)))))))
     (going-up
      (and (or (equal signal2 0) (equal signal2 1))
           (lessp (c 4) 30000))
      ((going-up () (equal a2 'f)) (up () (true)))))))

(defn quiz-program-spec ()
  '(read-b1
    ((read-b1
```

```
      (lessp (c 9) 50)
      ((read-b1 ()
         (and (equal (old b1) b1) (and (equal (old b2) b2) (and (equal (old s) s)
              (and (equal (old l1) l1) (equal (old l2) l2))))))
       (read-b2 ()
         (and (equal (old signal1) b1) (and (equal (old b2) b2)
              (and (equal (old s) s)
              (and (equal (old l1) l1) (equal (old l2) l2))))))))
     (read-b2
      (lessp (c 9) 100)
      ((read-b2 ()
         (and (equal (old b1) b1) (and (equal (old b2) b2) (and (equal (old s) s)
              (and (equal (old l1) l1) (equal (old l2) l2))))))
       (update-state ()
         (and (equal (old b1) b1) (and (equal (old signal2) b2) (and (equal (old s) s)
              (and (equal (old l1) l1) (equal (old l2) l2))))))))
     (update-state
      (lessp (c 9) 300)
      ((update-state ()
         (and (equal (old b1) b1) (and (equal (old b2) b2) (and (equal (old s) s)
              (and (equal (old l1) l1) (equal (old l2) l2))))))
       (update-l1 ()
         (and (equal (old b1) b1) (and (equal (old b2) b2)
              (and (equal s (if (or (equal (old s) 0) (equal (old s) 1))
                                 (if (equal b1 1) 1 (if (equal b2 1) 2 0))
                               (if (equal b2 1) 2 (if (equal b1 1) 1 0))))
              (and (equal (old l1) l1) (equal (old l2) l2))))))))
     (update-l1
      (lessp (c 9) 350)
      ((update-l1 ()
         (and (equal (old b1) b1) (and (equal (old b2) b2) (and (equal (old s) s)
              (and (equal (old l1) l1) (equal (old l2) l2))))))
       (update-l2 ()
         (and (equal (old b1) b1) (and (equal (old b2) b2) (and (equal s (old s))
              (and (equal l1 (if (equal s 1) 0 1)) (equal (old l2) l2)))))))))
     (update-l2
      (lessp (c 9) 400)
      ((update-l2 ()
         (and (equal (old b1) b1) (and (equal (old b2) b2) (and (equal (old s) s)
              (and (equal (old l1) l1) (equal (old l2) l2))))))
       (read-b1 ((c 9))
         (and (equal (old b1) b1) (and (equal (old b2) b2) (and (equal s (old s))
              (and (equal (old l1) l1) (equal l2 (if (equal s 2) 0 1))))))))))))

(defn quiz-spec ()
  '(none
    ((none
      (or (lessp (c 6) 40000) (or (lessp (c 8) 40000)
          (and (equal l1 1) (equal l2 1))))
      ((none () (and (equal a1 'f) (equal a2 'f)))
       (one ((c 5)) (and (equal a1 'depress) (equal a2 'f)))
       (two ((c 7)) (and (equal a1 'f) (equal a2 'depress)))
       (both ((c 5) (c 7)) (and (equal a1 'depress) (equal a2 'depress)))))
     (one
      (or (lessp (c 5) 40000) (or (lessp (c 8) 40000)
          (and (equal l1 0) (equal l2 1))))
      ((none ((c 6)) (and (equal a1 'release) (equal a2 'f)))
       (one () (and (equal a1 'f) (and (equal a2 'f)
                  (or (not (and (lessp 40000 (c 5)) (equal (old l1) 0)))
                      (equal l1 0)))))
       (two ((c 6) (c 7)) (and (equal a1 'release) (equal a2 'depress)))
       (both ((c 7)) (and (equal a1 'f) (and (equal a2 'depress)
```

```
                         (or (not (and (lessp 40000 (c 5)) (equal (old l1) 0)))
                             (equal l1 0)))))))
     (two
      (or (lessp (c 7) 40000) (or (lessp (c 6) 40000)
          (and (equal l1 1) (equal l2 0))))
      ((none ((c 8)) (and (equal a1 'f) (equal a2 'release)))
       (one ((c 5) (c 8)) (and (equal a1 'depress) (equal a2 'release)))
       (two () (and (equal a1 'f) (and (equal a2 'f)
                   (or (not (and (lessp 40000 (c 7)) (equal (old l2) 0)))
                       (equal l2 0)))))
       (both ((c 5)) (and (equal a1 'depress) (and (equal a2 'f)
                   (or (not (and (lessp 40000 (c 7)) (equal (old l2) 0)))
                       (equal l2 0)))))))
     (both
      (or (lessp (c 7) 40000) (or (lessp (c 5) 40000)
      (or (and (equal l1 1) (equal l2 0))
          (and (equal l1 0) (equal l2 1)))))
      ((none ((c 6) (c 8)) (and (equal a1 'release) (equal a2 'release)))
       (one ((c 8)) (and (equal a1 'f) (and (equal a2 'release)
                   (or (not (and (lessp 40000 (c 5)) (equal (old l1) 0)))
                       (equal l1 0)))))
       (two ((c 6)) (and (equal a1 'release) (and (equal a2 'f)
                   (or (not (and (lessp 40000 (c 7)) (equal (old l2) 0)))
                       (equal l2 0)))))
       (both () (and (equal a1 'f) (and (equal a2 'f)
               (and (or (not (and (lessp 40000 (c 5)) (equal (old l1) 0)))
                        (equal l1 0))
                    (or (not (and (lessp 40000 (c 7)) (equal (old l2) 0)))
                       (equal l2 0)))))))))))

(defn quiz-invariant-counter-correspondence (c1 c2 c3 c4 c5 c6 c7 c8)
  (and (equal c1 c5) (equal c2 c6) (equal c3 c7) (equal c4 c8)))

(defn quiz-invariant-button1-reflects-signal1 (p1 pp c1 c2 c9 b1)
  (and
   (implies
    (and (or (equal pp 'read-b2) (equal pp 'update-state))
         (lessp (plus c9 30000) c2) (equal p1 'up))
    (equal b1 0))

   (implies
    (and (or (equal pp 'read-b2) (equal pp 'update-state))
         (lessp (plus c9 30000) c1) (equal p1 'down))
    (equal b1 1))))

(defn quiz-invariant-button2-reflects-signal2 (p2 pp c3 c4 c9 b2)
  (and
   (implies
    (and (equal pp 'update-state)
         (lessp (plus c9 30000) c4) (equal p2 'up))
    (equal b2 0))

   (implies
    (and (equal pp 'update-state)
         (lessp (plus c9 30000) c3) (equal p2 'down))
    (equal b2 1))))

(defn quiz-invariant-s-reflects-signal1 (p1 pp c1 c2 c9 s)
  (and
   (implies
    (and (or (equal pp 'update-l1) (equal pp 'update-l2))
         (lessp (plus c9 30000) c1) (equal p1 'down))
```

```
   (or (equal s 1) (equal s 2)))

  (implies
   (and (or (equal pp 'read-b1) (equal pp 'read-b2) (equal pp 'update-state))
        (lessp (plus 30400 c9) c1) (equal p1 'down))
   (or (equal s 1) (equal s 2)))

  (implies
   (and (or (equal pp 'update-l1) (equal pp 'update-l2))
        (lessp (plus c9 30000) c2) (equal p1 'up))
   (or (equal s 0) (equal s 2)))

  (implies
   (and (or (equal pp 'read-b1) (equal pp 'read-b2) (equal pp 'update-state))
        (lessp (plus 30400 c9) c2) (equal p1 'up))
   (or (equal s 0) (equal s 2)))))

(defn quiz-invariant-s-reflects-signal2 (p2 pp c3 c4 c9 s)
 (and
  (implies
   (and (or (equal pp 'update-l1) (equal pp 'update-l2))
        (lessp (plus c9 30000) c3) (equal p2 'down))
   (or (equal s 1) (equal s 2)))

  (implies
   (and (or (equal pp 'read-b1) (equal pp 'read-b2) (equal pp 'update-state))
        (lessp (plus 30400 c9) c3) (equal p2 'down))
   (or (equal s 1) (equal s 2)))

  (implies
   (and (or (equal pp 'update-l1) (equal pp 'update-l2))
        (lessp (plus c9 30000) c4) (equal p2 'up))
   (or (equal s 0) (equal s 1)))

  (implies
   (and (or (equal pp 'read-b1) (equal pp 'read-b2) (equal pp 'update-state))
        (lessp (plus 30400 c9) c4) (equal p2 'up))
   (or (equal s 0) (equal s 1)))))


(defn quiz-invariant-l1-reflects-signals (p1 p2 pp c1 c2 c3 c4 c9 s l1)
 (and
  (implies
   (and (equal pp 'update-l2) (equal p1 'up) (lessp (plus 30000 c9) c2))
   (equal l1 1))

  (implies
   (and (equal p1 'up) (lessp (plus 30400 c9) c2))
   (equal l1 1))

  (implies
   (and (equal pp 'update-l2) (equal p1 'down) (equal p2 'up)
        (lessp (plus 30000 c9) c1) (lessp (plus 30000 c9) c4))
   (equal l1 0))

  (implies
   (and (equal p1 'down) (equal p2 'up)
        (lessp (plus 30400 c9) c1) (lessp (plus 30400 c9) c4))
   (equal l1 0))

  (implies
   (and (equal pp 'update-l2) (equal p1 'down) (equal p2 'down)
```

```
         (lessp (plus 30000 c9) c1) (lessp (plus 30000 c9) c3))
    (equal l1 (if (equal s 1) 0 1)))

   (implies
    (and (equal p1 'down) (equal p2 'down)
         (lessp (plus 30400 c9) c1) (lessp (plus 30400 c9) c3))
    (equal l1 (if (equal s 1) 0 1)))))

(defn quiz-invariant-l2-reflects-signals (p1 p2 pp c1 c2 c3 c4 c9 s l2)
  (and
   (implies
    (and (equal p2 'up) (lessp (plus 30400 c9) c4))
    (equal l2 1))

   (implies
    (and (equal p2 'down) (equal p1 'up)
         (lessp (plus 30400 c9) c3) (lessp (plus 30400 c9) c2))
    (equal l2 0))

   (implies
    (and (equal p2 'down) (equal p1 'down)
         (lessp (plus 30400 c9) c3) (lessp (plus 30400 c9) c1))
    (equal l2 (if (equal s 2) 0 1)))))

(defn quiz-invariant-l1-means-s (p1 pp c1 c9 s l1)
  (and
   (implies
    (and (equal l1 0) (equal p1 'down) (equal pp 'update-l2)
         (lessp (plus 30000 c9) c1))
    (equal s 1))
   (implies
    (and (equal l1 0) (equal p1 'down) (lessp (plus 30400 c9) c1))
    (equal s 1))))

(defn quiz-invariant-l2-means-s (p2 pp c3 c9 s l2)
  (implies
   (and (equal l2 0) (equal p2 'down) (lessp (plus 30400 c9) c3))
   (equal s 2)))

(disable button1-spec)
(disable button2-spec)
(disable quiz-program-spec)
(disable quiz-spec)

(prove-lemma assoc-constant-quiz-program-spec (rewrite)
  (and
   (equal (assoc 'read-b1 (cadr (quiz-program-spec)))
          '(read-b1
           (lessp (c 9) 50)
           ((read-b1 ()
                     (and (equal (old b1) b1) (and (equal (old b2) b2) (and (equal (old s) s)
              (and (equal (old l1) l1) (equal (old l2) l2))))))
            (read-b2 ()
                     (and (equal (old signal1) b1) (and (equal (old b2) b2)
                     (and (equal (old s) s)
              (and (equal (old l1) l1) (equal (old l2) l2)))))))))
   (equal (assoc 'read-b2 (cadr (quiz-program-spec)))
     '(read-b2
      (lessp (c 9) 100)
      ((read-b2 ()
         (and (equal (old b1) b1) (and (equal (old b2) b2) (and (equal (old s) s)
              (and (equal (old l1) l1) (equal (old l2) l2))))))
```

```
       (update-state ()
         (and (equal (old b1) b1) (and (equal (old signal2) b2) (and (equal (old s) s)
              (and (equal (old l1) l1) (equal (old l2) l2)))))))))
   (equal (assoc 'update-state (cadr (quiz-program-spec)))
     '(update-state
      (lessp (c 9) 300)
      ((update-state ()
         (and (equal (old b1) b1) (and (equal (old b2) b2) (and (equal (old s) s)
              (and (equal (old l1) l1) (equal (old l2) l2))))))
       (update-l1 ()
         (and (equal (old b1) b1) (and (equal (old b2) b2)
              (and (equal s (if (or (equal (old s) 0) (equal (old s) 1))
                                (if (equal b1 1) 1 (if (equal b2 1) 2 0))
                              (if (equal b2 1) 2 (if (equal b1 1) 1 0))))
              (and (equal (old l1) l1) (equal (old l2) l2)))))))))
   (equal (assoc 'update-l1 (cadr (quiz-program-spec)))
     '(update-l1
      (lessp (c 9) 350)
      ((update-l1 ()
         (and (equal (old b1) b1) (and (equal (old b2) b2) (and (equal (old s) s)
              (and (equal (old l1) l1) (equal (old l2) l2))))))
       (update-l2 ()
         (and (equal (old b1) b1) (and (equal (old b2) b2) (and (equal s (old s))
              (and (equal l1 (if (equal s 1) 0 1)) (equal (old l2) l2)))))))))
   (equal (assoc 'update-l2 (cadr (quiz-program-spec)))
     '(update-l2
      (lessp (c 9) 400)
      ((update-l2 ()
         (and (equal (old b1) b1) (and (equal (old b2) b2) (and (equal (old s) s)
              (and (equal (old l1) l1) (equal (old l2) l2))))))
       (read-b1 ((c 9))
         (and (equal (old b1) b1) (and (equal (old b2) b2) (and (equal s (old s))
              (and (equal (old l1) l1) (equal l2 (if (equal s 2) 0 1)))))))))))

(prove-lemma assoc-constant-button1-spec (rewrite)
  (and
   (equal (assoc 'up (cadr (button1-spec)))
          '(up
             (and (equal signal1 0) (equal a1 'f))
             ((up () (true))
              (going-down ((c 1))
                          (and (equal signal1 1)
                               (and (equal a1 'depress) (lessp 40000 (c 2))))))))
   (equal (assoc 'going-down (cadr (button1-spec)))
          '(going-down
            (and (or (equal signal1 0) (equal signal1 1))
                 (lessp (c 1) 30000))
            ((going-down () (equal a1 'f)) (down () (true)))))
   (equal (assoc 'down (cadr (button1-spec)))
          '(down
            (and (equal signal1 1)  (equal a1 'f))
            ((down () (true))
             (going-up ((c 2))
                       (and (equal signal1 0)
                            (and (equal a1 'release) (lessp 40000 (c 1))))))))
   (equal (assoc 'going-up (cadr (button1-spec)))
          '(going-up
            (and (or (equal signal1 0) (equal signal1 1))
                 (lessp (c 2) 30000))
            ((going-up () (equal a1 'f)) (up () (true)))))))

(prove-lemma assoc-constant-button2-spec (rewrite)
```

```
  (and
   (equal (assoc 'up (cadr (button2-spec)))
          '(up
             (and (equal signal2 0) (equal a2 'f))
             ((up () (true))
              (going-down ((c 3))
                          (and (equal signal2 1)
                               (and (equal a2 'depress) (lessp 40000 (c 4)))))))))
   (equal (assoc 'going-down (cadr (button2-spec)))
          '(going-down
            (and (or (equal signal2 0) (equal signal2 1))
                 (lessp (c 3) 30000))
            ((going-down () (equal a2 'f)) (down () (true)))))
   (equal (assoc 'down (cadr (button2-spec)))
          '(down
            (and (equal signal2 1)  (equal a2 'f))
            ((down () (true))
             (going-up ((c 4))
                       (and (equal signal2 0)
                            (and (equal a2 'release) (lessp 40000 (c 3))))))))
   (equal (assoc 'going-up (cadr (button2-spec)))
          '(going-up
            (and (or (equal signal2 0) (equal signal2 1))
                 (lessp (c 4) 30000))
            ((going-up () (equal a2 'f)) (up () (true)))))))


;; We embed each of our specifications in Nqthm directly so that we
;; don't have the inefficiency associated with opening up our behavior
;; semantics all the time during our proofs.

(defn button1-spec-embedded-path-helper (h c p)
  (if (listp h)
      (let ((signal1 (cadr (assoc 'signal1 (car h))))
            (a1 (cadr (assoc 'a1 (car h))))
            (c1 (cadr (assoc '(c 1) c)))
            (c2 (cadr (assoc '(c 2) c))))
        (if (nlistp (cdr h))
            (case (car p)
                  (up (and (equal signal1 0) (equal a1 'f)))
                  (going-down (and (or (equal signal1 0) (equal signal1 1))
                                   (lessp c1 30000)))
                  (down (and (equal signal1 1) (equal a1 'f)))
                  (going-up (and (or (equal signal1 0) (equal signal1 1))
                                 (lessp c2 30000)))
                  (otherwise f))
          (if
              (case (car p)
                    (up
                     (if (equal (cadr p) 'up)
                         (and (equal signal1 0) (equal a1 'f))
                       (if (equal (cadr p) 'going-down)
                           (and
                            (equal signal1 0) (equal a1 'f)
                            (equal (cadr (assoc 'a1 (cadr h))) 'depress)
                            (not (lessp c2 40000))
                            (equal (cadr (assoc 'signal1 (cadr h))) 1))
                         f)))
                    (going-down
                     (if (equal (cadr p) 'going-down)
                         (and (or (equal signal1 0) (equal signal1 1))
                              (lessp c1 30000) (equal (cadr (assoc 'a1 (cadr h))) 'f))
```

```
                    (if (equal (cadr p) 'down)
                        (and (or (equal signal1 0) (equal signal1 1))
                             (lessp c1 30000))
                      f)))
                 (down
                  (if (equal (cadr p) 'down)
                      (and (equal signal1 1) (equal a1 'f))
                    (if (equal (cadr p) 'going-up)
                        (and
                         (equal (cadr (assoc 'a1 (cadr h))) 'release)
                         (not (lessp c1 40000))
                         (equal (cadr (assoc 'signal1 (cadr h))) 0)
                         (equal signal1 1) (equal a1 'f))
                      f)))
                 (going-up
                  (if (equal (cadr p) 'going-up)
                      (and (or (equal signal1 0) (equal signal1 1))
                           (lessp c2 30000)
                           (equal (cadr (assoc 'a1 (cadr h))) 'f))
                    (if (equal (cadr p) 'up)
                        (and (or (equal signal1 0) (equal signal1 1))
                             (lessp c2 30000))
                      f)))
                 (otherwise f))
           (button1-spec-embedded-path-helper
            (cdr h)
            (update-counter-alist
             c
             (if (and (equal (car p) 'up) (equal (cadr p) 'going-down))
                 '((c 1))
               (if (and (equal (car p) 'down) (equal (cadr p) 'going-up))
                   '((c 2))
                 nil)))
            (cdr p))
          f)))
    t))

(defn button1-spec-embedded-path-helper-first2 (h c p)
  (if (listp h)
      (let ((signal1 (cadr (assoc 'signal1 (car h))))
            (newsignal1 (cadr (assoc 'signal1 (cadr h))))
            (a1 (cadr (assoc 'a1 (car h))))
            (newa1 (cadr (assoc 'a1 (cadr h))))
            (c1 (cadr (assoc '(c 1) c)))
            (c2 (cadr (assoc '(c 2) c))))
        (if (nlistp (cdr h))
            (case (car p)
                  (up (and (equal signal1 0) (equal a1 'f)))
                  (going-down (and (or (equal signal1 0) (equal signal1 1))
                                   (lessp c1 30000)))
                  (down (and (equal signal1 1) (equal a1 'f)))
                  (going-up (and (or (equal signal1 0) (equal signal1 1))
                                 (lessp c2 30000)))
                  (otherwise f))
          (case (car p)
                (up
                 (if (equal (cadr p) 'up)
                     (and (equal signal1 0) (equal a1 'f)
                          (equal newsignal1 0) (equal newa1 'f))
                   (if (equal (cadr p) 'going-down)
                       (and
                        (equal signal1 0) (equal a1 'f)
```

```
                     (equal (cadr (assoc 'a1 (cadr h))) 'depress)
                     (not (lessp c2 40000))
                     (equal (cadr (assoc 'signal1 (cadr h))) 1))
                  f)))
              (going-down
               (if (equal (cadr p) 'going-down)
                   (and (or (equal signal1 0) (equal signal1 1))
                        (lessp c1 29999) (equal (cadr (assoc 'a1 (cadr h))) 'f)
                        (or (equal newsignal1 0) (equal newsignal1 1)))
                 (if (equal (cadr p) 'down)
                     (and (or (equal signal1 0) (equal signal1 1))
                          (equal newsignal1 1) (equal newa1 'f)
                          (lessp c1 30000))
                   f)))
              (down
               (if (equal (cadr p) 'down)
                   (and (equal signal1 1) (equal a1 'f)
                        (equal newsignal1 1) (equal newa1 'f))
                 (if (equal (cadr p) 'going-up)
                     (and
                      (equal (cadr (assoc 'a1 (cadr h))) 'release)
                      (not (lessp c1 40000))
                      (equal (cadr (assoc 'signal1 (cadr h))) 0)
                      (equal signal1 1) (equal a1 'f))
                   f)))
              (going-up
               (if (equal (cadr p) 'going-up)
                   (and (or (equal signal1 0) (equal signal1 1))
                        (lessp c2 29999)
                        (equal (cadr (assoc 'a1 (cadr h))) 'f)
                        (or (equal newsignal1 0) (equal newsignal1 1)))
                 (if (equal (cadr p) 'up)
                     (and (or (equal signal1 0) (equal signal1 1))
                          (equal newsignal1 0) (equal newa1 'f)
                          (lessp c2 30000))
                   f)))
              (otherwise f))))
   t))

(prove-lemma button1-spec-embedded-means-first2 nil
  (implies
   (button1-spec-embedded-path-helper h c p)
   (button1-spec-embedded-path-helper-first2 h c p)))

(prove-lemma satisfiesp-with-path-assoc-helper-simple-open (rewrite)
  (and
   (implies
    (and
     (not (listp p))
     (all-cars-litatoms bs))
    (equal (satisfiesp-with-path-assoc-helper pr tr h c bs p) (not (listp h))))
  (implies
   (and
    (not (listp (cdr p)))
    (all-cars-litatoms bs))
   (equal
    (satisfiesp-with-path-assoc-helper pr tr h c bs p)
    (if (not (listp h))
        t
      (and
       (not (listp (cdr h)))
       (assoc (car p) bs)
```

```
      (my-eval pr (append c (car h)) nil))))))
  ((expand (satisfiesp-with-path-assoc-helper pr tr h c bs p))))

(prove-lemma all-cars-cadr-button1-spec (rewrite)
  (all-cars-litatoms (cadr (button1-spec))))

(prove-lemma assoc-button1-iff (rewrite)
  (iff
   (assoc x (cadr (button1-spec)))
   (or (equal x 'up) (equal x 'going-down) (equal x 'going-up) (equal x 'down))))

(prove-lemma assoc-assoc-button1-iff (rewrite)
  (iff
   (assoc y (caddr (assoc x (cadr (button1-spec)))))
   (or
    (and (equal x 'up) (or (equal y 'up) (equal y 'going-down)))
    (and (equal x 'going-down) (or (equal y 'going-down) (equal y 'down)))
    (and (equal x 'down) (or (equal y 'going-up) (equal y 'down)))
    (and (equal x 'going-up) (or (equal y 'going-up) (equal y 'up))))))

(prove-lemma button1-spec-s-w-p-a-embedded (rewrite)
  (implies
   (and
    (equal bs (cadr (button1-spec)))
    (equal pr (state-pred (assoc (car p) bs)))
    (equal tr (state-trans-list (assoc (car p) bs)))
    (assoc '(c 1) c)
    (assoc '(c 2) c)
    (no-litatom-cars c))
   (equal
    (satisfiesp-with-path-assoc-helper pr tr h c bs p)
    (button1-spec-embedded-path-helper h c p)))
  ((induct (satisfiesp-with-path-assoc-helper pr tr h c bs p))
   (disable-theory t)
   (expand (button1-spec-embedded-path-helper h c p))
   (enable-theory ground-zero naturals)
   (enable button1-spec-embedded-path-helper satisfiesp-first-trans assoc-append
           assoc-button1-iff my-eval no-litatom-cars-update-assoc
           assoc-litatom-no-litatom-cars assoc-update-counter-alist
           assoc-update-counter-alist-iff satisfiesp-with-path-assoc-helper-simple-open
           trans-to trans-resets trans-pred all-cars-litatoms-assoc-trans-list-open
           all-transitions-litatoms my-eval-open all-cars-cadr-button1-spec all-cars-litatoms
           car-assoc assoc-constant-button1-spec state-name state-trans-list state-pred
           assoc-assoc-button1-iff open-satisfiesp-with-path-assoc satisfiesp-base-case
           open-satisfiesp-with-path-assoc2)))

(prove-lemma button1-spec-s-w-p-embedded (rewrite)
  (implies
   (and
    (equal pr (state-pred (assoc (car p) (cadr (button1-spec)))))
    (equal tr (state-trans-list (assoc (car p) (cadr (button1-spec)))))
    (assoc '(c 1) c)
    (assoc '(c 2) c)
    (no-litatom-cars c))
   (equal
    (satisfiesp-with-path-helper pr tr h c (cadr (button1-spec)) p)
    (button1-spec-embedded-path-helper h c p)))
  ((use (satisfiesp-with-path-assoc-same-helper
         (pred pr) (trans1 tr) (trans2 tr) (counters c)
         (b (cadr (button1-spec))) (path p)))
   (disable-theory t)
   (enable-theory ground-zero)
```

```
   (enable button1-spec-s-w-p-a-embedded all-transitions-litatoms button1-spec
           all-cars-litatoms-assoc-trans-list-open all-cars-litatoms state-trans-list)))

(prove-lemma car-find-path (rewrite)
  (implies
   (and
    (satisfiesp h b)
    (assoc (behavior-initial b) (behavior-state-list b))
    (all-cars-litatoms (caddr (assoc (car b) (cadr b))))
    (all-transitions-litatoms (cadr b)))
   (equal (car (find-path h b)) (if (listp h) (behavior-initial b) 0))))

(prove-lemma button1-spec-s-helper-from-embedded (rewrite)
  (implies
   (satisfiesp h (button1-spec))
   (button1-spec-embedded-path-helper h '(((c 1) 0) ((c 2) 0)) (find-path h (button1-spec))))
  ((use (button1-spec-s-w-p-embedded
         (p (find-path h (button1-spec))) (c '(((c 1) 0) ((c 2) 0)))
         (pr (state-pred (assoc (car (find-path h (button1-spec))) (cadr (button1-spec)))))
         (tr (state-trans-list (assoc (car (find-path h (button1-spec)))
                                      (cadr (button1-spec))))))
        (satisfiesp-means-with-find-path (b (button1-spec))))
   (disable-theory t)
   (enable-theory ground-zero)
   (enable satisfiesp button1-spec satisfiesp-with-path satisfiesp-with-path-helper
           state-pred all-cars-litatoms state-name behavior-state-list behavior-initial
           state-trans-list car-find-path no-litatom-cars all-transitions-uniquep
           all-transitions-litatoms counters-of-state-list make-list-alist cars-uniquep
           counters-of-term counters-of-translist counters-of-state counters-of-trans
           trans-resets trans-pred member-cars set-union)))


(defn button2-spec-embedded-path-helper (h c p)
  (if (listp h)
      (let ((signal2 (cadr (assoc 'signal2 (car h))))
            (a2 (cadr (assoc 'a2 (car h))))
            (c3 (cadr (assoc '(c 3) c)))
            (c4 (cadr (assoc '(c 4) c))))
        (if (nlistp (cdr h))
         (case (car p)
           (up (and (equal signal2 0) (equal a2 'f)))
           (going-down (and (or (equal signal2 0) (equal signal2 1))
                            (lessp c3 30000)))
           (down (and (equal signal2 1) (equal a2 'f)))
           (going-up (and (or (equal signal2 0) (equal signal2 1))
                            (lessp c4 30000)))
           (otherwise f))
         (if
         (case (car p)
           (up
            (if (equal (cadr p) 'up)
                (and (equal signal2 0) (equal a2 'f))
                  (if (equal (cadr p) 'going-down)
                      (and
                       (equal signal2 0) (equal a2 'f)
                       (equal (cadr (assoc 'a2 (cadr h))) 'depress)
                       (not (lessp c4 40000))
                       (equal (cadr (assoc 'signal2 (cadr h))) 1))
                    f)))
           (going-down
            (if (equal (cadr p) 'going-down)
                (and (or (equal signal2 0) (equal signal2 1))
```

```
                (lessp c3 30000) (equal (cadr (assoc 'a2 (cadr h))) 'f))
            (if (equal (cadr p) 'down)
                (and (or (equal signal2 0) (equal signal2 1))
                         (lessp c3 30000))
                 f)))
         (down
          (if (equal (cadr p) 'down)
              (and (equal signal2 1) (equal a2 'f))
            (if (equal (cadr p) 'going-up)
                (and
                 (equal (cadr (assoc 'a2 (cadr h))) 'release)
                 (not (lessp c3 40000))
                 (equal (cadr (assoc 'signal2 (cadr h))) 0)
                 (equal signal2 1) (equal a2 'f))
              f)))
         (going-up
          (if (equal (cadr p) 'going-up)
              (and (or (equal signal2 0) (equal signal2 1))
                   (lessp c4 30000)
                   (equal (cadr (assoc 'a2 (cadr h))) 'f))
            (if (equal (cadr p) 'up)
                (and (or (equal signal2 0) (equal signal2 1))
                     (lessp c4 30000))
              f)))
         (otherwise f))
           (button2-spec-embedded-path-helper
            (cdr h)
            (update-counter-alist
             c
             (if (and (equal (car p) 'up) (equal (cadr p) 'going-down))
                 '((c 3))
               (if (and (equal (car p) 'down) (equal (cadr p) 'going-up))
                   '((c 4))
                 nil)))
            (cdr p))
         f)))
   t))

(defn button2-spec-embedded-path-helper-first2 (h c p)
  (if (listp h)
      (let ((signal2 (cadr (assoc 'signal2 (car h))))
            (newsignal2 (cadr (assoc 'signal2 (cadr h))))
            (a2 (cadr (assoc 'a2 (car h))))
            (newa2 (cadr (assoc 'a2 (cadr h))))
            (c3 (cadr (assoc '(c 3) c)))
            (c4 (cadr (assoc '(c 4) c))))
        (if (nlistp (cdr h))
         (case (car p)
           (up (and (equal signal2 0) (equal a2 'f)))
           (going-down (and (or (equal signal2 0) (equal signal2 1))
                            (lessp c3 30000)))
           (down (and (equal signal2 1) (equal a2 'f)))
           (going-up (and (or (equal signal2 0) (equal signal2 1))
                          (lessp c4 30000)))
           (otherwise f))
         (case (car p)
           (up
            (if (equal (cadr p) 'up)
                (and (equal signal2 0) (equal a2 'f)
                     (equal newsignal2 0) (equal newa2 'f))
                  (if (equal (cadr p) 'going-down)
                      (and
```

```
                    (equal signal2 0) (equal a2 'f)
                    (equal (cadr (assoc 'a2 (cadr h))) 'depress)
                    (not (lessp c4 40000))
                    (equal (cadr (assoc 'signal2 (cadr h))) 1))
                 f)))
          (going-down
           (if (equal (cadr p) 'going-down)
               (and (or (equal signal2 0) (equal signal2 1))
                    (lessp c3 29999) (equal (cadr (assoc 'a2 (cadr h))) 'f)
                    (or (equal newsignal2 0) (equal newsignal2 1)))
             (if (equal (cadr p) 'down)
                 (and (or (equal signal2 0) (equal signal2 1))
                      (equal newsignal2 1) (equal newa2 'f)
                      (lessp c3 30000))
                 f)))
          (down
           (if (equal (cadr p) 'down)
               (and (equal signal2 1) (equal a2 'f)
                    (equal newsignal2 1) (equal newa2 'f))
             (if (equal (cadr p) 'going-up)
                 (and
                  (equal (cadr (assoc 'a2 (cadr h))) 'release)
                  (not (lessp c3 40000))
                  (equal (cadr (assoc 'signal2 (cadr h))) 0)
                  (equal signal2 1) (equal a2 'f))
               f)))
          (going-up
           (if (equal (cadr p) 'going-up)
               (and (or (equal signal2 0) (equal signal2 1))
                    (lessp c4 29999)
                    (equal (cadr (assoc 'a2 (cadr h))) 'f)
                    (or (equal newsignal2 0) (equal newsignal2 1)))
             (if (equal (cadr p) 'up)
                 (and (or (equal signal2 0) (equal signal2 1))
                      (lessp c4 30000)
                      (equal newsignal2 0) (equal newa2 'f))
               f)))
          (otherwise f))))
    t))

(prove-lemma button2-spec-embedded-means-first2 nil
  (implies
   (button2-spec-embedded-path-helper h c p)
   (button2-spec-embedded-path-helper-first2 h c p)))

(prove-lemma all-cars-cadr-button2-spec (rewrite)
  (all-cars-litatoms (cadr (button2-spec))))

(prove-lemma assoc-button2-iff (rewrite)
  (iff
   (assoc x (cadr (button2-spec)))
   (or (equal x 'up) (equal x 'going-down) (equal x 'going-up) (equal x 'down))))

(prove-lemma assoc-assoc-button2-iff (rewrite)
  (iff
   (assoc y (caddr (assoc x (cadr (button2-spec)))))
   (or
    (and (equal x 'up) (or (equal y 'up) (equal y 'going-down)))
    (and (equal x 'going-down) (or (equal y 'going-down) (equal y 'down)))
    (and (equal x 'down) (or (equal y 'going-up) (equal y 'down)))
    (and (equal x 'going-up) (or (equal y 'going-up) (equal y 'up))))))
```

```
(prove-lemma button2-spec-s-w-p-a-embedded (rewrite)
  (implies
   (and
    (equal bs (cadr (button2-spec)))
    (equal pr (state-pred (assoc (car p) bs)))
    (equal tr (state-trans-list (assoc (car p) bs)))
    (assoc '(c 3) c)
    (assoc '(c 4) c)
    (no-litatom-cars c))
   (equal
    (satisfiesp-with-path-assoc-helper pr tr h c bs p)
    (button2-spec-embedded-path-helper h c p)))
  ((induct (satisfiesp-with-path-assoc-helper pr tr h c bs p))
   (disable-theory t)
   (expand (button2-spec-embedded-path-helper h c p))
   (enable-theory ground-zero naturals)
   (enable button2-spec-embedded-path-helper satisfiesp-first-trans assoc-append
           assoc-button2-iff assoc-assoc-button2-iff my-eval assoc-litatom-no-litatom-cars
           no-litatom-cars-update-assoc assoc-update-counter-alist
           assoc-update-counter-alist-iff satisfiesp-with-path-assoc-helper-simple-open
           trans-to trans-resets trans-pred all-cars-litatoms-assoc-trans-list-open
           all-transitions-litatoms my-eval-open all-cars-cadr-button2-spec all-cars-litatoms
           car-assoc assoc-constant-button2-spec state-name state-trans-list state-pred
           open-satisfiesp-with-path-assoc satisfiesp-base-case
           open-satisfiesp-with-path-assoc2)))

(prove-lemma button2-spec-s-w-p-embedded (rewrite)
  (implies
   (and
    (equal pr (state-pred (assoc (car p) (cadr (button2-spec)))))
    (equal tr (state-trans-list (assoc (car p) (cadr (button2-spec)))))
    (assoc '(c 3) c)
    (assoc '(c 4) c)
    (no-litatom-cars c))
   (equal
    (satisfiesp-with-path-helper pr tr h c (cadr (button2-spec)) p)
    (button2-spec-embedded-path-helper h c p)))
  ((use (satisfiesp-with-path-assoc-same-helper
         (pred pr) (trans1 tr) (trans2 tr) (counters c)
         (b (cadr (button2-spec))) (path p)))
   (disable-theory t)
   (enable-theory ground-zero)
   (enable button2-spec-s-w-p-a-embedded all-transitions-litatoms button2-spec
           all-cars-litatoms-assoc-trans-list-open all-cars-litatoms state-trans-list)))

(prove-lemma button2-spec-s-helper-from-embedded (rewrite)
  (implies
   (satisfiesp h (button2-spec))
   (button2-spec-embedded-path-helper h '(((c 3) 0) ((c 4) 0)) (find-path h (button2-spec))))
  ((use (button2-spec-s-w-p-embedded
         (p (find-path h (button2-spec))) (c '(((c 3) 0) ((c 4) 0)))
         (pr (state-pred (assoc (car (find-path h (button2-spec))) (cadr (button2-spec)))))
         (tr (state-trans-list (assoc (car (find-path h (button2-spec)))
                                      (cadr (button2-spec))))))
        (satisfiesp-means-with-find-path (b (button2-spec))))
   (disable-theory t)
   (enable-theory ground-zero)
   (enable satisfiesp button2-spec satisfiesp-with-path satisfiesp-with-path-helper
           state-pred all-cars-litatoms state-name behavior-state-list behavior-initial
           state-trans-list car-find-path no-litatom-cars all-transitions-uniquep
           all-transitions-litatoms counters-of-state-list make-list-alist cars-uniquep
           counters-of-term counters-of-translist counters-of-state counters-of-trans
```

```
          trans-resets trans-pred member-cars set-union)))

(prove-lemma all-cars-cadr-quiz-program-spec (rewrite)
  (all-cars-litatoms (cadr (quiz-program-spec))))

(prove-lemma assoc-quiz-program-spec-iff (rewrite)
  (iff
   (assoc x (cadr (quiz-program-spec)))
   (or (equal x 'read-b1) (equal x 'read-b2) (equal x 'update-state)
       (equal x 'update-l1) (equal x 'update-l2))))

(prove-lemma listp-caddr-assoc-quiz-program-spec (rewrite)
  (equal (listp (cadr (assoc y (caddr (assoc x (cadr (quiz-program-spec)))))))
         (and (equal x 'update-l2) (equal y 'read-b1))))

(prove-lemma equal-update-counter-alist-nlistp (rewrite)
  (implies
   (and
    (not (listp x))
    (not (listp y)))
   (equal
    (equal (update-counter-alist c x) (update-counter-alist c y))
    t)))

(defn quiz-program-spec-embedded-path-helper (h c p)
  (if (listp h)
      (let ((oldb1 (cadr (assoc 'b1 (car h))))
            (oldb2 (cadr (assoc 'b2 (car h))))
            (olds (cadr (assoc 's (car h))))
            (oldl1 (cadr (assoc 'l1 (car h))))
            (oldl2 (cadr (assoc 'l2 (car h))))
            (c9 (cadr (assoc '(c 9) c))))
        (if (nlistp (cdr h))
            (case (car p)
              (read-b1 (lessp c9 50))
              (read-b2 (lessp c9 100))
              (update-state (lessp c9 300))
              (update-l1 (lessp c9 350))
              (update-l2 (lessp c9 400))
              (otherwise f))
          (let ((b1 (cadr (assoc 'b1 (cadr h))))
                (b2 (cadr (assoc 'b2 (cadr h))))
                (s (cadr (assoc 's (cadr h))))
                (l1 (cadr (assoc 'l1 (cadr h))))
                (l2 (cadr (assoc 'l2 (cadr h)))))
            (if
             (case (car p)
               (read-b1
                (if (equal (cadr p) 'read-b1)
                 (and (lessp c9 49) (equal b1 oldb1) (equal b2 oldb2)
                      (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                 (if (equal (cadr p) 'read-b2)
                     (and
                      (lessp c9 50)
                      (equal b1 (cadr (assoc 'signal1 (car h)))) (equal b2 oldb2)
                      (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                   f)))
               (read-b2
                (if (equal (cadr p) 'read-b2)
                    (and (lessp c9 99) (equal b1 oldb1) (equal b2 oldb2)
                         (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                  (if (equal (cadr p) 'update-state)
```

```
                    (and
                     (lessp c9 100)
                     (equal b1 oldb1) (equal b2 (cadr (assoc 'signal2 (car h))))
                     (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                  f)))
             (update-state
              (if (equal (cadr p) 'update-state)
                  (and (lessp c9 299) (equal b1 oldb1) (equal b2 oldb2)
                    (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                (if (equal (cadr p) 'update-l1)
                    (and
                     (lessp c9 300) (equal b1 oldb1) (equal b2 oldb2)
                     (equal s (if (or (equal olds 0) (equal olds 1))
                                  (if (equal b1 1) 1 (if (equal b2 1) 2 0))
                                (if (equal b2 1) 2 (if (equal b1 1) 1 0))))
                     (equal l1 oldl1) (equal l2 oldl2))
                  f)))
             (update-l1
              (if (equal (cadr p) 'update-l1)
                  (and (lessp c9 349) (equal b1 oldb1) (equal b2 oldb2)
                       (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                (if (equal (cadr p) 'update-l2)
                    (and
                     (lessp c9 350)
                     (equal b1 oldb1) (equal b2 oldb2) (equal s olds)
                     (equal l1 (if (equal s 1) 0 1)) (equal l2 oldl2))
                  f)))
              (update-l2
               (if (equal (cadr p) 'update-l2)
                   (and  (lessp c9 399) (equal b1 oldb1) (equal b2 oldb2)
                    (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                 (if (equal (cadr p) 'read-b1)
                     (and
                      (lessp c9 400)
                      (equal b1 oldb1) (equal b2 oldb2) (equal s olds)
                      (equal l1 oldl1) (equal l2 (if (equal s 2) 0 1)))
                   f)))
              (otherwise f))
           (quiz-program-spec-embedded-path-helper
            (cdr h)
            (update-counter-alist
             c
             (if (and (equal (car p) 'update-l2) (equal (cadr p) 'read-b1))
                 '((c 9)) nil))
            (cdr p))
           f))))
    t))

(defn quiz-program-spec-embedded-path-helper-first2 (h c p)
  (if (listp h)
      (let ((oldb1 (cadr (assoc 'b1 (car h))))
            (oldb2 (cadr (assoc 'b2 (car h))))
            (olds (cadr (assoc 's (car h))))
            (oldl1 (cadr (assoc 'l1 (car h))))
            (oldl2 (cadr (assoc 'l2 (car h))))
            (c9 (cadr (assoc '(c 9) c))))
        (if (nlistp (cdr h))
            (case (car p)
              (read-b1 (lessp c9 50))
              (read-b2 (lessp c9 100))
              (update-state (lessp c9 300))
              (update-l1 (lessp c9 350))
```

```
           (update-l2 (lessp c9 400))
           (otherwise f))
       (let ((b1 (cadr (assoc 'b1 (cadr h))))
             (b2 (cadr (assoc 'b2 (cadr h))))
             (s (cadr (assoc 's (cadr h))))
             (l1 (cadr (assoc 'l1 (cadr h))))
             (l2 (cadr (assoc 'l2 (cadr h)))))
         (case (car p)
           (read-b1
            (if (equal (cadr p) 'read-b1)
             (and (lessp c9 50) (equal b1 oldb1) (equal b2 oldb2)
                  (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
             (if (equal (cadr p) 'read-b2)
                 (and
                  (lessp c9 50)
                  (equal b1 (cadr (assoc 'signal1 (car h)))) (equal b2 oldb2)
                  (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
               f)))
           (read-b2
            (if (equal (cadr p) 'read-b2)
                (and (lessp c9 100) (equal b1 oldb1) (equal b2 oldb2)
                     (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
              (if (equal (cadr p) 'update-state)
                  (and
                   (lessp c9 100)
                   (equal b1 oldb1) (equal b2 (cadr (assoc 'signal2 (car h))))
                   (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                f)))
           (update-state
            (if (equal (cadr p) 'update-state)
                (and (lessp c9 300) (equal b1 oldb1) (equal b2 oldb2)
                  (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
              (if (equal (cadr p) 'update-l1)
                  (and
                   (lessp c9 300) (equal b1 oldb1) (equal b2 oldb2)
                   (equal s (if (or (equal olds 0) (equal olds 1))
                                (if (equal b1 1) 1 (if (equal b2 1) 2 0))
                              (if (equal b2 1) 2 (if (equal b1 1) 1 0))))
                   (equal l1 oldl1) (equal l2 oldl2))
                f)))
           (update-l1
            (if (equal (cadr p) 'update-l1)
                (and (lessp c9 350) (equal b1 oldb1) (equal b2 oldb2)
                     (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
              (if (equal (cadr p) 'update-l2)
                  (and
                   (lessp c9 350)
                   (equal b1 oldb1) (equal b2 oldb2) (equal s olds)
                   (equal l1 (if (equal s 1) 0 1)) (equal l2 oldl2))
                f)))
            (update-l2
             (if (equal (cadr p) 'update-l2)
                 (and  (lessp c9 400) (equal b1 oldb1) (equal b2 oldb2)
                  (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
               (if (equal (cadr p) 'read-b1)
                   (and
                    (lessp c9 400)
                    (equal b1 oldb1) (equal b2 oldb2) (equal s olds)
                    (equal l1 oldl1) (equal l2 (if (equal s 2) 0 1)))
                 f)))
            (otherwise f)))))
t))
```

```
(prove-lemma quiz-program-spec-embedded-means-first2 nil
  (implies
   (quiz-program-spec-embedded-path-helper h c p)
   (quiz-program-spec-embedded-path-helper-first2 h c p)))

(prove-lemma quiz-program-spec-paths (rewrite)
  (iff
   (assoc y (caddr (assoc x (cadr (quiz-program-spec)))))
   (or
    (and (equal x 'read-b1) (or (equal y 'read-b1) (equal y 'read-b2)))
    (and (equal x 'read-b2) (or (equal y 'read-b2) (equal y 'update-state)))
    (and (equal x 'update-state) (or (equal y 'update-state) (equal y 'update-l1)))
    (and (equal x 'update-l1) (or (equal y 'update-l1) (equal y 'update-l2)))
    (and (equal x 'update-l2) (or (equal y 'update-l2) (equal y 'read-b1))))))

(prove-lemma quiz-program-spec-s-w-p-a-embedded (rewrite)
  (implies
   (and
    (equal bs (cadr (quiz-program-spec)))
    (equal pr (state-pred (assoc (car p) bs)))
    (equal tr (state-trans-list (assoc (car p) bs)))
    (assoc '(c 9) c)
    (no-litatom-cars c))
   (equal
    (satisfiesp-with-path-assoc-helper pr tr h c bs p)
    (quiz-program-spec-embedded-path-helper h c p)))
  ((induct (satisfiesp-with-path-assoc-helper pr tr h c bs p))
   (disable-theory t)
   (expand (quiz-program-spec-embedded-path-helper h c p))
   (enable-theory ground-zero naturals)
   (enable quiz-program-spec-embedded-path-helper satisfiesp-first-trans assoc-append
           assoc-quiz-program-spec-iff assoc-litatom-no-litatom-cars
           no-litatom-cars-update-assoc assoc-update-counter-alist
           assoc-update-counter-alist-iff satisfiesp-with-path-assoc-helper-simple-open
           trans-to trans-resets trans-pred all-cars-litatoms-assoc-trans-list-open
           all-transitions-litatoms my-eval-open all-cars-cadr-quiz-program-spec
           all-cars-litatoms quiz-program-spec-paths car-assoc
           assoc-constant-quiz-program-spec state-name state-trans-list state-pred
           open-satisfiesp-with-path-assoc satisfiesp-base-case
           open-satisfiesp-with-path-assoc2)))

(prove-lemma quiz-program-spec-s-w-p-embedded (rewrite)
  (implies
   (and
    (equal pr (state-pred (assoc (car p) (cadr (quiz-program-spec)))))
    (equal tr (state-trans-list (assoc (car p) (cadr (quiz-program-spec)))))
    (assoc '(c 9) c)
    (no-litatom-cars c))
   (equal
    (satisfiesp-with-path-helper pr tr h c (cadr (quiz-program-spec)) p)
    (quiz-program-spec-embedded-path-helper h c p)))
  ((use (satisfiesp-with-path-assoc-same-helper
         (pred pr) (trans1 tr) (trans2 tr) (counters c)
         (b (cadr (quiz-program-spec))) (path p)))
   (disable-theory t)
   (enable-theory ground-zero)
   (enable quiz-program-spec-s-w-p-a-embedded all-transitions-litatoms quiz-program-spec
           all-cars-litatoms-assoc-trans-list-open all-cars-litatoms state-trans-list)))
```

```
(prove-lemma quiz-program-spec-s-helper-from-embedded (rewrite)
  (implies
   (satisfiesp h (quiz-program-spec))
   (quiz-program-spec-embedded-path-helper h '(((c 9) 0)) (find-path h (quiz-program-spec))))
  ((use (quiz-program-spec-s-w-p-embedded
         (p (find-path h (quiz-program-spec))) (c '(((c 9) 0)))
         (pr (state-pred (assoc (car (find-path h (quiz-program-spec)))
                                (cadr (quiz-program-spec)))))
         (tr (state-trans-list (assoc (car (find-path h (quiz-program-spec)))
                                      (cadr (quiz-program-spec))))))
        (satisfiesp-means-with-find-path (b (quiz-program-spec))))
   (disable-theory t)
   (enable-theory ground-zero)
   (enable satisfiesp quiz-program-spec satisfiesp-with-path satisfiesp-with-path-helper
           state-pred all-cars-litatoms state-name behavior-state-list behavior-initial
           state-trans-list car-find-path no-litatom-cars all-transitions-uniquep
           all-transitions-litatoms counters-of-state-list make-list-alist cars-uniquep
           counters-of-term counters-of-translist counters-of-state counters-of-trans
           trans-resets trans-pred member-cars set-union)))


(prove-lemma member-quiz-spec-counters (rewrite)
  (and
   (equal (member '(c 5) (cadr (assoc y (caddr (assoc x (cadr (quiz-spec)))))))
          (and
           (or (equal y 'one) (equal y 'both))
           (or (equal x 'none) (equal x 'two))))
   (equal (member '(c 6) (cadr (assoc y (caddr (assoc x (cadr (quiz-spec)))))))
          (and
           (or (equal y 'none) (equal y 'two))
           (or (equal x 'one) (equal x 'both))))
   (equal (member '(c 7) (cadr (assoc y (caddr (assoc x (cadr (quiz-spec)))))))
          (and
           (or (equal y 'two) (equal y 'both))
           (or (equal x 'none) (equal x 'one))))
   (equal (member '(c 8) (cadr (assoc y (caddr (assoc x (cadr (quiz-spec)))))))
          (and
           (or (equal y 'none) (equal y 'one))
           (or (equal x 'two) (equal x 'both))))))

(prove-lemma all-cars-cadr-quiz-spec (rewrite)
  (all-cars-litatoms (cadr (quiz-spec))))

(prove-lemma assoc-quiz-spec-iff (rewrite)
  (iff
   (assoc x (cadr (quiz-spec)))
   (or (equal x 'none) (equal x 'one) (equal x 'two) (equal x 'both))))

(prove-lemma assoc-assoc-quiz-spec-iff (rewrite)
  (iff
   (assoc y (caddr (assoc x (cadr (quiz-spec)))))
   (or
    (and (equal x 'none) (or (equal y 'none) (equal y 'one)
                             (equal y 'two) (equal y 'both)))
    (and (equal x 'one) (or (equal y 'none) (equal y 'one)
                             (equal y 'two) (equal y 'both)))
    (and (equal x 'two) (or (equal y 'none) (equal y 'one)
                             (equal y 'two) (equal y 'both)))
    (and (equal x 'both) (or (equal y 'none) (equal y 'one)
                             (equal y 'two) (equal y 'both))))))

(prove-lemma assoc-constant-quiz-spec (rewrite)
```

```
(and
 (equal
  (assoc 'none (cadr (quiz-spec)))
  '(none
    (or (lessp (c 6) 40000) (or (lessp (c 8) 40000)
        (and (equal l1 1) (equal l2 1))))
    ((none () (and (equal a1 'f) (equal a2 'f)))
     (one ((c 5)) (and (equal a1 'depress) (equal a2 'f)))
     (two ((c 7)) (and (equal a1 'f) (equal a2 'depress)))
     (both ((c 5) (c 7)) (and (equal a1 'depress) (equal a2 'depress))))))
 (equal
  (assoc 'one (cadr (quiz-spec)))
   '(one
    (or (lessp (c 5) 40000) (or (lessp (c 8) 40000)
        (and (equal l1 0) (equal l2 1))))
    ((none ((c 6)) (and (equal a1 'release) (equal a2 'f)))
     (one () (and (equal a1 'f) (and (equal a2 'f)
                 (or (not (and (lessp 40000 (c 5)) (equal (old l1) 0)))
                     (equal l1 0)))))
     (two ((c 6) (c 7)) (and (equal a1 'release) (equal a2 'depress)))
     (both ((c 7)) (and (equal a1 'f) (and (equal a2 'depress)
                       (or (not (and (lessp 40000 (c 5)) (equal (old l1) 0)))
                           (equal l1 0))))))))
 (equal
  (assoc 'two (cadr (quiz-spec)))
   '(two
    (or (lessp (c 7) 40000) (or (lessp (c 6) 40000)
        (and (equal l1 1) (equal l2 0))))
    ((none ((c 8)) (and (equal a1 'f) (equal a2 'release)))
     (one ((c 5) (c 8)) (and (equal a1 'depress) (equal a2 'release)))
     (two () (and (equal a1 'f) (and (equal a2 'f)
                 (or (not (and (lessp 40000 (c 7)) (equal (old l2) 0)))
                     (equal l2 0)))))
     (both ((c 5)) (and (equal a1 'depress) (and (equal a2 'f)
                 (or (not (and (lessp 40000 (c 7)) (equal (old l2) 0)))
                     (equal l2 0))))))))
 (equal
  (assoc 'both (cadr (quiz-spec)))
   '(both
    (or (lessp (c 7) 40000) (or (lessp (c 5) 40000)
    (or (and (equal l1 1) (equal l2 0))
        (and (equal l1 0) (equal l2 1)))))
    ((none ((c 6) (c 8)) (and (equal a1 'release) (equal a2 'release)))
     (one ((c 8)) (and (equal a1 'f) (and (equal a2 'release)
                 (or (not (and (lessp 40000 (c 5)) (equal (old l1) 0)))
                     (equal l1 0)))))
     (two ((c 6)) (and (equal a1 'release) (and (equal a2 'f)
                 (or (not (and (lessp 40000 (c 7)) (equal (old l2) 0)))
                     (equal l2 0)))))
     (both () (and (equal a1 'f) (and (equal a2 'f)
             (and (or (not (and (lessp 40000 (c 5)) (equal (old l1) 0)))
                      (equal l1 0))
                  (or (not (and (lessp 40000 (c 7)) (equal (old l2) 0)))
                     (equal l2 0))))))))))))


(defn quiz-spec-embedded-helper (h c path)
  (let
     ((c5 (cadr (assoc '(c 5) c))) (c6 (cadr (assoc '(c 6) c)))
      (c7 (cadr (assoc '(c 7) c))) (c8 (cadr (assoc '(c 8) c)))
      (a1 (cadr (assoc 'a1 (cadr h)))) (a2 (cadr (assoc 'a2 (cadr h))))
      (oldl1 (cadr (assoc 'l1 (car h)))) (oldl2 (cadr (assoc 'l2 (car h))))
```

```
       (l1 (cadr (assoc 'l1 (cadr h)))) (l2 (cadr (assoc 'l2 (cadr h)))))
    (if (listp h)
        (and
         (case (car path)
           (none (or (lessp c6 40000) (lessp c8 40000) (and (equal oldl1 1) (equal oldl2 1))))
           (one (or (lessp c5 40000) (lessp c8 40000) (and (equal oldl1 0) (equal oldl2 1))))
           (two (or (lessp c7 40000) (lessp c6 40000) (and (equal oldl1 1) (equal oldl2 0))))
           (both (or (lessp c7 40000) (lessp c5 40000) (and (equal oldl1 0) (equal oldl2 1))
                     (and (equal oldl1 1) (equal oldl2 0))))
           (otherwise f))
         (if (listp (cdr h))
             (let
                 ((p1 (implies (and (lessp 39999 c5) (equal oldl1 0)) (equal l1 0)))
                  (p2 (implies (and (lessp 39999 c7) (equal oldl2 0)) (equal l2 0))))
               (and
                (case (car path)
                   (none
                    (case (cadr path)
                      (none (and (equal a1 'f) (equal a2 'f)))
                      (one (and (equal a1 'depress) (equal a2 'f)))
                      (two (and (equal a1 'f) (equal a2 'depress)))
                      (both (and (equal a1 'depress) (equal a2 'depress)))
                      (otherwise f)))
                   (one
                    (case (cadr path)
                      (none (and (equal a1 'release) (equal a2 'f)))
                      (one (and (equal a1 'f) (equal a2 'f) p1))
                      (two (and (equal a1 'release) (equal a2 'depress)))
                      (both (and (equal a1 'f) (equal a2 'depress) p1))
                      (otherwise f)))
                   (two
                    (case (cadr path)
                      (none (and (equal a1 'f) (equal a2 'release)))
                      (one (and (equal a1 'depress) (equal a2 'release)))
                      (two (and (equal a1 'f) (equal a2 'f) p2))
                      (both (and (equal a1 'depress) (equal a2 'f) p2))
                      (otherwise f)))
                   (both
                    (case (cadr path)
                      (none (and (equal a1 'release) (equal a2 'release)))
                      (one (and (equal a1 'f) (equal a2 'release) p1))
                      (two (and (equal a1 'release) (equal a2 'f) p2))
                      (both (and (equal a1 'f) (equal a2 'f) p1 p2))
                      (otherwise f)))
                   (otherwise f))
                (quiz-spec-embedded-helper
                 (cdr h)
                 (update-counter-alist
                  c
                  (append
                   (if (equal a1 'depress) '((c 5)) (if (equal a1 'release) '((c 6)) nil))
                   (if (equal a2 'depress) '((c 7)) (if (equal a2 'release) '((c 8)) nil))))
                 (cdr path))))
           t))
         t)))

(prove-lemma quiz-spec-s-w-p-a-embedded (rewrite)
  (implies
   (and
    (equal bs (cadr (quiz-spec)))
    (equal pr (state-pred (assoc (car p) bs)))
    (equal tr (state-trans-list (assoc (car p) bs)))
```

```
    (assoc '(c 5) c)
    (assoc '(c 6) c)
    (assoc '(c 7) c)
    (assoc '(c 8) c)
    (numberp (cadr (assoc '(c 5) c)))
    (numberp (cadr (assoc '(c 6) c)))
    (numberp (cadr (assoc '(c 7) c)))
    (numberp (cadr (assoc '(c 8) c)))
    (no-litatom-cars c))
   (equal
    (satisfiesp-with-path-assoc-helper pr tr h c bs p)
    (quiz-spec-embedded-helper h c p)))
  ((induct (satisfiesp-with-path-assoc-helper pr tr h c bs p))
   (disable-theory t)
   (expand (quiz-spec-embedded-helper h c p))
   (enable-theory ground-zero naturals)
   (enable quiz-spec-embedded-helper satisfiesp-first-trans assoc-append
           member-quiz-spec-counters assoc-quiz-spec-iff assoc-assoc-quiz-spec-iff my-eval
           assoc-litatom-no-litatom-cars no-litatom-cars-update-assoc
           assoc-update-counter-alist assoc-update-counter-alist-iff
           satisfiesp-with-path-assoc-helper-simple-open trans-to trans-resets trans-pred
           all-cars-litatoms-assoc-trans-list-open all-transitions-litatoms my-eval-open
           all-cars-cadr-quiz-spec all-cars-litatoms car-assoc assoc-constant-quiz-spec
           state-name state-trans-list state-pred open-satisfiesp-with-path-assoc
           satisfiesp-base-case open-satisfiesp-with-path-assoc2)))

(prove-lemma quiz-spec-s-w-p-embedded (rewrite)
  (implies
   (and
    (equal pr (state-pred (assoc (car p) (cadr (quiz-spec)))))
    (equal tr (state-trans-list (assoc (car p) (cadr (quiz-spec)))))
    (assoc '(c 5) c)
    (assoc '(c 6) c)
    (assoc '(c 7) c)
    (assoc '(c 8) c)
    (numberp (cadr (assoc '(c 5) c)))
    (numberp (cadr (assoc '(c 6) c)))
    (numberp (cadr (assoc '(c 7) c)))
    (numberp (cadr (assoc '(c 8) c)))
    (no-litatom-cars c))
   (equal
    (satisfiesp-with-path-helper pr tr h c (cadr (quiz-spec)) p)
    (quiz-spec-embedded-helper h c p)))
  ((use (satisfiesp-with-path-assoc-same-helper
        (pred pr) (trans1 tr) (trans2 tr) (counters c)
        (b (cadr (quiz-spec))) (path p)))
   (disable-theory t)
   (enable-theory ground-zero)
   (enable quiz-spec-s-w-p-a-embedded all-transitions-litatoms quiz-spec
           all-cars-litatoms-assoc-trans-list-open all-cars-litatoms state-trans-list)))

(prove-lemma car-quiz-spec (rewrite)
  (equal (car (quiz-spec)) 'none))

(prove-lemma counters-of-state-list-quiz-spec (rewrite)
  (equal
   (counters-of-state-list (cadr (quiz-spec)))
   '((c 8) (c 6) (c 5) (c 7))))

(prove-lemma quiz-spec-s-helper-from-embedded (rewrite)
  (implies
   (and
```

```
    (quiz-spec-embedded-helper  h '(((c 8) 0) ((c 6) 0) ((c 5) 0) ((c 7) 0)) p)
    (or (nlistp h) (equal (car p) 'none)))
   (satisfiesp h (quiz-spec)))
  ((use (quiz-spec-s-w-p-embedded
         (c '(((c 8) 0) ((c 6) 0) ((c 5) 0) ((c 7) 0)))
         (pr (state-pred (assoc (car p) (cadr (quiz-spec)))))
         (tr (state-trans-list (assoc (car p) (cadr (quiz-spec))))))
        (satisfiesp-with-path-helper-means-satisfiesp-helper
         (p (state-pred (assoc (car p) (cadr (quiz-spec)))))
         (tr (state-trans-list (assoc (car p) (cadr (quiz-spec)))))
         (c '(((c 8) 0) ((c 6) 0) ((c 5) 0) ((c 7) 0)))
         (b (cadr (quiz-spec)))
         (path p)))
   (disable-theory t)
   (enable-theory ground-zero)
   (enable satisfiesp satisfiesp-nlistp counters-of-state-list-quiz-spec car-quiz-spec
           car-assoc assoc-quiz-spec-iff state-pred all-cars-litatoms state-name
           behavior-state-list behavior-initial state-trans-list car-find-path
           no-litatom-cars all-transitions-uniquep all-transitions-litatoms
           counters-of-state-list make-list-alist cars-uniquep counters-of-term
           counters-of-translist counters-of-state counters-of-trans trans-resets trans-pred
           member-cars set-union)))


;;; For each of the invariants preserved during the execution of the
;;; system, we prove a theorem that a single step of the computation
;;; does not falsify it.  (These theorems have "-preserved" appended
;;; onto their names.)

(prove-lemma member-quiz-specs-simplify (rewrite)
  (and
   (equal
    (member '(c 1) (cadr (assoc y (caddr (assoc x (cadr (button1-spec)))))))
    (and (equal x 'up) (equal y 'going-down)))
   (equal
    (member '(c 2) (cadr (assoc y (caddr (assoc x (cadr (button1-spec)))))))
    (and (equal x 'down) (equal y 'going-up)))
   (equal
    (member '(c 3) (cadr (assoc y (caddr (assoc x (cadr (button2-spec)))))))
    (and (equal x 'up) (equal y 'going-down)))
   (equal
    (member '(c 4) (cadr (assoc y (caddr (assoc x (cadr (button2-spec)))))))
    (and (equal x 'down) (equal y 'going-up)))
   (equal
    (member '(c 9) (cadr (assoc y (caddr (assoc x (cadr (quiz-program-spec)))))))
    (and (equal x 'update-l2) (equal y 'read-b1))))
  ((enable button1-spec button2-spec quiz-program-spec)))

(prove-lemma button1-reflects-signal1-invariant-preserved (rewrite)
  (implies
   (and
    (listp h) (listp (cdr h))
    (assoc '(c 1) c1) (assoc '(c 2) c1)  (assoc '(c 9) c3)
    (quiz-invariant-button1-reflects-signal1
     (car path1) (car path3)
     (cadr (assoc '(c 1) c1)) (cadr (assoc '(c 2) c1)) (cadr (assoc '(c 9) c3))
     (cadr (assoc 'b1 (car h))))
    (quiz-program-spec-embedded-path-helper h c3 path3)
    (button1-spec-embedded-path-helper h c1 path1)
    (equal new-c1 (update-counter-alist
                   c1 (cadr (assoc (cadr path1)
                                   (caddr (assoc (car path1)
```

```
                                               (cadr (button1-spec))))))))
    (equal new-c3 (update-counter-alist
                   c3 (cadr (assoc (cadr path3)
                                   (caddr (assoc (car path3)
                                                 (cadr (quiz-program-spec)))))))))
   (quiz-invariant-button1-reflects-signal1
    (cadr path1) (cadr path3)
    (cadr (assoc '(c 1) new-c1)) (cadr (assoc '(c 2) new-c1))
    (cadr (assoc '(c 9) new-c3))
    (cadr (assoc 'b1 (cadr h)))))
  ((disable-theory t)
   (enable-theory ground-zero)
       (use
        (quiz-program-spec-embedded-means-first2 (c c3) (p path3))
        (button1-spec-embedded-means-first2 (c c1) (p path1)))
       (enable quiz-invariant-button1-reflects-signal1
               quiz-program-spec-embedded-path-helper-first2
               button1-spec-embedded-path-helper-first2 member-quiz-specs-simplify
               my-eval-open *1*plus assoc-constant-button1-spec assoc-constant-button2-spec
               assoc-constant-quiz-program-spec assoc-append state-name state-pred
               state-trans-list trans-pred trans-resets trans-to assoc-update-counter-alist
               assoc-update-counter-alist-iff firstn length)))

(prove-lemma button2-reflects-signal2-invariant-preserved (rewrite)
  (implies
     (and
      (listp h) (listp (cdr h))
      (assoc '(c 3) c2) (assoc '(c 4) c2)  (assoc '(c 9) c3)
      (quiz-invariant-button2-reflects-signal2
       (car path2) (car path3)
       (cadr (assoc '(c 3) c2)) (cadr (assoc '(c 4) c2)) (cadr (assoc '(c 9) c3))
       (cadr (assoc 'b2 (car h))))
      (quiz-program-spec-embedded-path-helper h c3 path3)
      (button2-spec-embedded-path-helper h c2 path2)
      (equal new-c2 (update-counter-alist
                     c2 (cadr (assoc (cadr path2)
                                     (caddr (assoc (car path2)
                                                   (cadr (button2-spec))))))))
      (equal new-c3 (update-counter-alist
                     c3 (cadr (assoc (cadr path3)
                                     (caddr (assoc (car path3)
                                                   (cadr (quiz-program-spec)))))))))
     (quiz-invariant-button2-reflects-signal2
            (cadr path2) (cadr path3)
            (cadr (assoc '(c 3) new-c2)) (cadr (assoc '(c 4) new-c2))
            (cadr (assoc '(c 9) new-c3))
            (cadr (assoc 'b2 (cadr h)))))
  ((disable-theory t)
   (enable-theory ground-zero)
       (use
        (quiz-program-spec-embedded-means-first2 (c c3) (p path3))
        (button2-spec-embedded-means-first2 (c c2) (p path2)))
       (enable quiz-invariant-button2-reflects-signal2
               quiz-program-spec-embedded-path-helper-first2
               button2-spec-embedded-path-helper-first2 member-quiz-specs-simplify
               my-eval-open *1*plus assoc-constant-button1-spec assoc-constant-button2-spec
               assoc-constant-quiz-program-spec assoc-append state-name state-pred
               state-trans-list trans-pred trans-resets trans-to assoc-update-counter-alist
               assoc-update-counter-alist-iff firstn length)))

(prove-lemma sub1-plus-add1 (rewrite)
  (and
```

```
   (equal (sub1 (plus (add1 x) y)) (plus x y))
   (equal (sub1 (plus x (add1 y))) (plus x y))))

(prove-lemma plus-nnumberp (rewrite)
  (implies
   (not (numberp x))
   (and (equal (plus x y) (fix y)) (equal (plus y x) (fix y)))))

(prove-lemma s-reflects-signal1-invariant-preserved (rewrite)
    (implies
     (and
      (listp h) (listp (cdr h))
      (assoc '(c 1) c1) (assoc '(c 2) c1)  (assoc '(c 9) c3)
      (quiz-invariant-button1-reflects-signal1
            (car path1) (car path3)
            (cadr (assoc '(c 1) c1)) (cadr (assoc '(c 2) c1))
            (cadr (assoc '(c 9) c3))
            (cadr (assoc 'b1 (car h))))
      (quiz-invariant-s-reflects-signal1
        (car path1) (car path3) (cadr (assoc '(c 1) c1))
        (cadr (assoc '(c 2) c1)) (cadr (assoc '(c 9) c3))
        (cadr (assoc 's (car h))))
      (button1-spec-embedded-path-helper h c1 path1)
      (quiz-program-spec-embedded-path-helper h c3 path3)
      (equal new-c1 (update-counter-alist
                     c1 (cadr (assoc (cadr path1)
                                     (caddr (assoc (car path1)
                                                   (cadr (button1-spec))))))))
      (equal new-c3 (update-counter-alist
                     c3 (cadr (assoc (cadr path3)
                                     (caddr (assoc (car path3)
                                                   (cadr (quiz-program-spec)))))))))
     (quiz-invariant-s-reflects-signal1
            (cadr path1) (cadr path3)
            (cadr (assoc '(c 1) new-c1)) (cadr (assoc '(c 2) new-c1))
            (cadr (assoc '(c 9) new-c3))
            (cadr (assoc 's (cadr h)))))
  ((disable-theory t)
   (enable-theory ground-zero)
   (use (quiz-program-spec-embedded-means-first2 (c c3) (p path3))
        (button1-spec-embedded-means-first2 (c c1) (p path1)))
   (enable quiz-invariant-button1-reflects-signal1 plus-nnumberp commutativity-of-plus
           commutativity2-of-plus quiz-invariant-s-reflects-signal1
           quiz-program-spec-embedded-path-helper-first2
           button1-spec-embedded-path-helper-first2 sub1-plus-add1 member-quiz-specs-simplify
           my-eval-open *1*plus assoc-constant-button1-spec assoc-constant-button2-spec
           assoc-constant-quiz-program-spec assoc-append state-name state-pred
           state-trans-list trans-pred trans-resets trans-to assoc-update-counter-alist
           assoc-update-counter-alist-iff firstn length)))

(prove-lemma s-reflects-signal2-invariant-preserved (rewrite)
  (implies
   (and
    (listp h) (listp (cdr h))
    (assoc '(c 3) c2) (assoc '(c 4) c2)  (assoc '(c 9) c3)
    (quiz-invariant-button2-reflects-signal2
     (car path2) (car path3)
     (cadr (assoc '(c 3) c2)) (cadr (assoc '(c 4) c2))
     (cadr (assoc '(c 9) c3))
     (cadr (assoc 'b2 (car h))))
    (quiz-invariant-s-reflects-signal2
     (car path2) (car path3) (cadr (assoc '(c 3) c2))
```

```
     (cadr (assoc '(c 4) c2)) (cadr (assoc '(c 9) c3))
     (cadr (assoc 's (car h))))
    (button2-spec-embedded-path-helper h c2 path2)
    (quiz-program-spec-embedded-path-helper h c3 path3)
    (equal new-c2 (update-counter-alist
                   c2 (cadr (assoc (cadr path2)
                                   (caddr (assoc (car path2)
                                                 (cadr (button2-spec))))))))
    (equal new-c3 (update-counter-alist
                   c3 (cadr (assoc (cadr path3)
                                   (caddr (assoc (car path3)
                                                 (cadr (quiz-program-spec)))))))))

   (quiz-invariant-s-reflects-signal2
    (cadr path2) (cadr path3)
    (cadr (assoc '(c 3) new-c2)) (cadr (assoc '(c 4) new-c2))
    (cadr (assoc '(c 9) new-c3))
    (cadr (assoc 's (cadr h)))))
  ((disable-theory t)
   (enable-theory ground-zero)
   (use (quiz-program-spec-embedded-means-first2 (c c3) (p path3))
        (button2-spec-embedded-means-first2 (c c2) (p path2)))
   (enable quiz-invariant-button2-reflects-signal2 quiz-invariant-s-reflects-signal2
           quiz-program-spec-embedded-path-helper-first2
           button2-spec-embedded-path-helper-first2 sub1-plus-add1 member-quiz-specs-simplify
           my-eval-open *1*plus assoc-constant-button1-spec assoc-constant-button2-spec
           assoc-constant-quiz-program-spec assoc-append state-name state-pred
           state-trans-list trans-pred trans-resets trans-to assoc-update-counter-alist
           assoc-update-counter-alist-iff firstn length)))

(prove-lemma l1-reflects-signals-invariant-preserved (rewrite)
    (implies
     (and
      (listp h) (listp (cdr h))
      (assoc '(c 1) c1) (assoc '(c 2) c1)
      (assoc '(c 3) c2) (assoc '(c 4) c2)  (assoc '(c 9) c3)
      (numberp (cadr (assoc '(c 1) c1))) (numberp (cadr (assoc '(c 2) c1)))
      (numberp (cadr (assoc '(c 3) c2))) (numberp (cadr (assoc '(c 4) c2)))
      (numberp (cadr (assoc '(c 9) c3)))
      (quiz-invariant-button1-reflects-signal1
            (car path1) (car path3)
            (cadr (assoc '(c 1) c1)) (cadr (assoc '(c 2) c1))
            (cadr (assoc '(c 9) c3))
            (cadr (assoc 'b1 (car h))))
      (quiz-invariant-button2-reflects-signal2
            (car path2) (car path3)
            (cadr (assoc '(c 3) c2)) (cadr (assoc '(c 4) c2))
            (cadr (assoc '(c 9) c3))
            (cadr (assoc 'b2 (car h))))
      (quiz-invariant-s-reflects-signal1
        (car path1) (car path3) (cadr (assoc '(c 1) c1))
        (cadr (assoc '(c 2) c1)) (cadr (assoc '(c 9) c3))
        (cadr (assoc 's (car h))))
      (quiz-invariant-s-reflects-signal1
        (cadr path1) (cadr path3) (cadr (assoc '(c 1) new-c1))
        (cadr (assoc '(c 2) new-c1)) (cadr (assoc '(c 9) new-c3))
        (cadr (assoc 's (cadr h))))
      (quiz-invariant-s-reflects-signal2
        (car path2) (car path3) (cadr (assoc '(c 3) c2))
        (cadr (assoc '(c 4) c2)) (cadr (assoc '(c 9) c3))
        (cadr (assoc 's (car h))))
      (quiz-invariant-s-reflects-signal2
```

```
        (cadr path2) (cadr path3) (cadr (assoc '(c 3) new-c2))
        (cadr (assoc '(c 4) new-c2)) (cadr (assoc '(c 9) new-c3))
        (cadr (assoc 's (cadr h))))
      (quiz-invariant-l1-reflects-signals
       (car path1) (car path2) (car path3)
       (cadr (assoc '(c 1) c1)) (cadr (assoc '(c 2) c1))
       (cadr (assoc '(c 3) c2)) (cadr (assoc '(c 4) c2))
       (cadr (assoc '(c 9) c3)) (cadr (assoc 's (car h)))
       (cadr (assoc 'l1 (car h))))
      (button1-spec-embedded-path-helper h c1 path1)
      (button2-spec-embedded-path-helper h c2 path2)
      (quiz-program-spec-embedded-path-helper h c3 path3)
      (equal new-c1 (update-counter-alist
                     c1 (cadr (assoc (cadr path1)
                                     (caddr (assoc (car path1)
                                                   (cadr (button1-spec))))))))
      (equal new-c2 (update-counter-alist
                      c2 (cadr (assoc (cadr path2)
                                      (caddr (assoc (car path2)
                                                    (cadr (button2-spec))))))))
      (equal new-c3 (update-counter-alist
                     c3 (cadr (assoc (cadr path3)
                                     (caddr (assoc (car path3)
                                                   (cadr (quiz-program-spec)))))))))
     (quiz-invariant-l1-reflects-signals
      (cadr path1) (cadr path2) (cadr path3)
      (cadr (assoc '(c 1) new-c1)) (cadr (assoc '(c 2) new-c1))
      (cadr (assoc '(c 3) new-c2)) (cadr (assoc '(c 4) new-c2))
      (cadr (assoc '(c 9) new-c3)) (cadr (assoc 's (cadr h)))
      (cadr (assoc 'l1 (cadr h)))))
  ((disable-theory t)
   (enable-theory ground-zero)
   (use (quiz-program-spec-embedded-means-first2 (c c3) (p path3))
        (button1-spec-embedded-means-first2 (c c1) (p path1))
        (button2-spec-embedded-means-first2 (c c2) (p path2)))
   (enable quiz-invariant-l1-reflects-signals quiz-invariant-s-reflects-signal1
           quiz-invariant-s-reflects-signal2 quiz-invariant-button1-reflects-signal1
           quiz-invariant-button2-reflects-signal2
           quiz-program-spec-embedded-path-helper-first2
           button1-spec-embedded-path-helper-first2 button2-spec-embedded-path-helper-first2
           sub1-plus-add1 member-quiz-specs-simplify my-eval-open *1*plus equal-plus-0
           commutativity-of-plus assoc-constant-button1-spec assoc-constant-button2-spec
           assoc-constant-quiz-program-spec assoc-append state-name state-pred
           state-trans-list trans-pred trans-resets trans-to assoc-update-counter-alist
           assoc-update-counter-alist-iff firstn length)))

(prove-lemma l2-reflects-signals-invariant-preserved (rewrite)
    (implies
     (and
      (listp h) (listp (cdr h))
      (assoc '(c 1) c1) (assoc '(c 2) c1)
      (assoc '(c 3) c2) (assoc '(c 4) c2)  (assoc '(c 9) c3)
      (numberp (cadr (assoc '(c 1) c1))) (numberp (cadr (assoc '(c 2) c1)))
      (numberp (cadr (assoc '(c 3) c2))) (numberp (cadr (assoc '(c 4) c2)))
      (numberp (cadr (assoc '(c 9) c3)))
      (quiz-invariant-button1-reflects-signal1
       (car path1) (car path3)
       (cadr (assoc '(c 1) c1)) (cadr (assoc '(c 2) c1))
       (cadr (assoc '(c 9) c3))
       (cadr (assoc 'b1 (car h))))
      (quiz-invariant-button2-reflects-signal2
       (car path2) (car path3)
```

```
     (cadr (assoc '(c 3) c2)) (cadr (assoc '(c 4) c2))
     (cadr (assoc '(c 9) c3))
     (cadr (assoc 'b2 (car h))))
    (quiz-invariant-s-reflects-signal1
     (car path1) (car path3) (cadr (assoc '(c 1) c1))
     (cadr (assoc '(c 2) c1)) (cadr (assoc '(c 9) c3))
     (cadr (assoc 's (car h))))
    (quiz-invariant-s-reflects-signal1
     (cadr path1) (cadr path3) (cadr (assoc '(c 1) new-c1))
     (cadr (assoc '(c 2) new-c1)) (cadr (assoc '(c 9) new-c3))
     (cadr (assoc 's (cadr h))))
    (quiz-invariant-s-reflects-signal2
     (car path2) (car path3) (cadr (assoc '(c 3) c2))
     (cadr (assoc '(c 4) c2)) (cadr (assoc '(c 9) c3))
     (cadr (assoc 's (car h))))
    (quiz-invariant-s-reflects-signal2
     (cadr path2) (cadr path3) (cadr (assoc '(c 3) new-c2))
     (cadr (assoc '(c 4) new-c2)) (cadr (assoc '(c 9) new-c3))
     (cadr (assoc 's (cadr h))))
    (quiz-invariant-l2-reflects-signals
     (car path1) (car path2) (car path3)
     (cadr (assoc '(c 1) c1)) (cadr (assoc '(c 2) c1))
     (cadr (assoc '(c 3) c2)) (cadr (assoc '(c 4) c2))
     (cadr (assoc '(c 9) c3)) (cadr (assoc 's (car h)))
     (cadr (assoc 'l2 (car h))))
    (button1-spec-embedded-path-helper h c1 path1)
    (button2-spec-embedded-path-helper h c2 path2)
    (quiz-program-spec-embedded-path-helper h c3 path3)
    (equal new-c1 (update-counter-alist
                   c1 (cadr (assoc (cadr path1)
                                   (caddr (assoc (car path1)
                                                 (cadr (button1-spec))))))))
    (equal new-c2 (update-counter-alist
                   c2 (cadr (assoc (cadr path2)
                                   (caddr (assoc (car path2)
                                                 (cadr (button2-spec))))))))
    (equal new-c3 (update-counter-alist
                   c3 (cadr (assoc (cadr path3)
                                   (caddr (assoc (car path3)
                                                 (cadr (quiz-program-spec)))))))))
   (quiz-invariant-l2-reflects-signals
    (cadr path1) (cadr path2) (cadr path3)
    (cadr (assoc '(c 1) new-c1)) (cadr (assoc '(c 2) new-c1))
    (cadr (assoc '(c 3) new-c2)) (cadr (assoc '(c 4) new-c2))
    (cadr (assoc '(c 9) new-c3)) (cadr (assoc 's (cadr h)))
    (cadr (assoc 'l2 (cadr h)))))
((disable-theory t)
 (enable-theory ground-zero)
 (use (quiz-program-spec-embedded-means-first2 (c c3) (p path3))
      (button1-spec-embedded-means-first2 (c c1) (p path1))
      (button2-spec-embedded-means-first2 (c c2) (p path2)))
(enable quiz-invariant-l2-reflects-signals quiz-invariant-s-reflects-signal1
        quiz-invariant-s-reflects-signal2 quiz-invariant-button1-reflects-signal1
        quiz-invariant-button2-reflects-signal2
        quiz-program-spec-embedded-path-helper-first2
        button1-spec-embedded-path-helper-first2 button2-spec-embedded-path-helper-first2
        sub1-plus-add1 member-quiz-specs-simplify my-eval-open *1*plus equal-plus-0
        commutativity-of-plus assoc-constant-button1-spec assoc-constant-button2-spec
        assoc-constant-quiz-program-spec assoc-append state-name state-pred
        state-trans-list trans-pred trans-resets trans-to assoc-update-counter-alist
        assoc-update-counter-alist-iff firstn length)))
```

```
(prove-lemma quiz-invariant-l1-means-s-preserved (rewrite)
    (implies
     (and
      (listp h) (listp (cdr h))
      (assoc '(c 1) c1) (assoc '(c 2) c1)
      (assoc '(c 9) c3)
      (numberp (cadr (assoc '(c 1) c1))) (numberp (cadr (assoc '(c 2) c1)))
      (numberp (cadr (assoc '(c 9) c3)))
      (quiz-invariant-button1-reflects-signal1
       (car path1) (car path3)
       (cadr (assoc '(c 1) c1)) (cadr (assoc '(c 2) c1))
       (cadr (assoc '(c 9) c3))
       (cadr (assoc 'b1 (car h))))
      (quiz-invariant-s-reflects-signal1
       (car path1) (car path3) (cadr (assoc '(c 1) c1))
       (cadr (assoc '(c 2) c1)) (cadr (assoc '(c 9) c3))
       (cadr (assoc 's (car h))))
      (quiz-invariant-s-reflects-signal1
       (cadr path1) (cadr path3) (cadr (assoc '(c 1) new-c1))
       (cadr (assoc '(c 2) new-c1)) (cadr (assoc '(c 9) new-c3))
       (cadr (assoc 's (cadr h))))
      (quiz-invariant-l1-means-s
       (car path1) (car path3)
       (cadr (assoc '(c 1) c1)) (cadr (assoc '(c 9) c3))
       (cadr (assoc 's (car h))) (cadr (assoc 'l1 (car h))))
      (button1-spec-embedded-path-helper h c1 path1)
      (quiz-program-spec-embedded-path-helper h c3 path3)
      (equal new-c1 (update-counter-alist
                     c1 (cadr (assoc (cadr path1)
                                     (caddr (assoc (car path1)
                                                   (cadr (button1-spec))))))))
      (equal new-c3 (update-counter-alist
                     c3 (cadr (assoc (cadr path3)
                                     (caddr (assoc (car path3)
                                                   (cadr (quiz-program-spec)))))))))
     (quiz-invariant-l1-means-s
      (cadr path1) (cadr path3)
      (cadr (assoc '(c 1) new-c1)) (cadr (assoc '(c 9) new-c3))
      (cadr (assoc 's (cadr h))) (cadr (assoc 'l1 (cadr h)))))
  ((disable-theory t)
   (enable-theory ground-zero)
   (use (quiz-program-spec-embedded-means-first2 (c c3) (p path3))
        (button1-spec-embedded-means-first2 (c c1) (p path1)))
   (enable quiz-invariant-l1-means-s quiz-invariant-l2-reflects-signals
           quiz-invariant-s-reflects-signal1 quiz-invariant-s-reflects-signal2
           quiz-invariant-button1-reflects-signal1 quiz-invariant-button2-reflects-signal2
           quiz-program-spec-embedded-path-helper-first2
           button1-spec-embedded-path-helper-first2 button2-spec-embedded-path-helper-first2
           sub1-plus-add1 member-quiz-specs-simplify my-eval-open *1*plus equal-plus-0
           commutativity-of-plus assoc-constant-button1-spec assoc-constant-button2-spec
           assoc-constant-quiz-program-spec assoc-append state-name state-pred
           state-trans-list trans-pred trans-resets trans-to assoc-update-counter-alist
           assoc-update-counter-alist-iff firstn length)))


(prove-lemma quiz-invariant-l2-means-s-preserved (rewrite)
    (implies
     (and
      (listp h) (listp (cdr h))
      (assoc '(c 3) c2) (assoc '(c 4) c2)  (assoc '(c 9) c3)
      (numberp (cadr (assoc '(c 3) c2))) (numberp (cadr (assoc '(c 4) c2)))
      (numberp (cadr (assoc '(c 9) c3)))
```

```
      (quiz-invariant-button2-reflects-signal2
       (car path2) (car path3)
       (cadr (assoc '(c 3) c2)) (cadr (assoc '(c 4) c2))
       (cadr (assoc '(c 9) c3))
       (cadr (assoc 'b2 (car h))))
      (quiz-invariant-s-reflects-signal2
       (car path2) (car path3) (cadr (assoc '(c 3) c2))
       (cadr (assoc '(c 4) c2)) (cadr (assoc '(c 9) c3))
       (cadr (assoc 's (car h))))
      (quiz-invariant-s-reflects-signal2
       (cadr path2) (cadr path3) (cadr (assoc '(c 3) new-c2))
       (cadr (assoc '(c 4) new-c2)) (cadr (assoc '(c 9) new-c3))
       (cadr (assoc 's (cadr h))))
      (quiz-invariant-l2-means-s
       (car path2) (car path3)
       (cadr (assoc '(c 3) c2)) (cadr (assoc '(c 9) c3))
       (cadr (assoc 's (car h)))
       (cadr (assoc 'l2 (car h))))
      (button2-spec-embedded-path-helper h c2 path2)
      (quiz-program-spec-embedded-path-helper h c3 path3)
      (equal new-c2 (update-counter-alist
                     c2 (cadr (assoc (cadr path2)
                                     (caddr (assoc (car path2)
                                                   (cadr (button2-spec))))))))
      (equal new-c3 (update-counter-alist
                     c3 (cadr (assoc (cadr path3)
                                     (caddr (assoc (car path3)
                                                   (cadr (quiz-program-spec)))))))))
     (quiz-invariant-l2-means-s
       (cadr path2) (cadr path3)
       (cadr (assoc '(c 3) new-c2)) (cadr (assoc '(c 9) new-c3))
       (cadr (assoc 's (cadr h)))
       (cadr (assoc 'l2 (cadr h)))))
  ((disable-theory t)
   (enable-theory ground-zero)
   (use (quiz-program-spec-embedded-means-first2 (c c3) (p path3))
        (button2-spec-embedded-means-first2 (c c2) (p path2)))
  (enable quiz-invariant-l2-means-s quiz-invariant-s-reflects-signal1
          quiz-invariant-s-reflects-signal2 quiz-invariant-button1-reflects-signal1
          quiz-invariant-button2-reflects-signal2
          quiz-program-spec-embedded-path-helper-first2
          button1-spec-embedded-path-helper-first2 button2-spec-embedded-path-helper-first2
          sub1-plus-add1 member-quiz-specs-simplify my-eval-open *1*plus equal-plus-0
          commutativity-of-plus assoc-constant-button1-spec assoc-constant-button2-spec
          assoc-constant-quiz-program-spec assoc-append state-name state-pred
          state-trans-list trans-pred trans-resets trans-to assoc-update-counter-alist
          assoc-update-counter-alist-iff firstn length)))

;; The functions we have used to represent the invariants we are
;; interested were necessarily more complex than what we actually
;; need.  We next introduce some simpler properties that result from
;; the complex ones and contain the properties we actualy need.
;; Basically, they are the same properties except not involving the
;; program behavior.  The lemma names end in -means

;; first we prove an invariant about our program spec

(defn quiz-invariant-simple-program-props (pp c9)
  (and
   (or (equal pp 'read-b1) (equal pp 'read-b2) (equal pp 'update-state)
       (equal pp 'update-l1) (equal pp 'update-l2))
   (lessp c9 400)))
```

```
(prove-lemma quiz-invariant-simple-program-props-preserved (rewrite)
  (implies
     (and
      (listp h) (listp (cdr h))
      (quiz-program-spec-embedded-path-helper h c3 path3)
      (quiz-invariant-simple-program-props (car path3) (cadr (assoc '(c 9) c3)))
      (assoc '(c 9) c3)
      (equal new-c3 (update-counter-alist
              c3 (if (and (equal (car path3) 'update-l2)
                                         (equal (cadr path3) 'read-b1))
                                     '((c 9))
                                     nil))))
     (quiz-invariant-simple-program-props (cadr path3) (cadr (assoc '(c 9) new-c3))))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable quiz-invariant-simple-program-props quiz-invariant-s-reflects-signal1
           quiz-invariant-s-reflects-signal2 quiz-invariant-button1-reflects-signal1
           quiz-invariant-button2-reflects-signal2 quiz-program-spec-embedded-path-helper
           button1-spec-embedded-path-helper-first2 button2-spec-embedded-path-helper-first2
           sub1-plus-add1 member-quiz-specs-simplify my-eval-open *1*plus equal-plus-0
           commutativity-of-plus assoc-constant-button1-spec assoc-constant-button2-spec
           assoc-constant-quiz-program-spec assoc-append state-name state-pred
           state-trans-list trans-pred trans-resets trans-to assoc-update-counter-alist
           assoc-update-counter-alist-iff firstn length)))


(prove-lemma s-reflects-signal1-means (rewrite)
  (implies
   (and
    (quiz-invariant-simple-program-props pp c9)
    (or
     (and (equal p1 'down) (lessp 30800 c1) (not (or (equal s 1) (equal s 2))))
     (and (equal p1 'up) (lessp 30800 c2) (not (or (equal s 0) (equal s 2))))))
   (not (quiz-invariant-s-reflects-signal1 p1 pp c1 c2 c9 s)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable quiz-invariant-simple-program-props quiz-invariant-s-reflects-signal1)))

(prove-lemma s-reflects-signal2-means (rewrite)
  (implies
   (and
    (quiz-invariant-simple-program-props pp c9)
    (or
     (and (equal p2 'down) (lessp 30800 c3) (not (or (equal s 1) (equal s 2))))
     (and (equal p2 'up) (lessp 30800 c4) (not (or (equal s 0) (equal s 1))))))
   (not (quiz-invariant-s-reflects-signal2 p2 pp c3 c4 c9 s)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable quiz-invariant-simple-program-props quiz-invariant-s-reflects-signal2)))


(prove-lemma l1-reflects-signals-means (rewrite)
  (implies
   (and
    (quiz-invariant-simple-program-props pp c9)
    (or
     (and (equal p1 'up) (lessp 30800 c2) (not (equal l1 1)))
     (and (equal p1 'down) (equal p2 'up)
          (lessp 30800 c1) (lessp 30800 c4)
          (not (equal l1 0)))
```

```
     (and (equal p1 'down) (equal p2 'down)
          (lessp 30800 c1) (lessp 30800 c3)
          (not (equal l1 (if (equal s 1) 0 1))))))
   (not (quiz-invariant-l1-reflects-signals p1 p2 pp c1 c2 c3 c4 c9 s l1)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable quiz-invariant-simple-program-props quiz-invariant-l1-reflects-signals)))


(prove-lemma l2-reflects-signals-means (rewrite)
  (implies
   (and
   (quiz-invariant-simple-program-props pp c9)
   (or
    (and (equal p2 'up) (lessp 30800 c4) (not (equal l2 1)))
    (and (equal p2 'down) (equal p1 'up)
         (lessp 30800 c3) (lessp 30800 c2)
         (not (equal l2 0)))
    (and (equal p2 'down) (equal p1 'down)
         (lessp 30800 c3) (lessp 30800 c1)
         (not (equal l2 (if (equal s 2) 0 1))))))
   (not (quiz-invariant-l2-reflects-signals p1 p2 pp c1 c2 c3 c4 c9 s l2)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable quiz-invariant-simple-program-props quiz-invariant-l2-reflects-signals)))

(prove-lemma l1-means-s-means (rewrite)
  (implies
   (and
    (quiz-invariant-simple-program-props pp c9)
    (equal l1 0) (equal p1 'down) (lessp 30800 c1)
    (not (equal s 1)))
   (not (quiz-invariant-l1-means-s p1 pp c1 c9 s l1)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable quiz-invariant-simple-program-props quiz-invariant-l1-means-s)))


(prove-lemma l2-means-s-means (rewrite)
  (implies
   (and
    (quiz-invariant-simple-program-props pp c9)
    (equal l2 0) (equal p2 'down) (lessp 30800 c3)
    (not (equal s 2)))
   (not (quiz-invariant-l2-means-s p2 pp c3 c9 s l2)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable quiz-invariant-simple-program-props quiz-invariant-l2-means-s)))



(prove-lemma lights-steady-means (rewrite)
  (implies
   (and
    (listp h) (listp (cdr h))
    (quiz-program-spec-embedded-path-helper h c3 path3))
   (and
    (implies
     (and (equal (cadr (assoc 'l1 (car h))) 0)
          (not (equal (cadr (assoc 'l1 (cadr h))) 0)))
     (not (equal (cadr (assoc 's (car h))) 1)))
    (implies
```

```
     (and (equal (cadr (assoc 'l2 (car h))) 0)
          (not (equal (cadr (assoc 'l2 (cadr h))) 0)))
     (not (equal (cadr (assoc 's (car h))) 2)))))
    ((disable-theory t)
     (enable-theory ground-zero)
     (use (quiz-program-spec-embedded-means-first2 (c c3) (p path3)))
     (enable quiz-program-spec-embedded-path-helper-first2 sub1-plus-add1
             member-quiz-specs-simplify my-eval-open *1*plus equal-plus-0
             commutativity-of-plus assoc-constant-button1-spec assoc-constant-button2-spec
             assoc-constant-quiz-program-spec assoc-append state-name state-pred
             state-trans-list trans-pred trans-resets trans-to assoc-update-counter-alist
             assoc-update-counter-alist-iff firstn length)))



(prove-lemma lessp-transitivity1 (rewrite)
  (implies
   (and
    (not (lessp b c))
    (lessp a c))
   (equal (lessp a b) t)))



;;; Finally, the big theorem about abstract executions of the system

(defn abstract-quiz-map (p1 p2)
  (if (and
       (listp p1)
       (listp p2))
      (cons
       (if (or (equal (car p1) 'up) (equal (car p1) 'going-up))
           (if (or (equal (car p2) 'up) (equal (car p2) 'going-up))
               'none
             'two)
         (if (or (equal (car p2) 'up) (equal (car p2) 'going-up))
             'one
           'both))
       (abstract-quiz-map (cdr p1) (cdr p2)))
    nil))

(prove-lemma car-abstract-quiz-map
             (rewrite)
             (equal (car (abstract-quiz-map p1 p2))
                    (if (and (listp p1) (listp p2))
                        (if (or (equal (car p1) 'up)
                                (equal (car p1) 'going-up))
                            (if (or (equal (car p2) 'up)
                                    (equal (car p2) 'going-up))
                                'none
                                'two)
                            (if (or (equal (car p2) 'up)
                                    (equal (car p2) 'going-up))
                                'one
                                'both))
                        0)))



;; We want to disable some of the functions in our correctness theorem,
;; so we set up an induction by hand.
```

```
(defn abstract-quiz-system-embedded-induct (c1 c2 c3 c path1 path2 pathp h)
  (if (and (listp h) (listp (cdr h)))
      (let ((a1 (cadr (assoc 'a1 (cadr h)))) (a2 (cadr (assoc 'a2 (cadr h)))))
      (abstract-quiz-system-embedded-induct
       (update-counter-alist
        c1 (if (equal a1 'depress) '((c 1)) (if (equal a1 'release) '((c 2)) nil)))
       (update-counter-alist
        c2 (if (equal a2 'depress) '((c 3)) (if (equal a2 'release) '((c 4)) nil)))
       (update-counter-alist
        c3 (if (and (equal (car pathp) 'update-l2) (equal (cadr pathp) 'read-b1))
               '((c 9)) nil))
       (update-counter-alist
        c
        (append
         (if (equal a1 'depress) '((c 5)) (if (equal a1 'release) '((c 6)) nil))
         (if (equal a2 'depress) '((c 7)) (if (equal a2 'release) '((c 8)) nil))))
       (cdr path1)
       (cdr path2)
       (cdr pathp)
       (cdr h)))
    t))

(prove-lemma button1-spec-embedded-path-helper-recurse (rewrite)
  (implies
   (and
    (button1-spec-embedded-path-helper h c1 path1)
    (equal (cdr path1) new-path1)
    (equal
     new-c1
     (update-counter-alist
      c1
      (if (equal (cadr (assoc 'a1 (cadr h))) 'depress)
          '((c 1))
        (if (equal (cadr (assoc 'a1 (cadr h))) 'release)
            '((c 2)) nil)))))
   (button1-spec-embedded-path-helper (cdr h) new-c1 new-path1))
  ((enable button1-spec-embedded-path-helper)
   (disable-theory t)
   (enable-theory ground-zero)))

(prove-lemma button2-spec-embedded-path-helper-recurse (rewrite)
  (implies
   (and
    (button2-spec-embedded-path-helper h c2 path2)
    (equal (cdr path2) new-path2)
    (equal
     new-c2
     (update-counter-alist
      c2
      (if (equal (cadr (assoc 'a2 (cadr h))) 'depress)
          '((c 3))
        (if (equal (cadr (assoc 'a2 (cadr h))) 'release)
            '((c 4)) nil)))))
   (button2-spec-embedded-path-helper (cdr h) new-c2 new-path2))
  ((enable button2-spec-embedded-path-helper)
   (disable-theory t)
   (enable-theory ground-zero)))

(prove-lemma quiz-program-spec-embedded-path-helper-recurse (rewrite)
  (implies
   (and
    (quiz-program-spec-embedded-path-helper h c3 path3)
```

```
    (equal (cdr path3) new-path3)
    (equal
     new-c3
     (update-counter-alist
      c3
      (if (and (equal (car path3) 'update-l2) (equal (cadr path3) 'read-b1))
          '((c 9)) nil))))
   (quiz-program-spec-embedded-path-helper (cdr h) new-c3 new-path3))
  ((enable quiz-program-spec-embedded-path-helper)
   (disable-theory t)
   (enable-theory ground-zero)))

(prove-lemma nlistp-path-specs (rewrite)
  (implies
   (not (listp path))
   (and
    (equal (button1-spec-embedded-path-helper h c path) (nlistp h))
    (equal (button2-spec-embedded-path-helper h c path) (nlistp h))
    (equal (quiz-program-spec-embedded-path-helper h c path) (nlistp h))
    (equal (quiz-spec-embedded-helper h c path) (nlistp h)))))

(prove-lemma length-button1-spec-path (rewrite)
  (implies
   (lessp (length path) (length h))
   (not (button1-spec-embedded-path-helper h c path))))

(prove-lemma length-button2-spec-path (rewrite)
  (implies
   (lessp (length path) (length h))
   (not (button2-spec-embedded-path-helper h c path))))

(prove-lemma length-quiz-program-spec-path (rewrite)
  (implies
   (lessp (length path) (length h))
   (not (quiz-program-spec-embedded-path-helper h c path)))
  ((induct (quiz-program-spec-embedded-path-helper h c path))
   (disable-theory t)
   (enable-theory ground-zero)
   (enable quiz-program-spec-embedded-path-helper length)))

(prove-lemma length-quiz-spec-path (rewrite)
  (implies
   (lessp (length path) (length h))
   (not (quiz-spec-embedded-helper h c path)))
  ((disable-theory t)
   (induct (quiz-spec-embedded-helper h c path))
   (enable-theory ground-zero)
   (enable quiz-spec-embedded-helper length)))

(defn reasonable-sp (s)
  (or (equal s 0) (equal s 1) (equal s 2)))

(prove-lemma reasonable-sp-preserved (rewrite)
  (implies
   (and
    (quiz-program-spec-embedded-path-helper h c3 path3)
    (or (equal (cadr (assoc 's (car h))) 0)
        (equal (cadr (assoc 's (car h))) 1)
        (equal (cadr (assoc 's (car h))) 2)))
   (reasonable-sp (cadr (assoc 's (cadr h)))))
  ((induct (quiz-program-spec-embedded-path-helper h c3 path3))))
```

```
(prove-lemma quiz-spec-embedded-helper-cons (rewrite)
  (and
   (equal
    (quiz-spec-embedded-helper h c (cons 'one path))
    (let
        ((c5 (cadr (assoc '(c 5) c))) (c6 (cadr (assoc '(c 6) c)))
         (c7 (cadr (assoc '(c 7) c))) (c8 (cadr (assoc '(c 8) c)))
         (a1 (cadr (assoc 'a1 (cadr h)))) (a2 (cadr (assoc 'a2 (cadr h))))
         (oldl1 (cadr (assoc 'l1 (car h)))) (oldl2 (cadr (assoc 'l2 (car h))))
         (l1 (cadr (assoc 'l1 (cadr h)))) (l2 (cadr (assoc 'l2 (cadr h)))))
      (if (listp h)
          (and
           (or (lessp c5 40000) (lessp c8 40000) (and (equal oldl1 0) (equal oldl2 1)))
           (if (listp (cdr h))
               (let
                   ((p1 (implies (and (lessp 39999 c5) (equal oldl1 0)) (equal l1 0)))
                    (p2 (implies (and (lessp 39999 c7) (equal oldl2 0)) (equal l2 0))))
                 (and
                  (case (car path)
                        (none (and (equal a1 'release) (equal a2 'f)))
                        (one (and (equal a1 'f) (equal a2 'f) p1))
                        (two (and (equal a1 'release) (equal a2 'depress)))
                        (both (and (equal a1 'f) (equal a2 'depress) p1))
                        (otherwise f))
                  (quiz-spec-embedded-helper
                   (cdr h)
                   (update-counter-alist
                    c
                    (append
                     (if (equal a1 'depress) '((c 5)) (if (equal a1 'release) '((c 6)) nil))
                     (if (equal a2 'depress) '((c 7)) (if (equal a2 'release) '((c 8)) nil))))
                   path)))
             t))
        t)))
   (equal
    (quiz-spec-embedded-helper h c (cons 'none path))
    (let
        ((c5 (cadr (assoc '(c 5) c))) (c6 (cadr (assoc '(c 6) c)))
         (c7 (cadr (assoc '(c 7) c))) (c8 (cadr (assoc '(c 8) c)))
         (a1 (cadr (assoc 'a1 (cadr h)))) (a2 (cadr (assoc 'a2 (cadr h))))
         (oldl1 (cadr (assoc 'l1 (car h)))) (oldl2 (cadr (assoc 'l2 (car h))))
         (l1 (cadr (assoc 'l1 (cadr h)))) (l2 (cadr (assoc 'l2 (cadr h)))))
      (if (listp h)
          (and
           (or (lessp c6 40000) (lessp c8 40000) (and (equal oldl1 1) (equal oldl2 1)))
           (if (listp (cdr h))
               (let
                   ((p1 (implies (and (lessp 39999 c5) (equal oldl1 0)) (equal l1 0)))
                    (p2 (implies (and (lessp 39999 c7) (equal oldl2 0)) (equal l2 0))))
                 (and
                  (case (car path)
                        (none (and (equal a1 'f) (equal a2 'f)))
                        (one (and (equal a1 'depress) (equal a2 'f)))
                        (two (and (equal a1 'f) (equal a2 'depress)))
                        (both (and (equal a1 'depress) (equal a2 'depress)))
                        (otherwise f))
                  (quiz-spec-embedded-helper
                   (cdr h)
                   (update-counter-alist
                    c
                    (append
                     (if (equal a1 'depress) '((c 5)) (if (equal a1 'release) '((c 6)) nil))
```

```
                (if (equal a2 'depress) '((c 7)) (if (equal a2 'release) '((c 8)) nil))))
              path)))
        t))
    t)))
(equal
 (quiz-spec-embedded-helper h c (cons 'two path))
 (let
     ((c5 (cadr (assoc '(c 5) c))) (c6 (cadr (assoc '(c 6) c)))
      (c7 (cadr (assoc '(c 7) c))) (c8 (cadr (assoc '(c 8) c)))
      (a1 (cadr (assoc 'a1 (cadr h)))) (a2 (cadr (assoc 'a2 (cadr h))))
      (oldl1 (cadr (assoc 'l1 (car h)))) (oldl2 (cadr (assoc 'l2 (car h))))
      (l1 (cadr (assoc 'l1 (cadr h)))) (l2 (cadr (assoc 'l2 (cadr h)))))
   (if (listp h)
       (and
        (or (lessp c7 40000) (lessp c6 40000) (and (equal oldl1 1) (equal oldl2 0)))
        (if (listp (cdr h))
            (let
                ((p1 (implies (and (lessp 39999 c5) (equal oldl1 0)) (equal l1 0)))
                 (p2 (implies (and (lessp 39999 c7) (equal oldl2 0)) (equal l2 0))))
              (and
               (case (car path)
                     (none (and (equal a1 'f) (equal a2 'release)))
                     (one (and (equal a1 'depress) (equal a2 'release)))
                     (two (and (equal a1 'f) (equal a2 'f) p2))
                     (both (and (equal a1 'depress) (equal a2 'f) p2))
                     (otherwise f))
               (quiz-spec-embedded-helper
                (cdr h)
                (update-counter-alist
                 c
                 (append
                  (if (equal a1 'depress) '((c 5)) (if (equal a1 'release) '((c 6)) nil))
                  (if (equal a2 'depress) '((c 7)) (if (equal a2 'release) '((c 8)) nil))))
                path)))
          t))
     t)))
(equal
 (quiz-spec-embedded-helper h c (cons 'both path))
 (let
     ((c5 (cadr (assoc '(c 5) c))) (c6 (cadr (assoc '(c 6) c)))
      (c7 (cadr (assoc '(c 7) c))) (c8 (cadr (assoc '(c 8) c)))
      (a1 (cadr (assoc 'a1 (cadr h)))) (a2 (cadr (assoc 'a2 (cadr h))))
      (oldl1 (cadr (assoc 'l1 (car h)))) (oldl2 (cadr (assoc 'l2 (car h))))
      (l1 (cadr (assoc 'l1 (cadr h)))) (l2 (cadr (assoc 'l2 (cadr h)))))
   (if (listp h)
       (and
        (or (lessp c7 40000) (lessp c5 40000) (and (equal oldl1 0) (equal oldl2 1))
            (and (equal oldl1 1) (equal oldl2 0)))
        (if (listp (cdr h))
            (let
                ((p1 (implies (and (lessp 39999 c5) (equal oldl1 0)) (equal l1 0)))
                 (p2 (implies (and (lessp 39999 c7) (equal oldl2 0)) (equal l2 0))))
              (and
               (case (car path)
                     (none (and (equal a1 'release) (equal a2 'release)))
                     (one (and (equal a1 'f) (equal a2 'release) p1))
                     (two (and (equal a1 'release) (equal a2 'f) p2))
                     (both (and (equal a1 'f) (equal a2 'f) p1 p2))
                     (otherwise f))
               (quiz-spec-embedded-helper
                (cdr h)
                (update-counter-alist
```

```
                    c
                    (append
                     (if (equal a1 'depress) '((c 5)) (if (equal a1 'release) '((c 6)) nil))
                     (if (equal a2 'depress) '((c 7)) (if (equal a2 'release) '((c 8)) nil))))
                   path)))
             t))
        t))))
  ((disable-theory t)
   (enable quiz-spec-embedded-helper)
   (enable-theory ground-zero)))


;; We "use" the "first2" functions to control things in our big
;; theorem.  We don't want the inductive hyp to open it up, however,
;; so we prove some special-case theorems that will eliminate the term
;; in the inductive cases but not in the main case


(prove-lemma button1-spec-embedded-means-first2-special-rewrite (rewrite)
  (implies
   (button1-spec-embedded-path-helper (cdr h) c p)
   (button1-spec-embedded-path-helper-first2 (cdr h) c p))
  ((use (button1-spec-embedded-means-first2 (h (cdr h))))))

(prove-lemma button2-spec-embedded-means-first2-special-rewrite (rewrite)
  (implies
   (button2-spec-embedded-path-helper (cdr h) c p)
   (button2-spec-embedded-path-helper-first2 (cdr h) c p))
  ((use (button2-spec-embedded-means-first2 (h (cdr h))))))


(prove-lemma cdr-abstract-quiz-map (rewrite)
  (equal
   (cdr (abstract-quiz-map x y))
   (if (and (listp x) (listp y))
       (abstract-quiz-map (cdr x) (cdr y))
     0)))

(deftheory abstract-useful
  (abstract-quiz-map cdr-abstract-quiz-map car-abstract-quiz-map
   s-reflects-signal1-means quiz-program-spec-paths s-reflects-signal2-means
   l1-reflects-signals-means l2-reflects-signals-means length-button1-spec-path
   length-button2-spec-path length-quiz-program-spec-path length-quiz-spec-path
   equal-length-small length-nlistp-cdr l1-means-s-means nlistp-path-specs
   l2-means-s-means length lights-steady-means lessp-transitivity1))

(prove-lemma abstract-quiz-system-correctness-embedded
 (rewrite)
 (implies
     (and (button1-spec-embedded-path-helper h c1 path1)
          (button2-spec-embedded-path-helper h c2 path2)
          (quiz-program-spec-embedded-path-helper h c3 path3)
          (reasonable-sp (cadr (assoc 's (car h))))
          (assoc '(c 1) c1)
          (assoc '(c 2) c1)
          (assoc '(c 3) c2)
          (assoc '(c 4) c2)
          (assoc '(c 9) c3)
          (assoc '(c 5) c4)
          (assoc '(c 6) c4)
          (assoc '(c 7) c4)
          (assoc '(c 8) c4)
```

```
(numberp (cadr (assoc '(c 1) c1)))
(numberp (cadr (assoc '(c 2) c1)))
(numberp (cadr (assoc '(c 3) c2)))
(numberp (cadr (assoc '(c 4) c2)))
(numberp (cadr (assoc '(c 9) c3)))
(quiz-invariant-simple-program-props (car path3)
                                     (cadr (assoc '(c 9) c3)))
(quiz-invariant-button1-reflects-signal1 (car path1)
                                         (car path3)
                                         (cadr (assoc '(c 1) c1))
                                         (cadr (assoc '(c 2) c1))
                                         (cadr (assoc '(c 9) c3))
                                         (cadr (assoc 'b1 (car h))))
(quiz-invariant-button2-reflects-signal2 (car path2)
                                         (car path3)
                                         (cadr (assoc '(c 3) c2))
                                         (cadr (assoc '(c 4) c2))
                                         (cadr (assoc '(c 9) c3))
                                         (cadr (assoc 'b2 (car h))))
(quiz-invariant-s-reflects-signal1 (car path1)
                                   (car path3)
                                   (cadr (assoc '(c 1) c1))
                                   (cadr (assoc '(c 2) c1))
                                   (cadr (assoc '(c 9) c3))
                                   (cadr (assoc 's (car h))))
(quiz-invariant-s-reflects-signal2 (car path2)
                                   (car path3)
                                   (cadr (assoc '(c 3) c2))
                                   (cadr (assoc '(c 4) c2))
                                   (cadr (assoc '(c 9) c3))
                                   (cadr (assoc 's (car h))))
(quiz-invariant-l1-reflects-signals (car path1)
                                    (car path2)
                                    (car path3)
                                    (cadr (assoc '(c 1) c1))
                                    (cadr (assoc '(c 2) c1))
                                    (cadr (assoc '(c 3) c2))
                                    (cadr (assoc '(c 4) c2))
                                    (cadr (assoc '(c 9) c3))
                                    (cadr (assoc 's (car h)))
                                    (cadr (assoc 'l1 (car h))))
(quiz-invariant-l2-reflects-signals (car path1)
                                    (car path2)
                                    (car path3)
                                    (cadr (assoc '(c 1) c1))
                                    (cadr (assoc '(c 2) c1))
                                    (cadr (assoc '(c 3) c2))
                                    (cadr (assoc '(c 4) c2))
                                    (cadr (assoc '(c 9) c3))
                                    (cadr (assoc 's (car h)))
                                    (cadr (assoc 'l2 (car h))))
(quiz-invariant-l1-means-s (car path1)
                           (car path3)
                           (cadr (assoc '(c 1) c1))
                           (cadr (assoc '(c 9) c3))
                           (cadr (assoc 's (car h)))
                           (cadr (assoc 'l1 (car h))))
(quiz-invariant-l2-means-s (car path2)
                           (car path3)
                           (cadr (assoc '(c 3) c2))
                           (cadr (assoc '(c 9) c3))
                           (cadr (assoc 's (car h)))
```

```
                                 (cadr (assoc 'l2 (car h))))
          (equal (cadr (assoc '(c 1) c1))
                 (cadr (assoc '(c 5) c4)))
          (equal (cadr (assoc '(c 2) c1))
                 (cadr (assoc '(c 6) c4)))
          (equal (cadr (assoc '(c 3) c2))
                 (cadr (assoc '(c 7) c4)))
          (equal (cadr (assoc '(c 4) c2))
                 (cadr (assoc '(c 8) c4))))
     (quiz-spec-embedded-helper h c4
                               (abstract-quiz-map path1 path2)))
((instructions
  (use-lemma button1-spec-embedded-means-first2 ((c c1) (p path1)))
  (use-lemma button2-spec-embedded-means-first2 ((c c2) (p path2)))
  (induct (abstract-quiz-system-embedded-induct c1 c2 c3 c4 path1 path2 path3 h)) promote
  promote promote (contradict 3) (dive 1 1) hide-hyps show-hyps nx (dive 1) (dive 1)
  (rewrite button1-spec-embedded-path-helper-recurse) nx (dive 1)
  (rewrite button2-spec-embedded-path-helper-recurse) nx (dive 1)
  (rewrite quiz-program-spec-embedded-path-helper-recurse) nx (dive 1)
  (rewrite reasonable-sp-preserved) change-goal (contradict 9) (dive 1) x top s-prop split
  nx (dive 1) (rewrite assoc-update-counter-alist-iff-repeat) s nx (dive 1)
  (rewrite assoc-update-counter-alist-iff-repeat) s nx (dive 1)
  (rewrite assoc-update-counter-alist-iff-repeat) s nx (dive 1)
  (rewrite assoc-update-counter-alist-iff-repeat) s nx (dive 1)
  (rewrite assoc-update-counter-alist-iff-repeat) s nx (dive 1)
  (rewrite assoc-update-counter-alist-iff-repeat) s nx (dive 1)
  (rewrite assoc-update-counter-alist-iff-repeat) s nx (dive 1)
  (rewrite assoc-update-counter-alist-iff-repeat) s nx (dive 1)
  (rewrite assoc-update-counter-alist-iff-repeat) s nx (dive 1) hide-hyps (= t) nx
  (dive 1) (= t) nx (dive 1) (= t) nx (dive 1) (= t) nx (dive 1) (= t) nx (dive 1)
  show-hyps (rewrite quiz-invariant-simple-program-props-preserved) nx (dive 1)
  (rewrite button1-reflects-signal1-invariant-preserved) change-goal hide-hyps
  (show-hyps 1 2 4 6) bash change-goal s-prop split hide-hyps (show-hyps 37 38) bash
  (rewrite equal-update-counter-alist-nlistp) (dive 1)
  (rewrite listp-caddr-assoc-quiz-program-spec) top s
  (rewrite equal-update-counter-alist-nlistp) (dive 1)
  (rewrite listp-caddr-assoc-quiz-program-spec) top s nx (dive 1)
  (rewrite button2-reflects-signal2-invariant-preserved) change-goal hide-hyps
  (show-hyps 1 2 5 7) bash change-goal s-prop split hide-hyps (show-hyps 37 38) bash
  (rewrite equal-update-counter-alist-nlistp) (dive 1)
  (rewrite listp-caddr-assoc-quiz-program-spec) top s
  (rewrite equal-update-counter-alist-nlistp) (dive 1)
  (rewrite listp-caddr-assoc-quiz-program-spec) top s nx (dive 1)
  (rewrite s-reflects-signal1-invariant-preserved) change-goal hide-hyps
  (show-hyps 1 2 4 6) bash change-goal s-prop split hide-hyps (show-hyps 37 38) bash
  (rewrite equal-update-counter-alist-nlistp) (dive 1)
  (rewrite listp-caddr-assoc-quiz-program-spec) top s
  (rewrite equal-update-counter-alist-nlistp) (dive 1)
  (rewrite listp-caddr-assoc-quiz-program-spec) top s nx (dive 1)
  (rewrite s-reflects-signal2-invariant-preserved) change-goal hide-hyps
  (show-hyps 1 2 5 7) bash change-goal s-prop split hide-hyps (show-hyps 37 38) bash
  (rewrite equal-update-counter-alist-nlistp) (dive 1)
  (rewrite listp-caddr-assoc-quiz-program-spec) top s
  (rewrite equal-update-counter-alist-nlistp) (dive 1)
  (rewrite listp-caddr-assoc-quiz-program-spec) top s nx (dive 1)
  (rewrite l1-reflects-signals-invariant-preserved) change-goal
  (rewrite s-reflects-signal1-invariant-preserved) hide-hyps (show-hyps 1 2 4 6) bash
  s-prop split hide-hyps (show-hyps 37 38) bash
  (rewrite equal-update-counter-alist-nlistp) (dive 1)
  (rewrite listp-caddr-assoc-quiz-program-spec) top s (dive 1) top
  (rewrite equal-update-counter-alist-nlistp) (dive 1)
  (rewrite listp-caddr-assoc-quiz-program-spec) top s (change-goal (main . 2))
```

```
(change-goal ((main . 1) . 14)) (change-goal ((main . 1) . 13))
(change-goal ((main . 1) . 12)) (change-goal ((main . 1) . 11))
(rewrite s-reflects-signal2-invariant-preserved) hide-hyps (show-hyps 1 2 5 7) bash
s-prop split hide-hyps (show-hyps 37 38) bash
(rewrite equal-update-counter-alist-nlistp) (dive 1)
(rewrite listp-caddr-assoc-quiz-program-spec) top s
(rewrite equal-update-counter-alist-nlistp) (dive 1)
(rewrite listp-caddr-assoc-quiz-program-spec) top s hide-hyps (show-hyps 1 2 4 6) bash
hide-hyps (show-hyps 1 2 5 7) bash s-prop split hide-hyps (show-hyps 37 38) bash
(rewrite equal-update-counter-alist-nlistp) (dive 1)
(rewrite listp-caddr-assoc-quiz-program-spec) top s
(rewrite equal-update-counter-alist-nlistp) (dive 1)
(rewrite listp-caddr-assoc-quiz-program-spec) top s promote promote split x change-goal
nx (dive 1) (rewrite l2-reflects-signals-invariant-preserved) change-goal
(rewrite s-reflects-signal1-invariant-preserved) hide-hyps (show-hyps 1 2 4 6) bash
s-prop split hide-hyps (show-hyps 37 38) bash
(rewrite equal-update-counter-alist-nlistp) (dive 1)
(rewrite listp-caddr-assoc-quiz-program-spec) top s
(rewrite equal-update-counter-alist-nlistp) (dive 1)
(rewrite listp-caddr-assoc-quiz-program-spec) top s change-goal
(rewrite s-reflects-signal2-invariant-preserved) hide-hyps (show-hyps 1 2 5 7) bash
s-prop split hide-hyps (show-hyps 37 38) bash
(rewrite equal-update-counter-alist-nlistp) (dive 1)
(rewrite listp-caddr-assoc-quiz-program-spec) top s
(rewrite equal-update-counter-alist-nlistp) (dive 1)
(rewrite listp-caddr-assoc-quiz-program-spec) top s change-goal hide-hyps
(show-hyps 1 2 4 6) bash change-goal hide-hyps (show-hyps 1 2 4 6) hide-hyps
(show-hyps 1 2 5 7) bash change-goal s-prop split hide-hyps (show-hyps 37 38) bash
(rewrite equal-update-counter-alist-nlistp) (dive 1)
(rewrite listp-caddr-assoc-quiz-program-spec) top s
(rewrite equal-update-counter-alist-nlistp) (dive 1)
(rewrite listp-caddr-assoc-quiz-program-spec) top s nx (dive 1)
(rewrite quiz-invariant-l1-means-s-preserved) change-goal
(rewrite s-reflects-signal1-invariant-preserved) hide-hyps (show-hyps 1 2 4 6) bash
s-prop split hide-hyps (show-hyps 37 38) bash
(rewrite equal-update-counter-alist-nlistp) (dive 1)
(rewrite listp-caddr-assoc-quiz-program-spec) top s
(rewrite equal-update-counter-alist-nlistp) (dive 1)
(rewrite listp-caddr-assoc-quiz-program-spec) top s change-goal hide-hyps
(show-hyps 1 2 4 6) bash change-goal s-prop split hide-hyps (show-hyps 37 38) bash
(rewrite equal-update-counter-alist-nlistp) (dive 1)
(rewrite listp-caddr-assoc-quiz-program-spec) top s
(rewrite equal-update-counter-alist-nlistp) (dive 1)
(rewrite listp-caddr-assoc-quiz-program-spec) top s nx (dive 1)
(rewrite quiz-invariant-l2-means-s-preserved) change-goal
(rewrite s-reflects-signal2-invariant-preserved) hide-hyps (show-hyps 1 2 5 7) bash
s-prop split hide-hyps (show-hyps 37 38) bash
(rewrite equal-update-counter-alist-nlistp) (dive 1)
(rewrite listp-caddr-assoc-quiz-program-spec) top s
(rewrite equal-update-counter-alist-nlistp) (dive 1)
(rewrite listp-caddr-assoc-quiz-program-spec) top s change-goal hide-hyps
(show-hyps 1 2 5 7) bash change-goal s-prop split hide-hyps (show-hyps 37 38) bash
(rewrite equal-update-counter-alist-nlistp) (dive 1)
(rewrite listp-caddr-assoc-quiz-program-spec) top s
(rewrite equal-update-counter-alist-nlistp) (dive 1)
(rewrite listp-caddr-assoc-quiz-program-spec) top s nx (dive 1) hide-hyps
(show-hyps 10 11 12 13 14 15 16 17 18 33 34 35 36) (= t) nx (= t) show-hyps top (dive 1)
(dive 1) hide-hyps
(= * t
   ((disable-theory t)
    (enable-theory ground-zero)
    (enable button1-spec-embedded-means-first2-special-rewrite
```

```
            button2-spec-embedded-means-first2-special-rewrite)))
up s top show-hyps (contradict 3) (contradict 4) (dive 1) s-prop top (contradict 5)
(dive 1) s-prop top (contradict 4) (dive 3) x (dive 1)
(= * t
   ((disable-theory t)
    (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2))
    (enable-theory ground-zero abstract-useful)))
nx (dive 1)
(= * t
   ((disable-theory t)
    (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2))
    (enable-theory ground-zero abstract-useful)))
top x s-prop
(claim (not (and (listp path1)
                 (listp (cdr path1))
                 (listp path2)
                 (listp (cdr path2))))
       0)
(contradict 37) (drop 37) hide-hyps (show-hyps 1 2 6 7)
(prove (disable-theory t)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2
                                                         path2))
       (enable-theory ground-zero abstract-useful))
split
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
```

```
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
```

```
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
```

```
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
```

```
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
```

```
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
```

```
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
```

```
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
```

```
         (button1-spec-embedded-path-helper-first2 h c1 path1)
         (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
         (button1-spec-embedded-path-helper-first2 h c1 path1)
         (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
         (button1-spec-embedded-path-helper-first2 h c1 path1)
         (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
         (button1-spec-embedded-path-helper-first2 h c1 path1)
         (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
         (button1-spec-embedded-path-helper-first2 h c1 path1)
         (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
         (button1-spec-embedded-path-helper-first2 h c1 path1)
         (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
         (button1-spec-embedded-path-helper-first2 h c1 path1)
         (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
         (button1-spec-embedded-path-helper-first2 h c1 path1)
         (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
         (button1-spec-embedded-path-helper-first2 h c1 path1)
         (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
         (button1-spec-embedded-path-helper-first2 h c1 path1)
         (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
         (button1-spec-embedded-path-helper-first2 h c1 path1)
         (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
         (button1-spec-embedded-path-helper-first2 h c1 path1)
         (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
         (button1-spec-embedded-path-helper-first2 h c1 path1)
         (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
         (button1-spec-embedded-path-helper-first2 h c1 path1)
         (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
         (button1-spec-embedded-path-helper-first2 h c1 path1)
         (button2-spec-embedded-path-helper-first2 h c2 path2)))
```

```
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
```

```
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
              (button1-spec-embedded-path-helper-first2 h c1 path1)
              (button2-spec-embedded-path-helper-first2 h c2 path2)))
```

```
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
```

```
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
```

```
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
```

```
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
```

```
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
```

```
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
```

```
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
```

```
             (button1-spec-embedded-path-helper-first2 h c1 path1)
             (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
             (button1-spec-embedded-path-helper-first2 h c1 path1)
             (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
             (button1-spec-embedded-path-helper-first2 h c1 path1)
             (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
             (button1-spec-embedded-path-helper-first2 h c1 path1)
             (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
             (button1-spec-embedded-path-helper-first2 h c1 path1)
             (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
             (button1-spec-embedded-path-helper-first2 h c1 path1)
             (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
             (button1-spec-embedded-path-helper-first2 h c1 path1)
             (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
             (button1-spec-embedded-path-helper-first2 h c1 path1)
             (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
             (button1-spec-embedded-path-helper-first2 h c1 path1)
             (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
             (button1-spec-embedded-path-helper-first2 h c1 path1)
             (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
             (button1-spec-embedded-path-helper-first2 h c1 path1)
             (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
             (button1-spec-embedded-path-helper-first2 h c1 path1)
             (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
             (button1-spec-embedded-path-helper-first2 h c1 path1)
             (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
             (button1-spec-embedded-path-helper-first2 h c1 path1)
             (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
             (button1-spec-embedded-path-helper-first2 h c1 path1)
             (button2-spec-embedded-path-helper-first2 h c2 path2)))
```

```
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
```

```
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove  (disable-theory t) (enable-theory ground-zero abstract-useful)
        (expand (reasonable-sp (cadr (assoc 's (car h))))
             (button1-spec-embedded-path-helper-first2 h c1 path1)
             (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
            (button1-spec-embedded-path-helper-first2 h c1 path1)
            (button2-spec-embedded-path-helper-first2 h c2 path2)))
```

```
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
x split
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
       (expand (reasonable-sp (cadr (assoc 's (car h))))
               (button1-spec-embedded-path-helper-first2 h c1 path1)
               (button2-spec-embedded-path-helper-first2 h c2 path2)))
(prove (disable-theory t) (enable-theory ground-zero abstract-useful)
```

```
        (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
   (prove (disable-theory t) (enable-theory ground-zero abstract-useful)
        (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
   (prove (disable-theory t) (enable-theory ground-zero abstract-useful)
        (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
   (prove (disable-theory t) (enable-theory ground-zero abstract-useful)
        (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
   (prove (disable-theory t) (enable-theory ground-zero abstract-useful)
        (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
   (prove (disable-theory t) (enable-theory ground-zero abstract-useful)
        (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2)))
   (prove (disable-theory t) (enable-theory ground-zero abstract-useful)
        (expand (reasonable-sp (cadr (assoc 's (car h))))
                (button1-spec-embedded-path-helper-first2 h c1 path1)
                (button2-spec-embedded-path-helper-first2 h c2 path2))))))

(prove-lemma car-behaviors (rewrite)
  (and
   (equal (car (quiz-program-spec)) 'read-b1)
   (equal (car (quiz-spec)) 'none)
   (equal (car (button1-spec)) 'up)
   (equal (car (button2-spec)) 'up)))

(prove-lemma all-transitions-litatoms-behaviors (rewrite)
  (and
   (all-transitions-litatoms (cadr (quiz-program-spec)))
   (all-transitions-litatoms (cadr (quiz-spec)))
   (all-transitions-litatoms (cadr (button1-spec)))
   (all-transitions-litatoms (cadr (button2-spec)))))

(prove-lemma listp-find-path-helper (rewrite)
  (implies
   (satisfiesp-helper name pred trans h c b)
   (equal (listp (find-path-helper name pred trans h c b)) (listp h))))

(prove-lemma listp-find-path (rewrite)
  (implies
   (satisfiesp h b)
   (equal (listp (find-path h b)) (listp h))))

(prove-lemma abstract-quiz-system-correctness nil
  (implies
   (and
    (or (nlistp h)
        (and (equal (cadr (assoc 's (car h))) 0)
             (equal (cadr (assoc 'l1 (car h))) 1)
             (equal (cadr (assoc 'l2 (car h))) 1)))
    (satisfiesp h (button1-spec))
    (satisfiesp h (button2-spec))
    (satisfiesp h (quiz-program-spec)))
```

```
   (satisfiesp h (quiz-spec)))
  ((use (abstract-quiz-system-correctness-embedded
        (c1 '(((c 1) 0) ((c 2) 0))) (path1 (find-path h (button1-spec)))
        (c2 '(((c 3) 0) ((c 4) 0))) (path2 (find-path h (button2-spec)))
        (c3 '(((c 9) 0))) (path3 (find-path h (quiz-program-spec)))
        (c4 '(((c 8) 0) ((c 6) 0) ((c 5) 0) ((c 7) 0)))))
   (disable-theory t)
   (enable-theory ground-zero)
   (enable assoc quiz-invariant-simple-program-props
          quiz-invariant-button1-reflects-signal1 quiz-invariant-button2-reflects-signal2
          quiz-invariant-s-reflects-signal1 quiz-invariant-s-reflects-signal2
          quiz-invariant-l1-reflects-signals quiz-invariant-l2-reflects-signals
          quiz-invariant-l1-means-s quiz-invariant-l2-means-s
          all-transitions-litatoms-behaviors reasonable-sp car-find-path
          car-abstract-quiz-map satisfiesp-nlistp button1-spec-s-helper-from-embedded
          button2-spec-s-helper-from-embedded quiz-program-spec-s-helper-from-embedded
          quiz-spec-s-helper-from-embedded counters-of-state-list-quiz-spec car-quiz-spec
          car-assoc assoc-quiz-spec-iff state-pred all-cars-litatoms state-name
          behavior-state-list behavior-initial state-trans-list car-find-path
          no-litatom-cars all-transitions-uniquep car-behaviors listp-find-path
          assoc-constant-button1-spec assoc-constant-button2-spec
          assoc-constant-quiz-program-spec assoc-constant-quiz-spec
          all-transitions-litatoms counters-of-state-list make-list-alist cars-uniquep
          counters-of-term counters-of-translist counters-of-state counters-of-trans
          trans-resets trans-pred member-cars set-union)))

))
```

E.3 "quiz-prog.events"

```
(proveall "quiz-prog"
'(
#|
 Copyright (C) 1995 by Matthew Wilding and Computational Logic, Inc.
 All Rights Reserved.

 This script is hereby placed in the public domain, and therefore unlimited
 editing and redistribution is permitted.

 NO WARRANTY

 Matthew Wilding and Computational Logic, Inc. PROVIDES ABSOLUTELY NO
 WARRANTY.  THE EVENT SCRIPT IS PROVIDED "AS IS" WITHOUT WARRANTY OF
 ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
 ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE.  THE ENTIRE RISK AS TO THE QUALITY AND
 PERFORMANCE OF THE SCRIPT IS WITH YOU.  SHOULD THE SCRIPT PROVE
 DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR
 CORRECTION.

 IN NO EVENT WILL Matthew Wilding and Computational Logic, Inc. BE
 LIABLE TO YOU FOR ANY DAMAGES, ANY LOST PROFITS, LOST MONIES, OR
 OTHER SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE
 USE OR INABILITY TO USE THIS SCRIPT (INCLUDING BUT NOT LIMITED TO
 LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY
 THIRD PARTIES), EVEN IF YOU HAVE ADVISED US OF THE POSSIBILITY OF
 SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

This file contains the events that lead to the proof of
```

```
FM9001 reasonableness lemma for the quiz-show system

|#

;; We use the library from the proof of the abstract lemma
(note-lib "quiz-abstract")

(compile-uncompiled-defns "temp")

;; We modify the "quick and dirty" FM9001 assembler so that we can
;; load code at non-0 addresses.

(defn my-asm (list offset)
  (asm-list (update-list list (resolve-names list offset nil))))

;; A few convenient functions for displaying numbers

(defn bv-to-nat-with-trues (bv)
  (if (nlistp bv) 0
    (plus (if (car bv) 1 0)
          (times 2 (bv-to-nat-with-trues (cdr bv))))))

(defn bv-to-nat-with-trues-list (list)
  (if (listp list)
      (cons (bv-to-nat-with-trues (car list))
            (bv-to-nat-with-trues-list (cdr list)))
    nil))

(defn hex-ascii (n)
  (if (lessp n 10)
      (plus 48 n)
    (plus 55 n)))

(defn bv-to-hex-list (bv)
  (if (listp bv)
    (append (bv-to-hex-list (cddddr bv))
            (cons (hex-ascii (bv-to-nat-with-trues (firstn 4 bv))) 0))
    nil))

(defn bv-to-hex (bv)
  (pack (cons 72 (bv-to-hex-list bv))))

(defn bv-list-to-hex (bvl)
  (if (listp bvl)
      (cons (bv-to-hex (car bvl))
            (bv-list-to-hex (cdr bvl)))
    nil))

;; The FM9001 quiz system

(defn quiz-prog ()
  (my-asm
   '(
   ; r0: temp reg
   ; r1, r2: button values
   ; r3: pick (0= neither picked)
   ; r4, r5: button addresses
   ; r6, r7: light addresses

   ; load constants
   (move t f r3 0)
   (move t f r4 (pc+))
```

```
#xc0000000
(move t f r5 (pc+))
#xc0000002
(move t f r6 (pc+))
#xc0000001
(move t f r7 (pc+))
#xc0000003
(move t f (r6) 1)
(move t f (r7) 1)

loop

; read button values
(move t f r1 (r4))     ;4000000b
(move t f r2 (r5))

; if button 2 currently picked handle separately
(xor f z r3 2)
(move eq f pc (pc+))
(value two-picked)

; button1 or no button selected.  Select 1 if button 1 is high,
; else 2 if button 2 high, else 0.
(asr f c r1 r1)        ; set carry flag if b1 is high
(move t f r0 1)
(move cs f pc (pc+))
(value update-status)
(asr f c r2 r2)        ; set carry flag if b2 is high
(move t f r0 2)
(move cs f pc (pc+))
(value update-status)
(move t f r0 0)
(move t f pc (pc+))
(value update-status)

two-picked
; button2 selected.  Select 2 if button 2 is high, else 1 if button
; 1 is high, else 0.
(asr f c r2 r2)        ; set carry flag if b2 high
(move t f r0 2)
(move cs f pc (pc+))
(value update-status)
(asr f c r1 r1)        ; set carry flag if b1 high
(move t f r0 1)
(move cs f pc (pc+))
(value update-status)
(move t f r0 0)

update-status
(move t f r3 r0)

; update lights depending on selected button
(move t f r0 1)
(xor f z r3 1)
(move eq f r0 0)
(move t f (r6) r0)
(move t f r0 1)
(xor f z r3 2)
(move eq f r0 0)
(move t f (r7) r0)

(move t f pc (pc+))
```

```
   (value loop))
   #x40000000))

#|
>(r-loop)

Trace Mode: Off   Abbreviated Output Mode:  On
Type ? for help.
*(bv-list-to-hex (quiz-prog))
 '(H00E00E00 H00E0103F HC0000000 H00E0143F HC0000002 H00E0183F HC0000001
             H00E01C3F HC0000003 H00E05A01 H00E05E01 H00E00414 H00E00815
             H0BF10E02 H00703C3F H4000001B H09F80401 H00E00201 H00103C3F
             H40000024 H09F80802 H00E00202 H00103C3F H40000024 H00E00200
             H00E03C3F H40000024 H09F80802 H00E00202 H00103C3F H40000024
             H09F80401 H00E00201 H00103C3F H40000024 H00E00200 H00E00C00
             H00E00201 H0BF10E01 H00700200 H00E05800 H00E00201 H0BF10E02
             H00700200 H00E05C00 H00E03C3F H4000000B)
|#
(defn quiz-initial-state ()
  (list
   (list (write-mem-ignore-stubs
          (nat-to-v 15 4)
          (list-to-mem (repeat 15 (nat-to-v 0 32)) 4 (repeat 32 f))
          (nat-to-v #x40000000 32))
         (list t f f f))
   (write-array-memory
    (repeat 4 (nat-to-v 0 32))
    #xc0000000
    (write-array-memory
     (quiz-prog) #x40000000
     (empty-memory 32)))))

#|

We write a function that makes calculating instruction timings easier

(defn quiz-state-setup (pc flags)
  (list
   (list (write-mem-ignore-stubs
          (nat-to-v 15 4)
          (list-to-mem (repeat 15 (nat-to-v 0 32)) 4 (repeat 32 f))
          (nat-to-v pc 32))
         flags)
   (write-array-memory
    (repeat 4 (nat-to-v 0 32))
    #xc0000000
    (write-array-memory
     (quiz-prog) #x40000000
     (empty-memory 32)))))

(iterate for x from #x40000000 to #x4000002D
         do
         (let ((s-off (*1*quiz-state-setup
                        x (list (*1*false) (*1*false) (*1*false) (*1*false) )))
               (s-on (*1*quiz-state-setup x (list  (*1*true) (*1*true) (*1*true) (*1*true)))))
           (format t "~&~X   OFF:(~A,~A) ON:(~A,~A)"
                  x (*1*init-time s-off 3) (*1*microcycles-constant s-off 3)
                  (*1*init-time s-on 3) (*1*microcycles-constant s-on 3))))

|#

;; initialization time of current instruction
```

```
(defn quiz-i (pc z c)
  (case pc
        (#x4000000b 11)
        (#x4000000c 11)
        (#x4000000d 1)
        (#x4000000e (if z 11 1))
        (#x40000010 1)
        (#x40000011 1)
        (#x40000012 (if c 11 1))
        (#x40000014 1)
        (#x40000015 1)
        (#x40000016 (if c 11 1))
        (#x40000018 1)
        (#x40000019 11)
        (#x4000001b 1)
        (#x4000001c 1)
        (#x4000001d (if c 11 1))
        (#x4000001f 1)
        (#x40000020 1)
        (#x40000021 (if c 11 1))
        (#x40000023 1)
        (#x40000024 1)
        (#x40000025 1)
        (#x40000026 1)
        (#x40000027 1)
        (#x40000028 13)
        (#x40000029 1)
        (#x4000002a 1)
        (#x4000002b 1)
        (#x4000002c 13)
        (#x4000002d 11)
        (otherwise 1)))

;; total time of current instruction
(defn quiz-t (pc z c)
  (case pc
        (#x4000000b 16)
        (#x4000000c 16)
        (#x4000000d 11)
        (#x4000000e (if z 16 10))
        (#x40000010 10)
        (#x40000011 10)
        (#x40000012 (if c 16 10))
        (#x40000014 10)
        (#x40000015 10)
        (#x40000016 (if c 16 10))
        (#x40000018 10)
        (#x40000019 16)
        (#x4000001b 10)
        (#x4000001c 10)
        (#x4000001d (if c 16 10))
        (#x4000001f 10)
        (#x40000020 10)
        (#x40000021 (if c 16 10))
        (#x40000023 10)
        (#x40000024 10)
        (#x40000025 10)
        (#x40000026 11)
        (#x40000027 (if z 10 8))
        (#x40000028 16)
        (#x40000029 10)
        (#x4000002a 11)
```

```
        (#x4000002b (if z 10 8))
        (#x4000002c 16)
        (#x4000002d 16)
        (otherwise 1)))

(prove-lemma lessp-0-quiz-i (rewrite)
  (lessp 0 (quiz-i pc z c)))

(prove-lemma lessp-0-quiz-t (rewrite)
  (lessp 0 (quiz-t pc z c)))

(disable quiz-t)
(disable quiz-i)

;; We write an Nqthm function that mimics the behavior of the quiz
;; program running on an FM9001.

(defn quiz-history (pc s b1 b2 s1-list s2-list l1 l2)
  (if (listp s1-list)
      (cons
       (list
        (list 'pc pc)
        (list 's s)
        (list 'b1 b1)
        (list 'b2 b2)
        (list 'signal1 (car s1-list))
        (list 'signal2 (car s2-list))
        (list 'l1 l1)
        (list 'l2 l2))
       (quiz-history pc s b1 b2 (cdr s1-list) (cdr s2-list) l1 l2))
    nil))

(defn newpc (pc z c)
  (cond
   ((equal pc #x4000000e) (if z #x4000001b (add1 (add1 pc))))
   ((equal pc #x40000012) (if c #x40000024 (add1 (add1 pc))))
   ((equal pc #x40000016) (if c #x40000024 (add1 (add1 pc))))
   ((equal pc #x40000019) #x40000024)
   ((equal pc #x4000001d) (if c #x40000024 (add1 (add1 pc))))
   ((equal pc #x40000021) (if c #x40000024 (add1 (add1 pc))))
   ((equal pc #x4000002d) #x4000000b)
   (t (add1 pc))))

(defn newtemp (pc temp z)
  (cond
   ((equal pc #x40000011) 1)
   ((equal pc #x40000015) 2)
   ((equal pc #x40000018) 0)
   ((equal pc #x4000001c) 2)
   ((equal pc #x40000020) 1)
   ((equal pc #x40000023) 0)
   ((equal pc #x40000025) 1)
   ((and (equal pc #x40000027) z) 0)
   ((equal pc #x40000029) 1)
   ((and (equal pc #x4000002b) z) 0)
   (t temp)))

(defn newz (pc z s)
  (cond
   ((equal pc #x4000000d) (equal s 2))
   ((equal pc #x40000026) (equal s 1))
   ((equal pc #x4000002a) (equal s 2))
```

```
   (t (b-buf z))))

(defn newc (pc c b1 b2)
  (cond
   ((equal pc #x40000010) (equal (remainder b1 2) 1))
   ((equal pc #x40000014) (equal (remainder b2 2) 1))
   ((equal pc #x4000001b) (equal (remainder b2 2) 1))
   ((equal pc #x4000001f) (equal (remainder b1 2) 1))
   (t (b-buf c))))

(defn quiz-program-operation (pc s b1 b2 s1-list s2-list l1 l2 temp z c)
  (let ((init (quiz-i pc z c)) (total (quiz-t pc z c)))
    (if (not (lessp init (length s1-list)))
        (quiz-history pc s b1 b2 s1-list s2-list l1 l2)
      (let
          ((newpc (newpc pc z c))
           (news (if (equal pc #x40000024) temp s))
           (newb1 (if (equal pc #x4000000b) (nth (sub1 init) s1-list) b1))
           (newb2 (if (equal pc #x4000000c) (nth (sub1 init) s2-list) b2))
           (newl1 (if (equal pc #x40000028) temp l1))
           (newl2 (if (equal pc #x4000002c) temp l2))
           (newtemp (newtemp pc temp z))
           (newz (newz pc z s))
           (newc (newc pc c b1 b2)))
        (if (not (lessp total (length s1-list)))
            (append
             (quiz-history pc s b1 b2 (firstn init s1-list) s2-list l1 l2)
             (quiz-history newpc news newb1 newb2 (nthcdr init s1-list)
                           (nthcdr init s2-list) newl1 newl2))
          (append
           (quiz-history pc s b1 b2 (firstn init s1-list) s2-list l1 l2)
           (append
            (quiz-history newpc news newb1 newb2
                          (firstn (difference total init) (nthcdr init s1-list))
                          (nthcdr init s2-list) newl1 newl2)
            (quiz-program-operation
             newpc news newb1 newb2 (nthcdr total s1-list)
             (nthcdr total s2-list) newl1 newl2 newtemp newz newc)))))))
  ((lessp (length s1-list))))

;; We prove that the quiz operation function is not hung up on whether
;; the flags are boolean

(prove-lemma quiz-i-b-buf (rewrite)
  (and
   (equal (quiz-i pc (b-buf z) c) (quiz-i pc z c))
   (equal (quiz-i pc z (b-buf c)) (quiz-i pc z c)))
  ((enable quiz-i b-buf)))

(prove-lemma quiz-t-b-buf (rewrite)
  (and
   (equal (quiz-t pc (b-buf z) c) (quiz-t pc z c))
   (equal (quiz-t pc z (b-buf c)) (quiz-t pc z c)))
  ((enable quiz-t b-buf)))

(prove-lemma newpc-b-buf (rewrite)
  (and
   (equal (newpc pc (b-buf z) c) (newpc pc z c))
   (equal (newpc pc z (b-buf c)) (newpc pc z c))))

(prove-lemma newtemp-b-buf (rewrite)
  (equal (newtemp pc temp (b-buf z)) (newtemp pc temp z)))
```

```
(prove-lemma quiz-program-operation-b-buf1 (rewrite)
   (equal
    (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp (b-buf z) c)
    (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp z c))
  ((induct (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp z c))
   (disable-theory t)
   (enable-theory ground-zero)
   (enable quiz-program-operation quiz-i-b-buf quiz-t-b-buf newpc-b-buf newtemp-b-buf newz)))

(prove-lemma quiz-program-operation-b-buf2 (rewrite)
   (equal
    (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp z (b-buf c))
    (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp z c))
  ((induct (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp z c))
   (disable-theory t)
   (enable-theory ground-zero)
   (enable quiz-program-operation quiz-i-b-buf quiz-t-b-buf newpc-b-buf newtemp-b-buf newc)))

;; we define the invariants that are needed for the quiz application
;; to work as expected.  Later we shall define the invariants that are
;; needed to show that the program solves the problem: for now, we're
;; content to list the things that allow the FM9001 to work reasonably
;; so that we can show that the Nqthm-embedded version is equivalent.

(defn quiz-program-functioning-invariants (s)
    (let
      ((mem (cadr s))
       (regs (caar s))
       (pc (v-to-nat (read-mem (nat-to-v 15 4) (caar s)))))

      (and
       ;; The quiz program is loaded

       (equal (read-mem (nat-to-v #x4000000b 32) mem) (nth 11 (quiz-prog)))
       (equal (read-mem (nat-to-v #x4000000c 32) mem) (nth 12 (quiz-prog)))
       (equal (read-mem (nat-to-v #x4000000d 32) mem) (nth 13 (quiz-prog)))
       (equal (read-mem (nat-to-v #x4000000e 32) mem) (nth 14 (quiz-prog)))
       (equal (read-mem (nat-to-v #x4000000f 32) mem) (nth 15 (quiz-prog)))
       (equal (read-mem (nat-to-v #x40000010 32) mem) (nth 16 (quiz-prog)))
       (equal (read-mem (nat-to-v #x40000011 32) mem) (nth 17 (quiz-prog)))
       (equal (read-mem (nat-to-v #x40000012 32) mem) (nth 18 (quiz-prog)))
       (equal (read-mem (nat-to-v #x40000013 32) mem) (nth 19 (quiz-prog)))
       (equal (read-mem (nat-to-v #x40000014 32) mem) (nth 20 (quiz-prog)))
       (equal (read-mem (nat-to-v #x40000015 32) mem) (nth 21 (quiz-prog)))
       (equal (read-mem (nat-to-v #x40000016 32) mem) (nth 22 (quiz-prog)))
       (equal (read-mem (nat-to-v #x40000017 32) mem) (nth 23 (quiz-prog)))
       (equal (read-mem (nat-to-v #x40000018 32) mem) (nth 24 (quiz-prog)))
       (equal (read-mem (nat-to-v #x40000019 32) mem) (nth 25 (quiz-prog)))
       (equal (read-mem (nat-to-v #x4000001a 32) mem) (nth 26 (quiz-prog)))
       (equal (read-mem (nat-to-v #x4000001b 32) mem) (nth 27 (quiz-prog)))
       (equal (read-mem (nat-to-v #x4000001c 32) mem) (nth 28 (quiz-prog)))
       (equal (read-mem (nat-to-v #x4000001d 32) mem) (nth 29 (quiz-prog)))
       (equal (read-mem (nat-to-v #x4000001e 32) mem) (nth 30 (quiz-prog)))
       (equal (read-mem (nat-to-v #x4000001f 32) mem) (nth 31 (quiz-prog)))
       (equal (read-mem (nat-to-v #x40000020 32) mem) (nth 32 (quiz-prog)))
       (equal (read-mem (nat-to-v #x40000021 32) mem) (nth 33 (quiz-prog)))
       (equal (read-mem (nat-to-v #x40000022 32) mem) (nth 34 (quiz-prog)))
       (equal (read-mem (nat-to-v #x40000023 32) mem) (nth 35 (quiz-prog)))
       (equal (read-mem (nat-to-v #x40000024 32) mem) (nth 36 (quiz-prog)))
       (equal (read-mem (nat-to-v #x40000025 32) mem) (nth 37 (quiz-prog)))
       (equal (read-mem (nat-to-v #x40000026 32) mem) (nth 38 (quiz-prog)))
```

```
       (equal (read-mem (nat-to-v #x40000027 32) mem) (nth 39 (quiz-prog)))
       (equal (read-mem (nat-to-v #x40000028 32) mem) (nth 40 (quiz-prog)))
       (equal (read-mem (nat-to-v #x40000029 32) mem) (nth 41 (quiz-prog)))
       (equal (read-mem (nat-to-v #x4000002a 32) mem) (nth 42 (quiz-prog)))
       (equal (read-mem (nat-to-v #x4000002b 32) mem) (nth 43 (quiz-prog)))
       (equal (read-mem (nat-to-v #x4000002c 32) mem) (nth 44 (quiz-prog)))
       (equal (read-mem (nat-to-v #x4000002d 32) mem) (nth 45 (quiz-prog)))
       (equal (read-mem (nat-to-v #x4000002e 32) mem) (nth 46 (quiz-prog)))

       ;; The memory-mapped input locations are 32-bit RAM
       (equal (length (read-mem (nat-to-v #xc0000000 32) mem)) 32)
       (ramp-mem (nat-to-v #xc0000000 32) mem)
       (equal (length (read-mem (nat-to-v #xc0000002 32) mem)) 32)
       (ramp-mem (nat-to-v #xc0000002 32) mem)

       ;; The memory-mapped output locations are RAM
       (ramp-mem (nat-to-v #xc0000001 32) mem)
       (ramp-mem (nat-to-v #xc0000003 32) mem)

       ;; The registers we use are 32-bit RAM
       (ramp-mem (nat-to-v 0 4) regs)
       (ramp-mem (nat-to-v 1 4) regs)
       (ramp-mem (nat-to-v 2 4) regs)
       (ramp-mem (nat-to-v 3 4) regs)
       (ramp-mem (nat-to-v 15 4) regs)
       (equal (length (read-mem (nat-to-v 0 4) regs)) 32)
       (equal (length (read-mem (nat-to-v 1 4) regs)) 32)
       (equal (length (read-mem (nat-to-v 2 4) regs)) 32)
       (equal (length (read-mem (nat-to-v 3 4) regs)) 32)
       (equal (length (read-mem (nat-to-v 15 4) regs)) 32)

       ;; Some registers have memory-mapped I/O device addresses
       (equal (read-mem (nat-to-v 4 4) regs) (nat-to-v #xc0000000 32))
       (equal (read-mem (nat-to-v 5 4) regs) (nat-to-v #xc0000002 32))
       (equal (read-mem (nat-to-v 6 4) regs) (nat-to-v #xc0000001 32))
       (equal (read-mem (nat-to-v 7 4) regs) (nat-to-v #xc0000003 32))

       ;; pc points to instruction within loop
       (or
        (equal pc #x4000000b) (equal pc #x4000000c)
        (equal pc #x4000000d) (equal pc #x4000000e)
        (equal pc #x40000010) (equal pc #x40000011)
        (equal pc #x40000012) (equal pc #x40000014)
        (equal pc #x40000015) (equal pc #x40000016)
        (equal pc #x40000018) (equal pc #x40000019)
        (equal pc #x4000001b) (equal pc #x4000001c)
        (equal pc #x4000001d) (equal pc #x4000001f)
        (equal pc #x40000020) (equal pc #x40000021)
        (equal pc #x40000023) (equal pc #x40000024)
        (equal pc #x40000025) (equal pc #x40000026)
        (equal pc #x40000027) (equal pc #x40000028)
        (equal pc #x40000029) (equal pc #x4000002a)
        (equal pc #x4000002b) (equal pc #x4000002c)
        (equal pc #x4000002d)))))

;; check to see that the functioning invariants hold after initialization.
(prove-lemma quiz-functioning-invariants-initial nil
  (quiz-program-functioning-invariants
   (fm9001-step
    (fm9001-step
     (fm9001-step
      (fm9001-step
```

```
       (fm9001-step
        (fm9001-step
         (fm9001-step
          (quiz-initial-state)
          (nat-to-v 15 4))
         (nat-to-v 15 4))
        (nat-to-v 15 4))
       (nat-to-v 15 4))
      (nat-to-v 15 4))
     (nat-to-v 15 4))
    (nat-to-v 15 4))))

(enable regs)
(enable *1*nth)

(prove-lemma init-time-is-quiz-i (rewrite)
  (implies
   (and
    (quiz-program-functioning-invariants s)
    (equal mat 3))
   (equal
    (init-time s mat)
    (quiz-i (v-to-nat (read-mem (nat-to-v 15 4) (regs (car s))))
            (z-flag (flags (car s)))
            (c-flag (flags (car s))))))
  ((enable init-time store-resultp flags z-flag c-flag quiz-i map-down)))

(prove-lemma microcycles-constant-is-quiz-t (rewrite)
  (implies
   (and
    (quiz-program-functioning-invariants s)
    (equal mat 3))
   (equal
    (microcycles-constant s mat)
    (quiz-t (v-to-nat (read-mem (nat-to-v 15 4) (regs (car s))))
            (z-flag (flags (car s)))
            (c-flag (flags (car s))))))
  ((enable microcycles-constant store-resultp flags z-flag c-flag quiz-t map-down)))

(prove-lemma read-write-statep-from-quiz-program-functioning (rewrite)
  (implies
   (quiz-program-functioning-invariants s)
   (not (read-write-statep s)))
  ((enable read-write-statep)))

(disable read-write-statep)

(prove-lemma length-list-of-cadrs (rewrite)
  (equal (length (list-of-cadrs x)) (length x)))

(prove-lemma listp-list-of-cadrs (rewrite)
  (equal (listp (list-of-cadrs x)) (listp x)))

(prove-lemma cdr-list-of-cadrs (rewrite)
  (equal (cdr (list-of-cadrs x))
         (if (nlistp x) 0 (list-of-cadrs (cdr x)))))

(prove-lemma car-list-of-cadrs (rewrite)
  (equal (car (list-of-cadrs x)) (cadar x)))

(prove-lemma length-list-of-cars (rewrite)
  (equal (length (list-of-cars x)) (length x)))
```

```
(prove-lemma listp-list-of-cars (rewrite)
  (equal (listp (list-of-cars x)) (listp x)))

(prove-lemma cdr-list-of-cars (rewrite)
  (equal (cdr (list-of-cars x))
         (if (nlistp x) 0 (list-of-cars (cdr x)))))

(prove-lemma car-list-of-cars (rewrite)
  (equal (car (list-of-cars x)) (caar x)))

(prove-lemma firstn-list-of-cadrs (rewrite)
  (equal (firstn n (list-of-cadrs x))
         (list-of-cadrs (firstn n x)))
  ((enable firstn)))

(prove-lemma firstn-list-of-cars (rewrite)
  (equal (firstn n (list-of-cars x))
         (list-of-cars (firstn n x)))
  ((enable firstn)))

(enable *1*v-iff)

;; Some of the invariant properties we want to use without opening up
;; the whole list
(prove-lemma quiz-program-functioning-invariants-means (rewrite)
  (implies
   (quiz-program-functioning-invariants s)
   (let
       ((mem (cadr s))
        (regs (caar s)))

      (and
       ;; the memory-mapped input locations are 32-bit ram
       (equal
        (length
         (read-mem
            (list f f f f f f f f f f f f f f f f f f f f f f f f f f f f f f t t) mem))
        32)
       (ramp-mem (list f f f f f f f f f f f f f f f f f f f f f f f f f f f f f f t t) mem)
       (equal
        (length
         (read-mem
             (list f t f f f f f f f f f f f f f f f f f f f f f f f f f f f f t t) mem))
        32)
       (ramp-mem (list f t f f f f f f f f f f f f f f f f f f f f f f f f f f f f t t) mem)

       ;; the memory-mapped output locations are ram
       (ramp-mem (list t f f f f f f f f f f f f f f f f f f f f f f f f f f f f f t t) mem)
       (ramp-mem (list t t f f f f f f f f f f f f f f f f f f f f f f f f f f f f t t) mem)

       ;; the registers we use are 32-bit ram
       (ramp-mem (list f f f f) regs)
       (ramp-mem (list t f f f) regs)
       (ramp-mem (list f t f f) regs)
       (ramp-mem (list t t f f) regs)
       (ramp-mem (list t t t t) regs)
       (equal (length (read-mem (list f f f f) regs)) 32)
       (equal (length (read-mem (list t f f f) regs)) 32)
       (equal (length (read-mem (list f t f f) regs)) 32)
       (equal (length (read-mem (list t t f f) regs)) 32)
       (equal (length (read-mem (list t t t t) regs)) 32)))))
```

```
(disable quiz-program-functioning-invariants-means)

(prove-lemma repeat-with-inputs-quiz-embedded (rewrite)
  (implies
   (and
    (quiz-program-functioning-invariants s)
    (variable-not-overflowed-in-historyp 'signal1 h)
    (variable-not-overflowed-in-historyp 'signal2 h))
   (equal
    (repeat-with-inputs
     s
     (extract-from-history '((#xc0000000 signal1) (#xc0000002 signal2)) h)
     '(((15) pc) ((3) s) ((1) b1) ((2) b2) (#xc0000000 signal1) (#xc0000002 signal2)
       (#xc0000001 l1) (#xc0000003 l2)))
    (quiz-history
     (v-to-nat (read-mem (nat-to-v 15 4) (regs (car s))))
     (v-to-nat (read-mem (nat-to-v 3 4) (regs (car s))))
     (v-to-nat (read-mem (nat-to-v 1 4) (regs (car s))))
     (v-to-nat (read-mem (nat-to-v 2 4) (regs (car s))))
     (list-of-cadrs (list-of-cars (extract-from-history '((#xc0000000 signal1)) h)))
     (list-of-cadrs (list-of-cars (extract-from-history '((#xc0000002 signal2)) h)))
     (v-to-nat (read-mem (nat-to-v #xc0000001 32) (cadr s)))
     (v-to-nat (read-mem (nat-to-v #xc0000003 32) (cadr s))))))
  ((induct (length h))
   (disable quiz-program-functioning-invariants)
   (enable quiz-program-functioning-invariants-means)))


(prove-lemma quiz-history-make-signal2-smaller (rewrite)
  (implies
   (lessp (length signal1) (length signal2))
   (equal (quiz-history pc r3 r1 r2 signal1 signal2 s1 s2)
          (quiz-history pc r3 r1 r2 signal1 (firstn (length signal1) signal2) s1 s2)))
  ((enable quiz-history)))

(prove-lemma equal-append-append (rewrite)
  (equal
   (equal (append a b) (append x y))
   (if (lessp (length a) (length x))
       (and
        (equal (make-properp a) (firstn (length a) x))
        (equal b (append (nthcdr (length a) x) y)))
     (and
      (equal (make-properp x) (firstn (length x) a))
      (equal y (append (nthcdr (length x) a) b)))))
  ((enable nthcdr firstn)
   (induct (double-cdr-induction x a))))

(prove-lemma length-quiz-history (rewrite)
  (equal (length (quiz-history pc r3 r1 r2 signal1 signal2 s1 s2)) (length signal1)))

(prove-lemma last-cdr-quiz-history (rewrite)
  (equal (last-cdr (quiz-history pc r3 r1 r2 signal1 signal2 s1 s2)) nil)
  ((enable last-cdr)))

(prove-lemma nthcdr-all (rewrite)
  (implies
   (equal (length l) n)
   (equal (nthcdr n l) (last-cdr l)))
  ((enable nthcdr last-cdr)))
```

```
(prove-lemma make-properp-quiz-history (rewrite)
  (equal (make-properp (quiz-history pc r3 r1 r2 signal1 signal2 s1 s2))
         (quiz-history pc r3 r1 r2 signal1 signal2 s1 s2)))

(prove-lemma quiz-history-make-properp (rewrite)
  (and
   (equal (quiz-history pc r3 r1 r2 (make-properp signal1) signal2 s1 s2)
          (quiz-history pc r3 r1 r2 signal1 signal2 s1 s2))
   (equal (quiz-history pc r3 r1 r2 signal1 (make-properp signal2) s1 s2)
          (quiz-history pc r3 r1 r2 signal1 signal2 s1 s2)))
  ((enable quiz-history)))


(prove-lemma quiz-history-signal2-append-smaller (rewrite)
  (implies
   (not (lessp (length a) (length signal1)))
   (equal (quiz-history pc r3 r1 r2 signal1 (append a b) s1 s2)
          (quiz-history pc r3 r1 r2 signal1 a s1 s2)))
  ((enable quiz-history)
   (induct (quiz-history pc r3 r1 r2 signal1 a s1 s2))))

(prove-lemma quiz-history-nil (rewrite)
  (equal (quiz-history pc r3 r1 r2 nil signal2 s1 s2) nil))

(prove-lemma nthcdr-0 (rewrite)
  (equal (nthcdr 0 l) l))

;; open up quiz-program-operation if in a term (equal q-p-o (append x y))
(prove-lemma equal-quiz-program-operation-append (rewrite)
  (equal
   (equal (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp z c)
          (append x y))
   (equal
    (let ((init (quiz-i pc z c)) (total (quiz-t pc z c)))
    (if (not (lessp init (length s1-list)))
        (quiz-history pc s b1 b2 s1-list s2-list l1 l2)
      (let
          ((newpc (newpc pc z c))
           (news (if (equal pc #x40000024) temp s))
           (newb1 (if (equal pc #x4000000b) (nth (sub1 init) s1-list) b1))
           (newb2 (if (equal pc #x4000000c) (nth (sub1 init) s2-list) b2))
           (newl1 (if (equal pc #x40000028) temp l1))
           (newl2 (if (equal pc #x4000002c) temp l2))
           (newtemp (newtemp pc temp z))
           (newz (newz pc z s))
           (newc (newc pc c b1 b2)))
        (if (not (lessp total (length s1-list)))
            (append
             (quiz-history pc s b1 b2 (firstn init s1-list) s2-list l1 l2)
             (quiz-history newpc news newb1 newb2 (nthcdr init s1-list)
                           (nthcdr init s2-list) newl1 newl2))
          (append
           (quiz-history pc s b1 b2 (firstn init s1-list) s2-list l1 l2)
           (append
            (quiz-history newpc news newb1 newb2
                          (firstn (difference total init) (nthcdr init s1-list))
                          (nthcdr init s2-list) newl1 newl2)
            (quiz-program-operation
             newpc news newb1 newb2 (nthcdr total s1-list)
             (nthcdr total s2-list) newl1 newl2 newtemp newz newc)))))))
    (append x y)))
  ((expand (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp z c))
```

```
   (disable-theory t)
   (enable-theory ground-zero)))


(prove-lemma quiz-program-operation-signal2-append-smaller nil
  (implies
   (equal (length signal1) (length a))
   (equal (quiz-program-operation pc r3 r1 r2 signal1 (append a b) s1 s2 s z c)
          (quiz-program-operation
           pc r3 r1 r2 signal1 a s1 s2 s z c)))
  ((induct (quiz-program-operation pc r3 r1 r2 signal1 a s1 s2 s z c))
   (expand (quiz-program-operation pc r3 r1 r2 signal1 (append a b) s1 s2 s z c))
   (disable-theory t)
   (enable-theory ground-zero)
   (enable quiz-program-operation quiz-history-signal2-append-smaller equal-append-append
           nth-firstn lessp-difference-cancellation length-quiz-history nthcdr-all
           *1*make-properp length firstn-0 nthcdr-0 append quiz-history-nil
           equal-quiz-program-operation-append nth-append nthcdr-firstn
           make-properp-quiz-history quiz-history-make-properp
           quiz-history-signal2-append-smaller firstn-too-big firstn-firstn length-nthcdr
           length-firstn last-cdr-quiz-history length-firstn firstn-append nthcdr-append)))

(prove-lemma quiz-program-operation-make-signal2-smaller (rewrite)
  (implies
   (lessp (length signal1) (length signal2))
   (equal (quiz-program-operation pc r3 r1 r2 signal1 signal2 s1 s2 s z c)
          (quiz-program-operation
           pc r3 r1 r2 signal1 (firstn (length signal1) signal2) s1 s2 s z c)))
  ((use (quiz-program-operation-signal2-append-smaller
         (a (firstn (length signal1) signal2))
         (b (nthcdr (length signal1) signal2))))
   (disable-theory t)
   (enable-theory ground-zero)
   (enable append-firstn-nthcdr length-firstn)))


(enable c-flag)
(enable z-flag)
(enable flags)

(prove-lemma v-adder-v-buf (rewrite)
  (equal (v-adder flag (v-buf x) y) (v-adder flag x y))
  ((enable v-adder v-buf)))

(prove-lemma v-inc-v-buf (rewrite)
  (equal (v-inc (v-buf x)) (v-inc x))
  ((enable v-inc v-adder-output length-v-buf v-adder-v-buf)
   (disable-theory t)
   (enable-theory ground-zero)))

(prove-lemma v-inc-when-length-and-value-known
             (rewrite)
             (implies (and (equal (length v) n)
                           (equal (v-to-nat v) n2))
                      (equal (v-inc v)
                             (v-inc (nat-to-v n2 n))))
             ((enable nat-to-v v-to-nat v-inc-v-buf)))

(prove-lemma quiz-program-functioning-invariants-preserved-by-update-state-with-inputs
     (rewrite)
  (implies
   (quiz-program-functioning-invariants s)
```

```
   (quiz-program-functioning-invariants
    (update-state-with-inputs
     (extract-from-assignment '((#xc0000000 signal1) (#xc0000002 signal2)) a)
     s))))


(prove-lemma ramp-fm9001-step-regs (rewrite)
  (equal (ramp-mem x (caar (fm9001-step s pc))) (ramp-mem x (caar s)))
  ((enable fm9001-step)))


;; These theorems should be less quiz-dependent, but the lemmas take
;; too long to prove that way, so I cheat a little.

(prove-lemma length-fm9001-step-r15 (rewrite)
  (implies
   (quiz-program-functioning-invariants s)
   (equal (length (read-mem (list t t t t) (caar (fm9001-step s (list t t t t))))) 32))
  ((enable fm9001-step quiz-program-functioning-invariants)))

(prove-lemma length-fm9001-step-r0 (rewrite)
  (implies
   (quiz-program-functioning-invariants s)
   (equal (length (read-mem (list f f f f) (caar (fm9001-step s (list t t t t))))) 32))
  ((enable fm9001-step quiz-program-functioning-invariants)))

(prove-lemma length-fm9001-step-r1 (rewrite)
  (implies
   (quiz-program-functioning-invariants s)
   (equal (length (read-mem (list t f f f) (caar (fm9001-step s (list t t t t))))) 32))
  ((enable fm9001-step quiz-program-functioning-invariants)))

(prove-lemma length-fm9001-step-r2 (rewrite)
  (implies
   (quiz-program-functioning-invariants s)
   (equal (length (read-mem (list f t f f) (caar (fm9001-step s (list t t t t))))) 32))
  ((enable fm9001-step quiz-program-functioning-invariants)))

(prove-lemma length-fm9001-step-r3 (rewrite)
  (implies
   (quiz-program-functioning-invariants s)
   (equal (length (read-mem (list t t f f) (caar (fm9001-step s (list t t t t))))) 32))
  ((enable fm9001-step quiz-program-functioning-invariants)))

(prove-lemma constants-preserved-fm9001-step-r4 (rewrite)
  (implies
   (quiz-program-functioning-invariants s)
    (equal (read-mem (list f f t f) (caar (fm9001-step s (list t t t t))))
           (read-mem (list f f t f) (caar s))))
  ((enable fm9001-step)))

(prove-lemma constants-preserved-fm9001-step-r5 (rewrite)
  (implies
   (quiz-program-functioning-invariants s)
    (equal (read-mem (list t f t f) (caar (fm9001-step s (list t t t t))))
           (read-mem (list t f t f) (caar s))))
  ((enable fm9001-step)))

(prove-lemma constants-preserved-fm9001-step-r6 (rewrite)
  (implies
   (quiz-program-functioning-invariants s)
    (equal (read-mem (list f t t f) (caar (fm9001-step s (list t t t t))))
```

```
          (read-mem (list f t t f) (caar s))))
  ((enable fm9001-step)))

(prove-lemma constants-preserved-fm9001-step-r7 (rewrite)
  (implies
   (quiz-program-functioning-invariants s)
    (equal (read-mem (list t t t f) (caar (fm9001-step s (list t t t t))))
          (read-mem (list t t t f) (caar s))))
  ((enable fm9001-step)))

(prove-lemma length-fm9001-step-mem-l1 (rewrite)
  (implies
   (quiz-program-functioning-invariants s)
   (equal
    (length
     (read-mem (list f f f f f f f f f f f f f f f f f f f f f f f f f f f f f f t t)
              (cadr (fm9001-step s (list t t t t)))))
    32))
  ((enable fm9001-step)))

(prove-lemma length-fm9001-step-mem-l2 (rewrite)
  (implies
   (quiz-program-functioning-invariants s)
   (equal
    (length
     (read-mem (list f t f f f f f f f f f f f f f f f f f f f f f f f f f f f f t t)
              (cadr (fm9001-step s (list t t t t)))))
    32))
  ((enable fm9001-step)))

(prove-lemma pc-stays-in-bounds-fm9001-step (rewrite)
  (implies
   (quiz-program-functioning-invariants s)
   (equal (v-to-nat (read-mem (list t t t t) (caar (fm9001-step s (list t t t t)))))
         (newpc (v-to-nat (read-mem (list t t t t) (caar s)))
                (z-flag (flags (car s)))
                (c-flag (flags (car s))))))
  ((enable fm9001-step store-resultp v-alu bv cvzbv)))


(prove-lemma memory-unchanged-fm9001-step (rewrite)
  (implies
   (and
    (quiz-program-functioning-invariants s)
    (equal (length a) 32)
    (not (v-iff a (nat-to-v #xc0000000 32)))
    (not (v-iff a (nat-to-v #xc0000001 32)))
    (not (v-iff a (nat-to-v #xc0000002 32)))
    (not (v-iff a (nat-to-v #xc0000003 32))))
   (equal (read-mem a (cadr (fm9001-step s (list t t t t))))
         (read-mem a (cadr s))))
  ((enable fm9001-step)))

(prove-lemma quiz-program-functioning-invariants-nlistp (rewrite)
  (implies
   (not (listp s))
   (not (quiz-program-functioning-invariants s))))

;; a very silly way to do this: the expand hints open invariant
;; definition just when we need it: the "stepped" invariant, and after
;; elimination.
```

```
(prove-lemma quiz-program-functioning-invariants-preserved-by-fm9001-step (rewrite)
  (implies
   (quiz-program-functioning-invariants s)
   (quiz-program-functioning-invariants (fm9001-step s (list t t t t))))
  ((disable quiz-program-functioning-invariants)
   (expand (quiz-program-functioning-invariants (fm9001-step s (list t t t t)))
           (quiz-program-functioning-invariants (cons z x))
           (quiz-program-functioning-invariants (cons x z))
           (quiz-program-functioning-invariants (cons (cons v w) z))
           (quiz-program-functioning-invariants (cons z (cons v w))))))

(prove-lemma listp-extract-from-assignment (rewrite)
  (equal (listp (extract-from-assignment l i)) (listp l)))

(prove-lemma fm9001-step-r3 (rewrite)
  (implies
   (quiz-program-functioning-invariants s)
   (equal
    (read-mem (list t t f f) (caar (fm9001-step s (list t t t t))))
    (if (equal (v-to-nat (read-mem (list t t t t) (caar s))) #x40000024)
        (v-buf (read-mem (list f f f f) (caar s)))
      (read-mem (list t t f f) (caar s)))))
  ((enable fm9001-step v-alu bv store-resultp)))

(prove-lemma fm9001-step-r1 (rewrite)
  (implies
   (quiz-program-functioning-invariants s)
   (equal
    (read-mem (list t f f f) (caar (fm9001-step s (list t t t t))))
    (if (equal (v-to-nat (read-mem (list t t t t) (caar s))) #x4000000b)
        (v-buf (read-mem (nat-to-v #xc0000000 32) (cadr s)))
      (read-mem (list t f f f) (caar s)))))
  ((enable fm9001-step v-alu bv store-resultp)))

(prove-lemma fm9001-step-r2 (rewrite)
  (implies
   (quiz-program-functioning-invariants s)
   (equal
    (read-mem (list f t f f) (caar (fm9001-step s (list t t t t))))
    (if (equal (v-to-nat (read-mem (list t t t t) (caar s))) #x4000000c)
        (v-buf (read-mem (nat-to-v #xc0000002 32) (cadr s)))
      (read-mem (list f t f f) (caar s)))))
  ((enable fm9001-step v-alu bv store-resultp)))

(prove-lemma fm9001-step-r0 (rewrite)
  (implies
   (quiz-program-functioning-invariants s)
   (equal
    (v-to-nat (read-mem (list f f f f) (caar (fm9001-step s (list t t t t)))))
    (newtemp
     (v-to-nat (read-mem (list t t t t) (caar s)))
     (v-to-nat (read-mem (list f f f f) (caar s)))
     (z-flag (flags (car s))))))
  ((enable fm9001-step v-alu bv store-resultp)))

(prove-lemma v-nzerop-v-xor (rewrite)
  (implies
   (and
    (not (v-nzerop (v-xor a y)))
    (equal (length a) (length y)))
   (equal (v-to-nat y) (v-to-nat a)))
  ((enable v-to-nat v-xor v-nzerop)))
```

```
(prove-lemma v-nzerop-v-xor-constant (rewrite)
  (implies
   (and
    (equal (v-to-nat x) n)
    (equal (length x) (length y)))
   (equal (v-nzerop (v-xor y x)) (not (equal (v-to-nat y) n))))
  ((enable v-to-nat v-xor v-nzerop)))

(prove-lemma fm9001-step-z-flag (rewrite)
  (implies
   (quiz-program-functioning-invariants s)
   (iff
    (caadar (fm9001-step s (list t t t t)))
    (newz (v-to-nat (read-mem (list t t t t) (caar s)))
          (caadar s)
          (v-to-nat (read-mem (list t t f f) (caar s))))))
  ((enable fm9001-step v-alu bv store-resultp update-flags zb)))

(prove-lemma quiz-program-operation-fm9001-step-z-flag (rewrite)
  (implies
   (quiz-program-functioning-invariants s)
  (equal
   (quiz-program-operation pc state b1 b2 s1-list s2-list l1 l2 temp
                           (caadar (fm9001-step s (list t t t t)))
                           c)
   (quiz-program-operation pc state b1 b2 s1-list s2-list l1 l2 temp
                           (newz (v-to-nat (read-mem (list t t t t) (caar s)))
                                 (caadar s)
                                 (v-to-nat (read-mem (list t t f f) (caar s))))
                           c)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable fm9001-step-z-flag b-buf)
   (use (quiz-program-operation-b-buf1
         (s state) (z (caadar (fm9001-step s (list t t t t))))))))

(prove-lemma equal-1-remainder-v-to-nat-2 (rewrite)
  (equal
   (equal (remainder (v-to-nat x) 2) 1)
   (and (listp x) (car x)))
  ((enable v-to-nat remainder)))

(prove-lemma fm9001-step-c-flag (rewrite)
  (implies
   (quiz-program-functioning-invariants s)
   (iff
    (cadddadar (fm9001-step s (list t t t t)))
    (newc (v-to-nat (read-mem (list t t t t) (caar s)))
          (cadddadar s)
          (v-to-nat (read-mem (list t f f f) (caar s)))
          (v-to-nat (read-mem (list f t f f) (caar s))))))
  ((enable fm9001-step v-alu bv store-resultp update-flags zb c)))

(prove-lemma quiz-program-operation-fm9001-step-c-flag (rewrite)
  (implies
   (quiz-program-functioning-invariants s)
  (equal
   (quiz-program-operation pc state b1 b2 s1-list s2-list l1 l2 temp z
                           (cadddadar (fm9001-step s (list t t t t))))
   (quiz-program-operation pc state b1 b2 s1-list s2-list l1 l2 temp z
                           (newc (v-to-nat (read-mem (list t t t t) (caar s)))
```

```
                              (cadddadar s)
                              (v-to-nat (read-mem (list t f f f) (caar s)))
                              (v-to-nat (read-mem (list f t f f) (caar s)))))))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable fm9001-step-c-flag b-buf)
   (use (quiz-program-operation-b-buf2
           (s state) (c (cadddadar (fm9001-step s (list t t t t))))))))


(prove-lemma fm9001-step-l1 (rewrite)
  (implies
   (quiz-program-functioning-invariants s)
   (equal
    (read-mem
     (list t f f f f f f f f f f f f f f f f f f f f f f f f f f f f f t t)
     (cadr (fm9001-step s (list t t t t))))
    (if (equal (v-to-nat (read-mem (list t t t t) (caar s))) #x40000028)
        (v-buf (read-mem (list f f f f) (caar s)))
      (read-mem
       (list t f f f f f f f f f f f f f f f f f f f f f f f f f f f f f t t)
       (cadr s)))))
  ((enable fm9001-step v-alu bv store-resultp)))

(prove-lemma fm9001-step-l2 (rewrite)
  (implies
   (quiz-program-functioning-invariants s)
   (equal
    (read-mem
     (list t t f f f f f f f f f f f f f f f f f f f f f f f f f f f f t t)
     (cadr (fm9001-step s (list t t t t))))
    (if (equal (v-to-nat (read-mem (list t t t t) (caar s))) #x4000002c)
        (v-buf (read-mem (list f f f f) (caar s)))
      (read-mem
       (list t t f f f f f f f f f f f f f f f f f f f f f f f f f f f f t t)
       (cadr s)))))
  ((enable fm9001-step v-alu bv store-resultp)))

(prove-lemma nthcdr-list-of-cadrs (rewrite)
  (implies
   (lessp n (length l))
   (equal (nthcdr n (list-of-cadrs l)) (list-of-cadrs (nthcdr n l))))
  ((enable nthcdr length)))

(prove-lemma nthcdr-list-of-cars (rewrite)
  (implies
   (lessp n (length l))
   (equal (nthcdr n (list-of-cars l)) (list-of-cars (nthcdr n l))))
  ((enable nthcdr length)))

(prove-lemma nth-list-of-cars (rewrite)
  (equal
   (nth n (list-of-cars l))
   (if (lessp n (length l)) (car (nth n l))  0))
  ((enable nth)
   (induct (nth n l))))

(prove-lemma nth-list-of-cadrs (rewrite)
  (equal
   (nth n (list-of-cadrs l))
   (if (lessp n (length l)) (cadr (nth n l))  0))
  ((enable nth)
```

```
   (induct (nth n l))))

(prove-lemma variable-not-overflowed-means (rewrite)
  (implies
   (variable-not-overflowed-in-historyp x h)
   (equal
    (remainder (cadr (assoc x (nth n h))) (exp 2 32))
    (cadr (assoc x (nth n h)))))
   ((induct (nth n h))))


;; We want to disable extract-from-assignment, but that interferes
;; with some update-state-with-inputs reasoning later on.  So, we add
;; a few bridge lemmas to help out.

(prove-lemma read-mem-update-state-with-inputs-bridge1 (rewrite)
  (implies
   (ramp-mem
    (list f t f f f f f f f f f f f f f f f f f f f f f f f f f f f f t t)
    (cadr s))
   (equal
    (read-mem
     (list f t f f f f f f f f f f f f f f f f f f f f f f f f f f f f t t)
     (cadr
      (update-state-with-inputs
       (extract-from-assignment '((3221225472 signal1) (3221225474 signal2)) a)
       s)))
    (nat-to-v (cadr (assoc 'signal2 a)) 32))))

(prove-lemma read-mem-update-state-with-inputs-bridge2 (rewrite)
  (implies
   (ramp-mem
    (list f f f f f f f f f f f f f f f f f f f f f f f f f f f f f f t t)
    (cadr s))
   (equal
    (read-mem
     (list f f f f f f f f f f f f f f f f f f f f f f f f f f f f f f t t)
     (cadr
      (update-state-with-inputs
       (extract-from-assignment '((3221225472 signal1) (3221225474 signal2)) a)
       s)))
    (nat-to-v (cadr (assoc 'signal1 a)) 32))))

;; subsumed by nthcdr-extract-from-history, but makes things go
(prove-lemma nthcdr-extract-from-history2
             (rewrite)
             (implies (lessp n (length h))
                      (equal (nthcdr n
                                     (extract-from-history out h))
                             (extract-from-history out
                                                   (nthcdr n h))))
             ((enable nthcdr)
              (induct (nthcdr n h))))

(prove-lemma lessp-quiz-i-quiz-t (rewrite)
  (not (lessp (quiz-t pc z c) (quiz-i pc z c)))
  ((enable quiz-t quiz-i)))

(prove-lemma equal-quiz-t-0 (rewrite)
  (not (equal (quiz-t pc z c) 0)))

(prove-lemma equal-quiz-i-0 (rewrite)
```

```
  (not (equal (quiz-i pc z c) 0)))

(prove-lemma variable-not-overflowed-in-historyp-cdr (rewrite)
  (implies
   (variable-not-overflowed-in-historyp x h)
   (variable-not-overflowed-in-historyp x (cdr h))))

(prove-lemma variable-not-overflowed-in-historyp-make-properp (rewrite)
  (equal
   (variable-not-overflowed-in-historyp x (make-properp h))
   (variable-not-overflowed-in-historyp x h)))


(defn fm9001-repeat-quiz-helper-induct (s h mat)
  (if (or
       (read-write-statep s)
       (not (lessp (init-time s mat) (length h)))
       (not (lessp (microcycles-constant s mat) (length h))))
      t
    (fm9001-repeat-quiz-helper-induct
     (fm9001-step
      (update-state-with-inputs
       (nth (sub1 (init-time s mat))
            (extract-from-history '((#xc0000000 signal1) (#xc0000002 signal2)) h))
       s)
      (list t t t t))
     (nthcdr (microcycles-constant s mat) h)
     mat))
  ((lessp (length h))))

(prove-lemma fm9001-repeat-helper-quiz-embedded (rewrite)
  (implies
   (and
    (quiz-program-functioning-invariants s)
    (variable-not-overflowed-in-historyp 'signal1 h)
    (variable-not-overflowed-in-historyp 'signal2 h)
    (equal mat 3))
   (equal
    (fm9001-rt-history-repeat-helper
     s mat
     (extract-from-history '((#xc0000000 signal1) (#xc0000002 signal2)) h)
     '(((15) pc) ((3) s) ((1) b1) ((2) b2) (#xc0000000 signal1) (#xc0000002 signal2)
       (#xc0000001 l1) (#xc0000003 l2)))
    (quiz-program-operation
     (v-to-nat (read-mem (nat-to-v 15 4) (regs (car s))))
     (v-to-nat (read-mem (nat-to-v 3 4) (regs (car s))))
     (v-to-nat (read-mem (nat-to-v 1 4) (regs (car s))))
     (v-to-nat (read-mem (nat-to-v 2 4) (regs (car s))))
     (list-of-cadrs (list-of-cars (extract-from-history '((#xc0000000 signal1)) h)))
     (list-of-cadrs (list-of-cars (extract-from-history '((#xc0000002 signal2)) h)))
     (v-to-nat (read-mem (nat-to-v #xc0000001 32) (cadr s)))
     (v-to-nat (read-mem (nat-to-v #xc0000003 32) (cadr s)))
     (v-to-nat (read-mem (nat-to-v 0 4) (regs (car s))))
     (z-flag (flags (car s)))
     (c-flag (flags (car s))))))
  ((induct (fm9001-repeat-quiz-helper-induct s h mat))
   (disable-theory t)
   (enable-theory ground-zero)
    (enable read-write-statep-from-quiz-program-functioning open-nth init-time-is-quiz-i
            microcycles-constant-is-quiz-t open-nthcdr cdr-extract-from-history
            nth-extract-from-history equal-quiz-i-0 equal-quiz-t-0 lessp-quiz-i-quiz-t
            variable-not-overflowed-in-historyp-make-properp
```

```
            variable-not-overflowed-in-historyp-cdr
            variable-not-overflowed-in-historyp-firstn nth-list-of-cadrs car-cons cdr-cons
            nth-list-of-cars car-extract-from-assignment cdr-list-of-cadrs cdr-list-of-cars
            listp-list-of-cars quiz-program-operation-fm9001-step-c-flag
            quiz-program-operation-fm9001-step-z-flag listp-extract-from-history flags z-flag
            sub1 c-flag not or fm9001-rt-history-repeat-helper quiz-program-operation
            reg-size cdr car listp lessp-0-quiz-i lessp-0-quiz-t
            variable-not-overflowed-in-historyp read-mem-update-state-with-inputs-bridge1
            nat-to-v equal v-to-nat-v-buf lessp-subsumed nthcdr-extract-from-history
            nthcdr-extract-from-history2 equal-length-small listp-firstn
            variable-not-overflowed-in-nthcdr-history variable-not-overflowed-in-cdr-history
            equal-append-x-x quiz-program-functioning-invariants-preserved-by-fm9001-step
            quiz-program-functioning-invariants-preserved-by-update-state-with-inputs
            nth-small car-update-state-with-inputs fm9001-step-r3
            quiz-program-functioning-invariants-means cdr-firstn
            quiz-program-operation-b-buf2 quiz-program-operation-b-buf1
            equal-quiz-program-operation-append pc-stays-in-bounds-fm9001-step
            read-mem-update-state-with-inputs-bridge2 bvp-nat-to-v v-buf-works fm9001-step-r0
            fm9001-step-r1 variable-not-overflowed-means v-to-nat-of-nat-to-v fm9001-step-r2
            read-mem-cadr-update-state-with-inputs-not fm9001-step-l1 fm9001-step-l2
            repeat-with-inputs-quiz-embedded length-list-of-cadrs length-list-of-cars
            quiz-program-functioning-invariants-nlistp length-extract-from-history
            length-firstn1 length-nthcdr firstn-extract-from-history firstn-list-of-cars
            firstn-list-of-cadrs quiz-history-make-signal2-smaller *1*nat-to-v *1*quiz-i
            *1*quiz-t *1*v-iff *1*list-of-cars *1*list-of-cadrs nthcdr-list-of-cars
            nthcdr-list-of-cadrs regs length-cdr-rewrite *1*length member-v-iff list-nat-to-v
            nthcdr-firstn)))

(prove-lemma fm9001-quiz-embedded nil
  (implies
   (and
    (quiz-program-functioning-invariants s)
    (variable-not-overflowed-in-historyp 'signal1 h)
    (variable-not-overflowed-in-historyp 'signal2 h)
    (equal mat 3))
   (equal
    (fm9001-rt-history
     s mat
     (extract-from-history '((#xc0000000 signal1) (#xc0000002 signal2)) h)
     '(((15) pc) ((3) s) ((1) b1) ((2) b2) (#xc0000000 signal1) (#xc0000002 signal2)
       (#xc0000001 l1) (#xc0000003 l2)))
    (quiz-program-operation
     (v-to-nat (read-mem (nat-to-v 15 4) (regs (car s))))
     (v-to-nat (read-mem (nat-to-v 3 4) (regs (car s))))
     (v-to-nat (read-mem (nat-to-v 1 4) (regs (car s))))
     (v-to-nat (read-mem (nat-to-v 2 4) (regs (car s))))
     (list-of-cadrs (list-of-cars (extract-from-history '((#xc0000000 signal1)) h)))
     (list-of-cadrs (list-of-cars (extract-from-history '((#xc0000002 signal2)) h)))
     (v-to-nat (read-mem (nat-to-v #xc0000001 32) (cadr s)))
     (v-to-nat (read-mem (nat-to-v #xc0000003 32) (cadr s)))
     (v-to-nat (read-mem (nat-to-v 0 4) (regs (car s))))
     (z-flag (flags (car s)))
     (c-flag (flags (car s))))))
  ((use (fm9001-repeat-helper-quiz-embedded))
   (disable-theory t)
   (enable-theory ground-zero)
   (enable fm9001-rt-history fm9001-rt-repeat-helper-equiv)))

))
```

E.4 "quiz-correct.events"

```
(proveall "quiz-correct"
'(
#|
 Copyright (C) 1995 by Matthew Wilding and Computational Logic, Inc.
 All Rights Reserved.

 This script is hereby placed in the public domain, and therefore unlimited
 editing and redistribution is permitted.

 NO WARRANTY

 Matthew Wilding and Computational Logic, Inc. PROVIDES ABSOLUTELY NO
 WARRANTY.  THE EVENT SCRIPT IS PROVIDED "AS IS" WITHOUT WARRANTY OF
 ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
 ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE.  THE ENTIRE RISK AS TO THE QUALITY AND
 PERFORMANCE OF THE SCRIPT IS WITH YOU.  SHOULD THE SCRIPT PROVE
 DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR
 CORRECTION.

 IN NO EVENT WILL Matthew Wilding and Computational Logic, Inc. BE
 LIABLE TO YOU FOR ANY DAMAGES, ANY LOST PROFITS, LOST MONIES, OR
 OTHER SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE
 USE OR INABILITY TO USE THIS SCRIPT (INCLUDING BUT NOT LIMITED TO
 LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY
 THIRD PARTIES), EVEN IF YOU HAVE ADVISED US OF THE POSSIBILITY OF
 SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

This file contains the events that lead to the proof of the program
correctness lemma for the quiz-show system

|#

;; We use the library from the proof of the reasonableness lemma
(note-lib "quiz-prog")

(compile-uncompiled-defns "temp")

(defn quiz-program-spec-map (pc)
    (case pc
        (#x4000000b 'read-b1)
        (#x4000000c 'read-b2)
        (#x4000000d 'update-state)
        (#x4000000e 'update-state)
        (#x40000010 'update-state)
        (#x40000011 'update-state)
        (#x40000012 'update-state)
        (#x40000014 'update-state)
        (#x40000015 'update-state)
        (#x40000016 'update-state)
        (#x40000018 'update-state)
        (#x40000019 'update-state)
        (#x4000001b 'update-state)
        (#x4000001c 'update-state)
        (#x4000001d 'update-state)
        (#x4000001f 'update-state)
        (#x40000020 'update-state)
        (#x40000021 'update-state)
        (#x40000023 'update-state)
        (#x40000024 'update-state)
```

```
        (#x40000025 'update-l1)
        (#x40000026 'update-l1)
        (#x40000027 'update-l1)
        (#x40000028 'update-l1)
        (#x40000029 'update-l2)
        (#x4000002a 'update-l2)
        (#x4000002b 'update-l2)
        (#x4000002c 'update-l2)
        (#x4000002d 'read-b1)
        (otherwise 'bad-state)))

(defn quiz-program-spec-map-list (h)
  (if (listp h)
      (cons (quiz-program-spec-map (cadr (assoc 'pc (car h))))
            (quiz-program-spec-map-list (cdr h)))
    nil))

(defn quiz-program-correctness-invariants (pc s b1 b2 temp z c c9)
  (and
   (or (equal s 0) (equal s 1) (equal s 2))
   (or (equal b1 0) (equal b1 1))
   (or (equal b2 0) (equal b2 1))
  (case pc
        (#x4000000b (lessp c9 20))
        (#x4000000c (lessp c9 36))
        (#x4000000d (lessp c9 52))
        (#x4000000e (and (lessp c9 63)
                         (equal z (equal s 2))))
        (#x40000010 (and (lessp c9 79)
                         (not (equal s 2))))
        (#x40000011 (and (lessp c9 89)
                         (not (equal s 2))
                         (equal c (equal b1 1))))
        (#x40000012 (and (lessp c9 99)
                         (not (equal s 2))
                         (equal c (equal b1 1))
                         (equal temp 1)))
        (#x40000014 (and (lessp c9 115)
                         (not (equal s 2))
                         (equal b1 0)))
        (#x40000015 (and (lessp c9 125)
                         (not (equal s 2))
                         (equal b1 0)
                         (equal c (equal b2 1))))
        (#x40000016 (and (lessp c9 135)
                         (not (equal s 2))
                         (equal b1 0)
                         (equal c (equal b2 1))
                         (equal temp 2)))
        (#x40000018 (and (lessp c9 151)
                         (not (equal s 2))
                         (equal b1 0)
                         (equal b2 0)))
        (#x40000019 (and (lessp c9 161)
                         (not (equal s 2))
                         (equal b1 0)
                         (equal b2 0)
                         (equal temp 0)))
        (#x4000001b (and (lessp c9 79)
                         (equal s 2)))
        (#x4000001c (and (lessp c9 89)
                         (equal s 2)
```

```
                      (equal c (equal b2 1))))
        (#x4000001d (and (lessp c9 99)
                      (equal s 2)
                      (equal c (equal b2 1))
                      (equal temp 2)))
        (#x4000001f (and (lessp c9 115)
                      (equal s 2)
                      (equal b2 0)))
        (#x40000020 (and (lessp c9 125)
                      (equal s 2)
                      (equal b2 0)
                      (equal c (equal b1 1))))
        (#x40000021 (and (lessp c9 135)
                      (equal s 2)
                      (equal b2 0)
                      (equal c (equal b1 1))
                      (equal temp 1)))
        (#x40000023 (and (lessp c9 151)
                      (equal s 2)
                      (equal b2 0)
                      (equal b1 0)))
        (#x40000024 (and (lessp c9 177)
                      (equal temp (if (or (equal s 0) (equal s 1))
                                      (if (equal b1 1) 1 (if (equal b2 1) 2 0))
                                    (if (equal b2 1) 2 (if (equal b1 1) 1 0))))))
        (#x40000025 (lessp c9 187))
        (#x40000026 (and (lessp c9 197)
                      (equal temp 1)))
        (#x40000027 (and (lessp c9 208)
                      (equal temp 1)
                      (equal z (equal s 1))))
        (#x40000028 (and (lessp c9 218)
                      (equal temp (if (equal s 1) 0 1))))
        (#x40000029 (lessp c9 234))
        (#x4000002a (and (lessp c9 244)
                      (equal temp 1)))
        (#x4000002b (and (lessp c9 255)
                      (equal temp 1)
                      (equal z (equal s 2))))
        (#x4000002c (and (lessp c9 265)
                      (equal temp (if (equal s 2) 0 1))))
        (#x4000002d (lessp c9 4))
        (otherwise f))))

(prove-lemma quiz-program-correctness-invariants-constant (rewrite)
  (and
   (equal (quiz-program-correctness-invariants #x4000000b s b1 b2 temp z c c9)
          (and
           (or (equal s 0) (equal s 1) (equal s 2))
           (or (equal b1 0) (equal b1 1))
           (or (equal b2 0) (equal b2 1))
           (lessp c9 20)))

   (equal (quiz-program-correctness-invariants #x4000000c s b1 b2 temp z c c9)
          (and
           (or (equal s 0) (equal s 1) (equal s 2))
           (or (equal b1 0) (equal b1 1))
           (or (equal b2 0) (equal b2 1))
           (lessp c9 36)))

   (equal (quiz-program-correctness-invariants #x4000000d s b1 b2 temp z c c9)
          (and
```

```
        (or (equal s 0) (equal s 1) (equal s 2))
        (or (equal b1 0) (equal b1 1))
        (or (equal b2 0) (equal b2 1))
        (lessp c9 52)))

(equal (quiz-program-correctness-invariants #x4000000e s b1 b2 temp z c c9)
       (and
        (or (equal s 0) (equal s 1) (equal s 2))
        (or (equal b1 0) (equal b1 1))
        (or (equal b2 0) (equal b2 1))
        (and (lessp c9 63) (equal z (equal s 2)))))

(equal (quiz-program-correctness-invariants #x40000010 s b1 b2 temp z c c9)
       (and
        (or (equal s 0) (equal s 1) (equal s 2))
        (or (equal b1 0) (equal b1 1))
        (or (equal b2 0) (equal b2 1))
        (and (lessp c9 79) (not (equal s 2)))))

(equal (quiz-program-correctness-invariants #x40000011 s b1 b2 temp z c c9)
       (and
        (or (equal s 0) (equal s 1) (equal s 2))
        (or (equal b1 0) (equal b1 1))
        (or (equal b2 0) (equal b2 1))
        (and (lessp c9 89) (not (equal s 2))
             (equal c (equal b1 1)))))

(equal (quiz-program-correctness-invariants #x40000012 s b1 b2 temp z c c9)
       (and
        (or (equal s 0) (equal s 1) (equal s 2))
        (or (equal b1 0) (equal b1 1))
        (or (equal b2 0) (equal b2 1))
        (and (lessp c9 99) (not (equal s 2))
             (equal c (equal b1 1)) (equal temp 1))))

(equal (quiz-program-correctness-invariants #x40000014 s b1 b2 temp z c c9)
       (and
        (or (equal s 0) (equal s 1) (equal s 2))
        (or (equal b1 0) (equal b1 1))
        (or (equal b2 0) (equal b2 1))
        (and (lessp c9 115) (not (equal s 2))
             (equal b1 0))))

(equal (quiz-program-correctness-invariants #x40000015 s b1 b2 temp z c c9)
       (and
        (or (equal s 0) (equal s 1) (equal s 2))
        (or (equal b1 0) (equal b1 1))
        (or (equal b2 0) (equal b2 1))
        (and (lessp c9 125) (not (equal s 2))
             (equal b1 0) (equal c (equal b2 1)))))

(equal (quiz-program-correctness-invariants #x40000016 s b1 b2 temp z c c9)
       (and
        (or (equal s 0) (equal s 1) (equal s 2))
        (or (equal b1 0) (equal b1 1))
        (or (equal b2 0) (equal b2 1))
        (and (lessp c9 135) (not (equal s 2))
             (equal b1 0) (equal c (equal b2 1)) (equal temp 2))))

(equal (quiz-program-correctness-invariants #x40000018 s b1 b2 temp z c c9)
       (and
        (or (equal s 0) (equal s 1) (equal s 2))
```

```
        (or (equal b1 0) (equal b1 1))
        (or (equal b2 0) (equal b2 1))
        (and (lessp c9 151) (not (equal s 2)) (equal b1 0)
             (equal b2 0))))

(equal (quiz-program-correctness-invariants #x40000019 s b1 b2 temp z c c9)
       (and
        (or (equal s 0) (equal s 1) (equal s 2))
        (or (equal b1 0) (equal b1 1))
        (or (equal b2 0) (equal b2 1))
        (and (lessp c9 161) (not (equal s 2))
             (equal b1 0) (equal b2 0) (equal temp 0))))

(equal (quiz-program-correctness-invariants #x4000001b s b1 b2 temp z c c9)
       (and
        (or (equal s 0) (equal s 1) (equal s 2))
        (or (equal b1 0) (equal b1 1))
        (or (equal b2 0) (equal b2 1))
        (and (lessp c9 79) (equal s 2))))

(equal (quiz-program-correctness-invariants #x4000001c s b1 b2 temp z c c9)
       (and
        (or (equal s 0) (equal s 1) (equal s 2))
        (or (equal b1 0) (equal b1 1))
        (or (equal b2 0) (equal b2 1))
        (and (lessp c9 89) (equal s 2) (equal c (equal b2 1)))))

(equal (quiz-program-correctness-invariants #x4000001d s b1 b2 temp z c c9)
       (and
        (or (equal s 0) (equal s 1) (equal s 2))
        (or (equal b1 0) (equal b1 1))
        (or (equal b2 0) (equal b2 1))
        (and (lessp c9 99) (equal s 2)
             (equal c (equal b2 1)) (equal temp 2))))

(equal (quiz-program-correctness-invariants #x4000001f s b1 b2 temp z c c9)
       (and
        (or (equal s 0) (equal s 1) (equal s 2))
        (or (equal b1 0) (equal b1 1))
        (or (equal b2 0) (equal b2 1))
        (and (lessp c9 115) (equal s 2) (equal b2 0))))

(equal (quiz-program-correctness-invariants #x40000020 s b1 b2 temp z c c9)
       (and
        (or (equal s 0) (equal s 1) (equal s 2))
        (or (equal b1 0) (equal b1 1))
        (or (equal b2 0) (equal b2 1))
        (and (lessp c9 125) (equal s 2)
             (equal b2 0) (equal c (equal b1 1)))))

(equal (quiz-program-correctness-invariants #x40000021 s b1 b2 temp z c c9)
       (and
        (or (equal s 0) (equal s 1) (equal s 2))
        (or (equal b1 0) (equal b1 1))
        (or (equal b2 0) (equal b2 1))
        (and (lessp c9 135) (equal s 2) (equal b2 0)
             (equal c (equal b1 1)) (equal temp 1))))

(equal (quiz-program-correctness-invariants #x40000023 s b1 b2 temp z c c9)
       (and
        (or (equal s 0) (equal s 1) (equal s 2))
        (or (equal b1 0) (equal b1 1))
```

```
        (or (equal b2 0) (equal b2 1))
        (and (lessp c9 151) (equal s 2)
             (equal b2 0) (equal b1 0))))

(equal (quiz-program-correctness-invariants #x40000024 s b1 b2 temp z c c9)
       (and
        (or (equal s 0) (equal s 1) (equal s 2))
        (or (equal b1 0) (equal b1 1))
        (or (equal b2 0) (equal b2 1))
        (and (lessp c9 177)
             (equal temp (if (or (equal s 0) (equal s 1))
                             (if (equal b1 1) 1 (if (equal b2 1) 2 0))
                           (if (equal b2 1) 2 (if (equal b1 1) 1 0)))))))

(equal (quiz-program-correctness-invariants #x40000025 s b1 b2 temp z c c9)
       (and
        (or (equal s 0) (equal s 1) (equal s 2))
        (or (equal b1 0) (equal b1 1))
        (or (equal b2 0) (equal b2 1))
        (lessp c9 187)))

(equal (quiz-program-correctness-invariants #x40000026 s b1 b2 temp z c c9)
       (and
        (or (equal s 0) (equal s 1) (equal s 2))
        (or (equal b1 0) (equal b1 1))
        (or (equal b2 0) (equal b2 1))
        (and (lessp c9 197) (equal temp 1))))

(equal (quiz-program-correctness-invariants #x40000027 s b1 b2 temp z c c9)
       (and
        (or (equal s 0) (equal s 1) (equal s 2))
        (or (equal b1 0) (equal b1 1))
        (or (equal b2 0) (equal b2 1))
        (and (lessp c9 208) (equal temp 1) (equal z (equal s 1)))))

(equal (quiz-program-correctness-invariants #x40000028 s b1 b2 temp z c c9)
       (and
        (or (equal s 0) (equal s 1) (equal s 2))
        (or (equal b1 0) (equal b1 1))
        (or (equal b2 0) (equal b2 1))
        (and (lessp c9 218) (equal temp (if (equal s 1) 0 1)))))

(equal (quiz-program-correctness-invariants #x40000029 s b1 b2 temp z c c9)
       (and
        (or (equal s 0) (equal s 1) (equal s 2))
        (or (equal b1 0) (equal b1 1))
        (or (equal b2 0) (equal b2 1))
        (lessp c9 234)))

(equal (quiz-program-correctness-invariants #x4000002a s b1 b2 temp z c c9)
       (and
        (or (equal s 0) (equal s 1) (equal s 2))
        (or (equal b1 0) (equal b1 1))
        (or (equal b2 0) (equal b2 1))
        (and (lessp c9 244) (equal temp 1))))

(equal (quiz-program-correctness-invariants #x4000002b s b1 b2 temp z c c9)
       (and
        (or (equal s 0) (equal s 1) (equal s 2))
        (or (equal b1 0) (equal b1 1))
        (or (equal b2 0) (equal b2 1))
        (and (lessp c9 255) (equal temp 1) (equal z (equal s 2)))))
```

```
   (equal (quiz-program-correctness-invariants #x4000002c s b1 b2 temp z c c9)
          (and
           (or (equal s 0) (equal s 1) (equal s 2))
           (or (equal b1 0) (equal b1 1))
           (or (equal b2 0) (equal b2 1))
           (and (lessp c9 265) (equal temp (if (equal s 2) 0 1)))))

   (equal (quiz-program-correctness-invariants #x4000002d s b1 b2 temp z c c9)
          (and
           (or (equal s 0) (equal s 1) (equal s 2))
           (or (equal b1 0) (equal b1 1))
           (or (equal b2 0) (equal b2 1))
           (lessp c9 4)))))

(prove-lemma listp-quiz-history (rewrite)
  (equal (listp (quiz-history pc s b1 b2 s1-list s2-list l1 l2)) (listp s1-list)))

(prove-lemma quiz-program-spec-map-list-quiz-history (rewrite)
  (equal
   (quiz-program-spec-map-list (quiz-history pc s b1 b2 s1-list s2-list l1 l2))
   (repeat (length s1-list) (quiz-program-spec-map pc)))
  ((enable repeat length)
   (disable length-cdr-rewrite)))

(prove-lemma quiz-program-spec-map-list-append (rewrite)
  (equal
   (quiz-program-spec-map-list (append a b))
   (append (quiz-program-spec-map-list a) (quiz-program-spec-map-list b))))

;; calculate counter when traversing states in list
(defn counters-of-quiz (counters list)
  (if (listp list)
      (if (listp (cdr list))
          (counters-of-quiz
           (update-counter-alist
            counters
            (if (and (equal (car list) 'update-l2) (equal (cadr list) 'read-b1))
                '((c 9))
              nil))
           (cdr list))
        counters)
    counters))

(prove-lemma assoc-counters-of-quiz (rewrite)
  (implies
   (not (equal x 0))
   (iff (assoc x (counters-of-quiz c l)) (assoc x c))))

(prove-lemma quiz-program-spec-embedded-path-helper-append-nlistp (rewrite)
  (implies
   (not (listp y))
   (equal (quiz-program-spec-embedded-path-helper (append x y) c m)
          (quiz-program-spec-embedded-path-helper x c m)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable quiz-program-spec-embedded-path-helper listp-append append car-append)
   (induct (quiz-program-spec-embedded-path-helper x c m))))

(prove-lemma quiz-program-spec-embedded-path-helper-append-nlistp-bridge (rewrite)
  (implies
   (not (listp y))
```

```
   (equal (quiz-program-spec-embedded-path-helper (cons a (append b y)) c m)
          (quiz-program-spec-embedded-path-helper (cons a b) c m)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable append)
   (use (quiz-program-spec-embedded-path-helper-append-nlistp (x (cons a b))))))

(prove-lemma lessp-length-maplist (rewrite)
  (implies
   (lessp (length maplist) (length h))
   (not (quiz-program-spec-embedded-path-helper h c maplist))))

(prove-lemma equal-sub1-length-0 (rewrite)
  (equal (equal (sub1 (length x)) 0) (nlistp (cdr x))))

(prove-lemma quiz-program-spec-embedded-cons (rewrite)
  (implies
   (or (equal a 'read-b1) (equal a 'read-b2) (equal a 'update-state)
       (equal a 'update-l1) (equal a 'update-l2))
  (equal (quiz-program-spec-embedded-path-helper h c (cons a p))
  (if (listp h)
      (let ((oldb1 (cadr (assoc 'b1 (car h))))
            (oldb2 (cadr (assoc 'b2 (car h))))
            (olds (cadr (assoc 's (car h))))
            (oldl1 (cadr (assoc 'l1 (car h))))
            (oldl2 (cadr (assoc 'l2 (car h))))
            (c9 (cadr (assoc '(c 9) c))))
        (if (nlistp (cdr h))
            (case a
              (read-b1 (lessp c9 50))
              (read-b2 (lessp c9 100))
              (update-state (lessp c9 300))
              (update-l1 (lessp c9 350))
              (update-l2 (lessp c9 400))
              (otherwise f))
          (let ((b1 (cadr (assoc 'b1 (cadr h))))
                (b2 (cadr (assoc 'b2 (cadr h))))
                (s (cadr (assoc 's (cadr h))))
                (l1 (cadr (assoc 'l1 (cadr h))))
                (l2 (cadr (assoc 'l2 (cadr h)))))
            (if
             (case a
               (read-b1
                (if (equal (car p) 'read-b1)
                 (and (lessp c9 49) (equal b1 oldb1) (equal b2 oldb2)
                      (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                 (if (equal (car p) 'read-b2)
                     (and
                      (lessp c9 50)
                      (equal b1 (cadr (assoc 'signal1 (car h)))) (equal b2 oldb2)
                      (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                   f)))
               (read-b2
                (if (equal (car p) 'read-b2)
                    (and (lessp c9 99) (equal b1 oldb1) (equal b2 oldb2)
                         (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                  (if (equal (car p) 'update-state)
                      (and
                       (lessp c9 100)
                       (equal b1 oldb1) (equal b2 (cadr (assoc 'signal2 (car h))))
                       (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                    f)))
```

```
              (update-state
               (if (equal (car p) 'update-state)
                   (and (lessp c9 299) (equal b1 oldb1) (equal b2 oldb2)
                     (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                 (if (equal (car p) 'update-l1)
                     (and
                      (lessp c9 300) (equal b1 oldb1) (equal b2 oldb2)
                      (equal s (if (or (equal olds 0) (equal olds 1))
                                   (if (equal b1 1) 1 (if (equal b2 1) 2 0))
                                 (if (equal b2 1) 2 (if (equal b1 1) 1 0))))
                      (equal l1 oldl1) (equal l2 oldl2))
                   f)))
              (update-l1
               (if (equal (car p) 'update-l1)
                   (and (lessp c9 349) (equal b1 oldb1) (equal b2 oldb2)
                        (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                 (if (equal (car p) 'update-l2)
                     (and
                      (lessp c9 350)
                      (equal b1 oldb1) (equal b2 oldb2) (equal s olds)
                      (equal l1 (if (equal s 1) 0 1)) (equal l2 oldl2))
                   f)))
               (update-l2
                (if (equal (car p) 'update-l2)
                    (and  (lessp c9 399) (equal b1 oldb1) (equal b2 oldb2)
                     (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                  (if (equal (car p) 'read-b1)
                      (and
                       (lessp c9 400)
                       (equal b1 oldb1) (equal b2 oldb2) (equal s olds)
                       (equal l1 oldl1) (equal l2 (if (equal s 2) 0 1)))
                    f)))
               (otherwise f))
            (quiz-program-spec-embedded-path-helper
             (cdr h)
             (update-counter-alist
              c
              (if (and (equal a 'update-l2) (equal (car p) 'read-b1))
                  '((c 9)) nil))
             p)
            f))))
    t)))
  ((disable-theory t)
   (enable length quiz-program-spec-embedded-path-helper)
   (enable-theory ground-zero)))

(prove-lemma quiz-program-spec-embedded-cons-special (rewrite)
  (implies
   (and (not (equal a 'read-b1)) (not (equal a 'read-b2))
        (not (equal a 'update-state)) (not (equal a 'update-l1)))
  (equal (quiz-program-spec-embedded-path-helper h c (cons a p))
  (if (listp h)
      (let ((oldb1 (cadr (assoc 'b1 (car h))))
            (oldb2 (cadr (assoc 'b2 (car h))))
            (olds (cadr (assoc 's (car h))))
            (oldl1 (cadr (assoc 'l1 (car h))))
            (oldl2 (cadr (assoc 'l2 (car h))))
            (c9 (cadr (assoc '(c 9) c))))
        (if (nlistp (cdr h))
            (case a
              (read-b1 (lessp c9 50))
              (read-b2 (lessp c9 100))
```

```
    (update-state (lessp c9 300))
    (update-l1 (lessp c9 350))
    (update-l2 (lessp c9 400))
    (otherwise f))
(let ((b1 (cadr (assoc 'b1 (cadr h))))
      (b2 (cadr (assoc 'b2 (cadr h))))
      (s (cadr (assoc 's (cadr h))))
      (l1 (cadr (assoc 'l1 (cadr h))))
      (l2 (cadr (assoc 'l2 (cadr h)))))
  (if
   (case a
     (read-b1
      (if (equal (car p) 'read-b1)
       (and (lessp c9 49) (equal b1 oldb1) (equal b2 oldb2)
            (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
       (if (equal (car p) 'read-b2)
           (and
            (lessp c9 50)
            (equal b1 (cadr (assoc 'signal1 (car h)))) (equal b2 oldb2)
            (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
         f)))
     (read-b2
      (if (equal (car p) 'read-b2)
          (and (lessp c9 99) (equal b1 oldb1) (equal b2 oldb2)
               (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
        (if (equal (car p) 'update-state)
            (and
             (lessp c9 100)
             (equal b1 oldb1) (equal b2 (cadr (assoc 'signal2 (car h))))
             (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
          f)))
     (update-state
      (if (equal (car p) 'update-state)
          (and (lessp c9 299) (equal b1 oldb1) (equal b2 oldb2)
            (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
        (if (equal (car p) 'update-l1)
            (and
             (lessp c9 300) (equal b1 oldb1) (equal b2 oldb2)
             (equal s (if (or (equal olds 0) (equal olds 1))
                          (if (equal b1 1) 1 (if (equal b2 1) 2 0))
                        (if (equal b2 1) 2 (if (equal b1 1) 1 0))))
             (equal l1 oldl1) (equal l2 oldl2))
          f)))
     (update-l1
      (if (equal (car p) 'update-l1)
          (and (lessp c9 349) (equal b1 oldb1) (equal b2 oldb2)
               (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
        (if (equal (car p) 'update-l2)
            (and
             (lessp c9 350)
             (equal b1 oldb1) (equal b2 oldb2) (equal s olds)
             (equal l1 (if (equal s 1) 0 1)) (equal l2 oldl2))
          f)))
      (update-l2
       (if (equal (car p) 'update-l2)
           (and  (lessp c9 399) (equal b1 oldb1) (equal b2 oldb2)
            (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
         (if (equal (car p) 'read-b1)
             (and
              (lessp c9 400)
              (equal b1 oldb1) (equal b2 oldb2) (equal s olds)
              (equal l1 oldl1) (equal l2 (if (equal s 2) 0 1)))
```

```
                    f)))
                 (otherwise f))
              (quiz-program-spec-embedded-path-helper
               (cdr h)
               (update-counter-alist
                c
                (if (and (equal a 'update-l2) (equal (car p) 'read-b1))
                    '((c 9)) nil))
               p)
              f))))
    t)))
   ((disable-theory t)
    (enable length quiz-program-spec-embedded-path-helper)
    (enable-theory ground-zero)))

(prove-lemma quiz-program-spec-embedded-open (rewrite)
  (implies
   (equal (cadr p) y)
   (equal (quiz-program-spec-embedded-path-helper h c p)
          (if (listp h)
      (let ((oldb1 (cadr (assoc 'b1 (car h))))
            (oldb2 (cadr (assoc 'b2 (car h))))
            (olds (cadr (assoc 's (car h))))
            (oldl1 (cadr (assoc 'l1 (car h))))
            (oldl2 (cadr (assoc 'l2 (car h))))
            (c9 (cadr (assoc '(c 9) c))))
        (if (nlistp (cdr h))
            (case (car p)
              (read-b1 (lessp c9 50))
              (read-b2 (lessp c9 100))
              (update-state (lessp c9 300))
              (update-l1 (lessp c9 350))
              (update-l2 (lessp c9 400))
              (otherwise f))
          (let ((b1 (cadr (assoc 'b1 (cadr h))))
                (b2 (cadr (assoc 'b2 (cadr h))))
                (s (cadr (assoc 's (cadr h))))
                (l1 (cadr (assoc 'l1 (cadr h))))
                (l2 (cadr (assoc 'l2 (cadr h)))))
            (if
             (case (car p)
               (read-b1
                (if (equal (cadr p) 'read-b1)
                 (and (lessp c9 49) (equal b1 oldb1) (equal b2 oldb2)
                      (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                 (if (equal (cadr p) 'read-b2)
                     (and
                      (lessp c9 50)
                      (equal b1 (cadr (assoc 'signal1 (car h)))) (equal b2 oldb2)
                      (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                   f)))
               (read-b2
                (if (equal (cadr p) 'read-b2)
                    (and (lessp c9 99) (equal b1 oldb1) (equal b2 oldb2)
                         (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                  (if (equal (cadr p) 'update-state)
                      (and
                       (lessp c9 100)
                       (equal b1 oldb1) (equal b2 (cadr (assoc 'signal2 (car h))))
                       (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                    f)))
               (update-state
```

```
                (if (equal (cadr p) 'update-state)
                    (and (lessp c9 299) (equal b1 oldb1) (equal b2 oldb2)
                      (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                  (if (equal (cadr p) 'update-l1)
                      (and
                       (lessp c9 300) (equal b1 oldb1) (equal b2 oldb2)
                       (equal s (if (or (equal olds 0) (equal olds 1))
                                    (if (equal b1 1) 1 (if (equal b2 1) 2 0))
                                  (if (equal b2 1) 2 (if (equal b1 1) 1 0))))
                       (equal l1 oldl1) (equal l2 oldl2))
                    f)))
               (update-l1
                (if (equal (cadr p) 'update-l1)
                    (and (lessp c9 349) (equal b1 oldb1) (equal b2 oldb2)
                         (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                  (if (equal (cadr p) 'update-l2)
                      (and
                       (lessp c9 350)
                       (equal b1 oldb1) (equal b2 oldb2) (equal s olds)
                       (equal l1 (if (equal s 1) 0 1)) (equal l2 oldl2))
                    f)))
                (update-l2
                 (if (equal (cadr p) 'update-l2)
                     (and  (lessp c9 399) (equal b1 oldb1) (equal b2 oldb2)
                      (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                   (if (equal (cadr p) 'read-b1)
                       (and
                        (lessp c9 400)
                        (equal b1 oldb1) (equal b2 oldb2) (equal s olds)
                        (equal l1 oldl1) (equal l2 (if (equal s 2) 0 1)))
                     f)))
                (otherwise f))
             (quiz-program-spec-embedded-path-helper
              (cdr h)
              (update-counter-alist
               c
               (if (and (equal (car p) 'update-l2) (equal (cadr p) 'read-b1))
                   '((c 9)) nil))
              (cdr p))
             f))))
    t)))
  ((disable-theory t)
   (expand (quiz-program-spec-embedded-path-helper h c p))
   (enable length)
   (enable-theory ground-zero)))

(prove-lemma quiz-program-spec-embedded-path-helper-append-simple (rewrite)
  (implies
   (not (quiz-program-spec-embedded-path-helper h1 counters maplist))
   (not (quiz-program-spec-embedded-path-helper (append h1 h2) counters maplist)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable quiz-program-spec-embedded-path-helper listp-append append car-append)
   (induct (quiz-program-spec-embedded-path-helper h1 counters maplist))))

(prove-lemma quiz-program-spec-embedded-path-helper-append-simple-bridge (rewrite)
  (implies
   (not (quiz-program-spec-embedded-path-helper h1 counters maplist))
   (not (quiz-program-spec-embedded-path-helper
         (cons (car h1) (append (cdr h1) h2)) counters maplist)))
  ((disable-theory t)
   (enable-theory ground-zero)
```

```
   (enable quiz-program-spec-embedded-path-helper)
   (use (quiz-program-spec-embedded-path-helper-append-simple))))

(prove-lemma quiz-program-spec-embedded-path-helper-badmap (rewrite)
  (implies
   (and
    (not (equal a 'read-b1))
    (not (equal a 'read-b2))
    (not (equal a 'update-state))
    (not (equal a 'update-l1))
    (not (equal a 'update-l2)))
   (equal (quiz-program-spec-embedded-path-helper h c (cons a m))
          (nlistp h))))

(prove-lemma quiz-program-spec-embedded-path-helper-badmap2 (rewrite)
  (implies
   (and
    (not (equal a 'read-b1))
    (not (equal a 'read-b2))
    (not (equal a 'update-state))
    (not (equal a 'update-l1))
    (not (equal b 'update-l2))
    (not (equal b 'read-b1)))
   (equal (quiz-program-spec-embedded-path-helper h c (cons a (cons b m)))
          (or (nlistp h)
          (and
           (equal a 'update-l2)
           (lessp (cadr (assoc '(c 9) c)) 400)
           (nlistp (cdr h)))))))

(prove-lemma quiz-program-spec-embedded-path-helper-open-when-change (rewrite)
  (implies
   (or
    (not (equal (cadr (assoc 'l1 (car h))) (cadr (assoc 'l1 (cadr h)))))
    (not (equal (cadr (assoc 'l2 (car h))) (cadr (assoc 'l2 (cadr h)))))
    (not (equal (cadr (assoc 'b1 (car h))) (cadr (assoc 'b1 (cadr h)))))
    (not (equal (cadr (assoc 'b2 (car h))) (cadr (assoc 'b2 (cadr h)))))
    (not (equal (cadr (assoc 's (car h))) (cadr (assoc 's (cadr h))))))
   (equal (quiz-program-spec-embedded-path-helper h c p)
          (if (listp h)
      (let ((oldb1 (cadr (assoc 'b1 (car h))))
            (oldb2 (cadr (assoc 'b2 (car h))))
            (olds (cadr (assoc 's (car h))))
            (oldl1 (cadr (assoc 'l1 (car h))))
            (oldl2 (cadr (assoc 'l2 (car h))))
            (c9 (cadr (assoc '(c 9) c))))
        (if (nlistp (cdr h))
            (case (car p)
              (read-b1 (lessp c9 50))
              (read-b2 (lessp c9 100))
              (update-state (lessp c9 300))
              (update-l1 (lessp c9 350))
              (update-l2 (lessp c9 400))
              (otherwise f))
          (let ((b1 (cadr (assoc 'b1 (cadr h))))
                (b2 (cadr (assoc 'b2 (cadr h))))
                (s (cadr (assoc 's (cadr h))))
                (l1 (cadr (assoc 'l1 (cadr h))))
                (l2 (cadr (assoc 'l2 (cadr h)))))
            (if
             (case (car p)
               (read-b1
```

```
                (if (equal (cadr p) 'read-b1)
                 (and (lessp c9 49) (equal b1 oldb1) (equal b2 oldb2)
                      (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                 (if (equal (cadr p) 'read-b2)
                     (and
                      (lessp c9 50)
                      (equal b1 (cadr (assoc 'signal1 (car h)))) (equal b2 oldb2)
                      (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                   f)))
               (read-b2
                (if (equal (cadr p) 'read-b2)
                    (and (lessp c9 99) (equal b1 oldb1) (equal b2 oldb2)
                         (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                  (if (equal (cadr p) 'update-state)
                      (and
                       (lessp c9 100)
                       (equal b1 oldb1) (equal b2 (cadr (assoc 'signal2 (car h))))
                       (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                    f)))
               (update-state
                (if (equal (cadr p) 'update-state)
                    (and (lessp c9 299) (equal b1 oldb1) (equal b2 oldb2)
                      (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                  (if (equal (cadr p) 'update-l1)
                      (and
                       (lessp c9 300) (equal b1 oldb1) (equal b2 oldb2)
                       (equal s (if (or (equal olds 0) (equal olds 1))
                                    (if (equal b1 1) 1 (if (equal b2 1) 2 0))
                                  (if (equal b2 1) 2 (if (equal b1 1) 1 0))))
                       (equal l1 oldl1) (equal l2 oldl2))
                    f)))
               (update-l1
                (if (equal (cadr p) 'update-l1)
                    (and (lessp c9 349) (equal b1 oldb1) (equal b2 oldb2)
                         (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                  (if (equal (cadr p) 'update-l2)
                      (and
                       (lessp c9 350)
                       (equal b1 oldb1) (equal b2 oldb2) (equal s olds)
                       (equal l1 (if (equal s 1) 0 1)) (equal l2 oldl2))
                    f)))
                (update-l2
                 (if (equal (cadr p) 'update-l2)
                     (and  (lessp c9 399) (equal b1 oldb1) (equal b2 oldb2)
                      (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
                   (if (equal (cadr p) 'read-b1)
                       (and
                        (lessp c9 400)
                        (equal b1 oldb1) (equal b2 oldb2) (equal s olds)
                        (equal l1 oldl1) (equal l2 (if (equal s 2) 0 1)))
                     f)))
                (otherwise f))
             (quiz-program-spec-embedded-path-helper
              (cdr h)
              (update-counter-alist
               c
               (if (and (equal (car p) 'update-l2) (equal (cadr p) 'read-b1))
                   '((c 9)) nil))
              (cdr p))
             f))))
  t)))
((disable-theory t)
```

```
   (enable length quiz-program-spec-embedded-path-helper)
   (enable-theory ground-zero)))

(prove-lemma quiz-program-spec-embedded-path-helper-append (rewrite)
  (equal
   (quiz-program-spec-embedded-path-helper (append h1 h2) counters maplist)
   (if (nlistp h1)
       (quiz-program-spec-embedded-path-helper h2 counters maplist)
     (if (nlistp h2)
         (quiz-program-spec-embedded-path-helper h1 counters maplist)
       (and
        (quiz-program-spec-embedded-path-helper h1 counters maplist)
        (quiz-program-spec-embedded-path-helper
         (list (last h1) (car h2))
         (counters-of-quiz counters (firstn (length h1) maplist))
         (list (nth (sub1 (length h1)) maplist) (nth (length h1) maplist)))
        (quiz-program-spec-embedded-path-helper
         h2
         (counters-of-quiz counters (firstn (add1 (length h1)) maplist))
         (nthcdr (length h1) maplist))))))
  ((induct (quiz-program-spec-embedded-path-helper h1 counters maplist))
   (disable-theory t)
   (enable-theory ground-zero)
   (enable lessp-subsumed lessp-add1-add1
           quiz-program-spec-embedded-path-helper-append-simple
           quiz-program-spec-embedded-path-helper-recurse counters-of-quiz car-append
           cdr-append append-nlistp nlistp last listp-append nth
           quiz-program-spec-embedded-path-helper-append-simple
           quiz-program-spec-embedded-path-helper-append-simple-bridge
           quiz-program-spec-embedded-open car-firstn
           quiz-program-spec-embedded-path-helper-badmap
           quiz-program-spec-embedded-path-helper-badmap2
           quiz-program-spec-embedded-path-helper-open-when-change listp-firstn length-nthcdr
           equal-sub1-length-0 length-nlistp-cdr firstn-1 nth-small car-cons cdr-cons firstn
           quiz-program-spec-embedded-path-helper-append-nlistp equal-length-small
           quiz-program-spec-embedded-path-helper-append-nlistp-bridge
           assoc-update-counter-alist-casesplit assoc-counters-of-quiz lessp-length-maplist
           length-cons *1*length length-cdr-rewrite quiz-program-spec-embedded-cons append
           firstn-2 nthcdr-0 open-nthcdr and equal-iff)))

(defn update-counter-alist-n (c counters n)
  (if (zerop n)
      c
    (update-counter-alist-n (update-counter-alist c counters) counters (sub1 n))))

(prove-lemma car-quiz-history (rewrite)
  (equal (car (quiz-history pc s b1 b2 s1-list s2-list l1 l2))
         (if (listp s1-list)
             (list
              (list 'pc pc)
              (list 's s)
              (list 'b1 b1)
              (list 'b2 b2)
              (list 'signal1 (car s1-list))
              (list 'signal2 (car s2-list))
              (list 'l1 l1)
              (list 'l2 l2))
           0))
  ((enable quiz-history)))


(prove-lemma car-repeat (rewrite)
```

```
  (equal (car (repeat n e)) (if (zerop n) 0 e)))

(prove-lemma cdr-repeat (rewrite)
  (equal (cdr (repeat n e)) (if (zerop n) 0 (repeat (sub1 n) e))))

(prove-lemma quiz-program-spec-embedded-path-helper-nil (rewrite)
  (quiz-program-spec-embedded-path-helper nil counters maplist))

(defn cdr-cdr-sub1-add1-counter-induction (l1 l2 n counter)
  (if (nlistp l1) t
    (cdr-cdr-sub1-add1-counter-induction
     (cdr l1) (cdr l2) (sub1 n)
     (update-counter-alist counter nil))))

(prove-lemma quiz-program-spec-embedded-path-helper-single (rewrite)
  (implies
   (equal (length h) 1)
   (equal (quiz-program-spec-embedded-path-helper h c p)
          (case (car p)
                (read-b1 (lessp (cadr (assoc '(c 9) c)) 50))
                (read-b2 (lessp (cadr (assoc '(c 9) c)) 100))
                (update-state (lessp (cadr (assoc '(c 9) c)) 300))
                (update-l1 (lessp (cadr (assoc '(c 9) c)) 350))
                (update-l2 (lessp (cadr (assoc '(c 9) c)) 400))
                (otherwise f)))))

(prove-lemma length-repeat (rewrite)
  (equal (length (repeat n e)) (fix n)))

(prove-lemma lessp-add1-plus-add1-2 (rewrite)
  (equal (lessp (add1 x) (plus y (add1 z)))
         (lessp x (plus y z))))

(disable lessp-add1-plus-add1-2)

(prove-lemma lessp-opposite-length-hack (rewrite)
  (implies
   (and
    (not (lessp a b))
    (listp x))
   (equal (lessp b (plus (length x) a)) t)))

(disable lessp-opposite-length-hack)

(prove-lemma quiz-program-spec-embedded-timed-out-simple (rewrite)
  (implies
   (and
    (assoc '(c 9) counters)
    (lessp 399 (plus (sub1 (length h)) (cadr (assoc '(c 9) counters)))))
   (equal (quiz-program-spec-embedded-path-helper h counters (repeat n state))
          (nlistp h)))
  ((induct (cdr-cdr-sub1-add1-counter-induction h l n counters))
   (disable-theory t)
   (enable quiz-program-spec-embedded-path-helper length assoc-update-counter-alist-casesplit
           lessp-add1-plus-add1-2 car-repeat cdr-repeat)
   (enable-theory ground-zero naturals)
   (expand (quiz-program-spec-embedded-path-helper h counters (repeat n state)))))

(prove-lemma repeat-add1 (rewrite)
  (equal (repeat (add1 x) e)  (cons e (repeat x e))))

(prove-lemma quiz-program-spec-embedded-quiz-history (rewrite)
```

```
  (implies
   (assoc '(c 9) counters)
   (equal
   (quiz-program-spec-embedded-path-helper
    (quiz-history pc s b1 b2 s1-list s2-list l1 l2)
    counters
    (repeat n state))
   (and
    (not (lessp n (length s1-list)))
    (or
     (nlistp s1-list)
     (case
      state
      (read-b1
       (not (lessp 50 (plus (length s1-list) (cadr (assoc '(c 9) counters))))))
      (read-b2
       (not (lessp 100 (plus (length s1-list) (cadr (assoc '(c 9) counters))))))
      (update-state
       (not (lessp 300 (plus (length s1-list) (cadr (assoc '(c 9) counters))))))
      (update-l1
       (not (lessp 350 (plus (length s1-list) (cadr (assoc '(c 9) counters))))))
      (update-l2
       (not (lessp 400 (plus (length s1-list) (cadr (assoc '(c 9) counters))))))
      (otherwise f))))))
  ((induct (cdr-cdr-sub1-add1-counter-induction s1-list s2-list n counters))
   (disable-theory t)
   (enable-theory ground-zero)
   (enable quiz-history car-quiz-history quiz-program-spec-embedded-path-helper-single
           quiz-program-spec-embedded-timed-out-simple length-repeat length-quiz-history
           listp-quiz-history quiz-program-spec-embedded-cons repeat-add1 length-cdr-rewrite
           lessp-add1-add1 quiz-program-spec-embedded-path-helper-badmap2
           quiz-program-spec-embedded-path-helper-badmap lessp-opposite-length-hack
           lessp-add1-plus-add1-2 length-nlistp length-cons equal-length-small
           quiz-program-spec-embedded-path-helper-nil lessp-length-maplist *1*length
           car-repeat assoc-cons assoc-update-counter-alist-casesplit)))


(prove-lemma quiz-program-correctness-invariants-badpc (rewrite)
  (implies
   (and
    (not (equal pc #x4000000b))
    (not (equal pc #x4000000c))
    (not (equal pc #x4000000d))
    (not (equal pc #x4000000e))
    (not (equal pc #x40000010))
    (not (equal pc #x40000011))
    (not (equal pc #x40000012))
    (not (equal pc #x40000014))
    (not (equal pc #x40000015))
    (not (equal pc #x40000016))
    (not (equal pc #x40000018))
    (not (equal pc #x40000019))
    (not (equal pc #x4000001b))
    (not (equal pc #x4000001c))
    (not (equal pc #x4000001d))
    (not (equal pc #x4000001f))
    (not (equal pc #x40000020))
    (not (equal pc #x40000021))
    (not (equal pc #x40000023))
    (not (equal pc #x40000024))
    (not (equal pc #x40000025))
    (not (equal pc #x40000026))
```

```
    (not (equal pc #x40000027))
    (not (equal pc #x40000028))
    (not (equal pc #x40000029))
    (not (equal pc #x4000002a))
    (not (equal pc #x4000002b))
    (not (equal pc #x4000002c))
    (not (equal pc #x4000002d)))
   (not (quiz-program-correctness-invariants pc s b1 b2 temp z c c9))))


(prove-lemma equal-assoc-c9-means-same-counters-of-quiz nil
  (implies
   (and
    (assoc '(c 9) c1)
    (assoc '(c 9) c2)
    (equal (cadr (assoc '(c 9) c1)) (cadr (assoc '(c 9) c2))))
   (equal (cadr (assoc '(c 9) (counters-of-quiz c1 l)))
          (cadr (assoc '(c 9) (counters-of-quiz c2 l))))))

(prove-lemma counters-of-quiz-not-assoc-c9 (rewrite)
  (implies
   (not (assoc '(c 9) c))
   (equal (cadr (assoc '(c 9) (counters-of-quiz c l))) 0)))

(prove-lemma counters-of-quiz-update-update-hack1 (rewrite)
  (equal
   (cadr (assoc '(c 9)
                (counters-of-quiz
                 (update-counter-alist (update-counter-alist c nil) '((c 9)))
                 l)))
   (cadr (assoc '(c 9)
                (counters-of-quiz
                 (update-counter-alist c '((c 9)))
                 l))))
  ((use (equal-assoc-c9-means-same-counters-of-quiz
         (c1 (update-counter-alist (update-counter-alist c nil) '((c 9))))
         (c2 (update-counter-alist c '((c 9))))))))

(prove-lemma counters-of-quiz-update-update-hack2 (rewrite)
  (equal
   (cadr (assoc '(c 9)
                (counters-of-quiz
                 (update-counter-alist (update-counter-alist c '((c 9))) '((c 9)))
                 l)))
   (cadr (assoc '(c 9)
                (counters-of-quiz
                 (update-counter-alist c '((c 9)))
                 l))))
  ((use (equal-assoc-c9-means-same-counters-of-quiz
         (c1 (update-counter-alist (update-counter-alist c '((c 9))) '((c 9))))
         (c2 (update-counter-alist c '((c 9))))))))

(prove-lemma reset-means-ignore-previous-counter-updates (rewrite)
  (implies
   (and (listp z)
        (equal (last z) 'update-l2)
        (listp b)
        (equal (car b) 'read-b1))
   (equal
    (cadr (assoc '(c 9) (counters-of-quiz counters (append z b))))
    (cadr (assoc '(c 9) (counters-of-quiz
                         (update-counter-alist counters '((c 9)))
```

```
                      b)))))
  ((induct (counters-of-quiz counters z))))

(prove-lemma cadr-assoc-counters-of-quiz-append (rewrite)
  (equal
   (cadr (assoc '(c 9) (counters-of-quiz counters (append x y))))
   (if (listp x)
       (if (listp y)
           (if (and (equal (last x) 'update-l2) (equal (car y) 'read-b1))
               (cadr (assoc '(c 9) (counters-of-quiz
                                    (update-counter-alist counters '((c 9))) y)))
             (cadr (assoc '(c 9)
                          (counters-of-quiz
                           (update-counter-alist (counters-of-quiz counters x) nil)
                           y))))
         (cadr (assoc '(c 9) (counters-of-quiz counters x))))
     (cadr (assoc '(c 9) (counters-of-quiz counters y)))))
  ((induct (counters-of-quiz counters x))))

(prove-lemma firstn-repeat (rewrite)
  (equal (firstn n1 (repeat n2 e))
         (if (lessp n1 n2) (repeat n1 e) (repeat n2 e)))
  ((enable firstn)))

(prove-lemma nthcdr-repeat (rewrite)
  (equal (nthcdr n1 (repeat n2 e))
         (if (lessp n2 n1) 0 (repeat (difference n2 n1) e)))
  ((enable nthcdr)))

(prove-lemma nth-repeat (rewrite)
  (equal (nth n1 (repeat n2 e)) (if (lessp n1 n2) e 0))
  ((enable nth)))

(prove-lemma repeat-0 (rewrite)
  (equal (repeat 0 e) nil))

(prove-lemma listp-repeat (rewrite)
  (equal (listp (repeat n e)) (not (zerop n))))

(prove-lemma cadr-assoc-counters-of-quiz-repeat-help nil
  (implies
   (equal l (repeat n e))
   (equal (cadr (assoc '(c 9) (counters-of-quiz c (repeat n e))))
          (if (assoc '(c 9) c)
              (if (or (zerop n) (equal n 1))
                  (cadr (assoc '(c 9) c))
                (plus (cadr (assoc '(c 9) c)) (sub1 n)))
            0)))
  ((induct (cdr-cdr-sub1-add1-counter-induction l l2 n c))))

(prove-lemma cadr-assoc-counters-of-quiz-repeat (rewrite)
   (equal (cadr (assoc '(c 9) (counters-of-quiz c (repeat n e))))
          (if (assoc '(c 9) c)
              (if (or (zerop n) (equal n 1))
                  (cadr (assoc '(c 9) c))
                (plus (cadr (assoc '(c 9) c)) (sub1 n)))
            0))
   ((use (cadr-assoc-counters-of-quiz-repeat-help (l (repeat n e))))))


(prove-lemma last-quiz-history (rewrite)
  (equal (last (quiz-history pc s b1 b2 s1-list s2-list l1 l2))
```

```
        (if (listp s1-list)
            (list
             (list 'pc pc)
             (list 's s)
             (list 'b1 b1)
             (list 'b2 b2)
             (list 'signal1 (last s1-list))
             (list 'signal2 (nth (sub1 (length s1-list)) s2-list))
             (list 'l1 l1)
             (list 'l2 l2))
          nil))
  ((enable quiz-history)))

(prove-lemma counters-of-quiz-append-nlistp (rewrite)
  (implies
   (nlistp y)
   (equal (counters-of-quiz c (append x y))
          (counters-of-quiz c x))))

(prove-lemma quiz-program-spec-embedded-nlistp (rewrite)
  (implies
   (nlistp h)
   (quiz-program-spec-embedded-path-helper h c maplist)))

(prove-lemma quiz-program-spec-embedded-path-helper-maplist-smaller (rewrite)
  (implies
   (lessp (length h) (length maplist))
   (equal (quiz-program-spec-embedded-path-helper h c maplist)
          (quiz-program-spec-embedded-path-helper h c (firstn (length h) maplist))))
  ((induct (quiz-program-spec-embedded-path-helper h c maplist))
   (disable-theory t)
   (enable-theory ground-zero)
   (expand (quiz-program-spec-embedded-path-helper h c maplist)
           (quiz-program-spec-embedded-path-helper h c (firstn (length h) maplist)))
   (enable quiz-program-spec-embedded-cons quiz-program-spec-embedded-cons-special
           listp-append append car-append quiz-program-spec-embedded-nlistp
           quiz-program-spec-embedded-path-helper-open-when-change
           quiz-program-spec-embedded-path-helper-badmap equal-length-small
           quiz-program-spec-embedded-path-helper-badmap2 length car-firstn firstn)))

(prove-lemma quiz-program-spec-embedded-maplist-append-nlistp (rewrite)
  (implies
   (nlistp y)
   (equal (quiz-program-spec-embedded-path-helper h c (append x y))
          (quiz-program-spec-embedded-path-helper h c x)))
  ((induct (quiz-program-spec-embedded-path-helper h c x))
   (enable quiz-program-spec-embedded-cons quiz-program-spec-embedded-cons-special
           listp-append append car-append quiz-program-spec-embedded-nlistp
           lessp-length-maplist quiz-program-spec-embedded-path-helper-open-when-change
           quiz-program-spec-embedded-path-helper-badmap equal-length-small
           quiz-program-spec-embedded-path-helper-badmap2 length car-firstn firstn)
   (disable-theory t)
   (enable-theory ground-zero)
   (expand (quiz-program-spec-embedded-path-helper h c x)
           (quiz-program-spec-embedded-path-helper h c (append x y)))))

(prove-lemma quiz-program-spec-embedded-open-2 (rewrite)
   (equal (quiz-program-spec-embedded-path-helper (cons a (cons b nil)) c p)
      (let ((oldb1 (cadr (assoc 'b1 a)))
            (oldb2 (cadr (assoc 'b2 a)))
            (olds (cadr (assoc 's a)))
            (oldl1 (cadr (assoc 'l1 a)))
```

```
  (oldl2 (cadr (assoc 'l2 a)))
  (c9 (cadr (assoc '(c 9) c))))
(let ((b1 (cadr (assoc 'b1 b)))
      (b2 (cadr (assoc 'b2 b)))
      (s (cadr (assoc 's b)))
      (l1 (cadr (assoc 'l1 b)))
      (l2 (cadr (assoc 'l2 b))))
  (and
   (case (car p)
     (read-b1
      (if (equal (cadr p) 'read-b1)
       (and (lessp c9 49) (equal b1 oldb1) (equal b2 oldb2)
            (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
       (if (equal (cadr p) 'read-b2)
           (and
            (lessp c9 50)
            (equal b1 (cadr (assoc 'signal1 a))) (equal b2 oldb2)
            (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
         f)))
     (read-b2
      (if (equal (cadr p) 'read-b2)
          (and (lessp c9 99) (equal b1 oldb1) (equal b2 oldb2)
               (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
        (if (equal (cadr p) 'update-state)
            (and
             (lessp c9 100)
             (equal b1 oldb1) (equal b2 (cadr (assoc 'signal2 a)))
             (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
          f)))
     (update-state
      (if (equal (cadr p) 'update-state)
          (and (lessp c9 299) (equal b1 oldb1) (equal b2 oldb2)
            (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
        (if (equal (cadr p) 'update-l1)
            (and
             (lessp c9 300) (equal b1 oldb1) (equal b2 oldb2)
             (equal s (if (or (equal olds 0) (equal olds 1))
                          (if (equal b1 1) 1 (if (equal b2 1) 2 0))
                        (if (equal b2 1) 2 (if (equal b1 1) 1 0))))
             (equal l1 oldl1) (equal l2 oldl2))
          f)))
     (update-l1
      (if (equal (cadr p) 'update-l1)
          (and (lessp c9 349) (equal b1 oldb1) (equal b2 oldb2)
               (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
        (if (equal (cadr p) 'update-l2)
            (and
             (lessp c9 350)
             (equal b1 oldb1) (equal b2 oldb2) (equal s olds)
             (equal l1 (if (equal s 1) 0 1)) (equal l2 oldl2))
          f)))
      (update-l2
       (if (equal (cadr p) 'update-l2)
           (and  (lessp c9 399) (equal b1 oldb1) (equal b2 oldb2)
            (equal s olds) (equal l1 oldl1) (equal l2 oldl2))
         (if (equal (cadr p) 'read-b1)
             (and
              (lessp c9 400)
              (equal b1 oldb1) (equal b2 oldb2) (equal s olds)
              (equal l1 oldl1) (equal l2 (if (equal s 2) 0 1)))
           f)))
      (otherwise f))
```

```
            (case (cadr p)
                  (read-b1 (or (lessp c9 49) (equal (car p) 'update-l2)))
                  (read-b2 (lessp c9 99))
                  (update-state (lessp c9 299))
                  (update-l1 (lessp c9 349))
                  (update-l2 (lessp c9 399))
                  (otherwise f))))))
  ((disable-theory t)
   (enable length quiz-program-spec-embedded-path-helper assoc-update-counter-alist-casesplit)
   (enable-theory ground-zero)))

(prove-lemma last-repeat (rewrite)
  (equal (last (repeat n e)) (if (zerop n) nil e)))


(prove-lemma update-counter-alist-n-iff (rewrite)
  (implies
   (not (equal x 0))
  (iff (assoc x (update-counter-alist-n c y n))
       (assoc x c))))

(prove-lemma cadr-update-counter-alist-n (rewrite)
  (implies
   (not (equal x 0))
   (equal (cadr (assoc x (update-counter-alist-n c y n)))
          (if (zerop n)
              (cadr (assoc x c))
            (if (member x y)
                0
              (if (assoc x c)
                  (plus n (cadr (assoc x c)))
                0)))))
  ((induct (update-counter-alist-n c y n))))

(defn list-of-zero-and-one (list)
  (if (listp list)
      (and
       (or (equal (car list) 0) (equal (car list) 1))
       (list-of-zero-and-one (cdr list)))
    t))

(prove-lemma list-of-zero-and-one-nthcdr (rewrite)
  (implies
   (list-of-zero-and-one x)
   (list-of-zero-and-one (nthcdr n x)))
  ((induct (nthcdr n x))
   (expand (nthcdr n x))))

(prove-lemma list-of-zero-and-one-means-nth (rewrite)
  (implies
   (list-of-zero-and-one x)
   (equal (equal (nth n x) 1) (not (equal (nth n x) 0))))
  ((enable nth)))

(prove-lemma open-quiz-program-operation (rewrite)
  (and
  (implies
   (not (lessp n (length s1-list)))
  (equal
   (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp z c)
    (let ((init (quiz-i pc z c)) (total (quiz-t pc z c)))
    (if (not (lessp init (length s1-list)))
```

```
        (quiz-history pc s b1 b2 s1-list s2-list l1 l2)
      (let
          ((newpc (newpc pc z c))
           (news (if (equal pc #x40000024) temp s))
           (newb1 (if (equal pc #x4000000b) (nth (sub1 init) s1-list) b1))
           (newb2 (if (equal pc #x4000000c) (nth (sub1 init) s2-list) b2))
           (newl1 (if (equal pc #x40000028) temp l1))
           (newl2 (if (equal pc #x4000002c) temp l2))
           (newtemp (newtemp pc temp z))
           (newz (newz pc z s))
           (newc (newc pc c b1 b2)))
        (if (not (lessp total (length s1-list)))
            (append
             (quiz-history pc s b1 b2 (firstn init s1-list) s2-list l1 l2)
             (quiz-history newpc news newb1 newb2 (nthcdr init s1-list)
                           (nthcdr init s2-list) newl1 newl2))
          (append
           (quiz-history pc s b1 b2 (firstn init s1-list) s2-list l1 l2)
           (append
            (quiz-history newpc news newb1 newb2
                          (firstn (difference total init) (nthcdr init s1-list))
                          (nthcdr init s2-list) newl1 newl2)
            (quiz-program-operation
             newpc news newb1 newb2 (nthcdr total s1-list)
             (nthcdr total s2-list) newl1 newl2 newtemp newz newc)))))))))
  (implies
   (lessp n (length s1-list))
  (equal
   (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp z c)
    (let ((init (quiz-i pc z c)) (total (quiz-t pc z c)))
    (if (not (lessp init (length s1-list)))
        (quiz-history pc s b1 b2 s1-list s2-list l1 l2)
      (let
          ((newpc (newpc pc z c))
           (news (if (equal pc #x40000024) temp s))
           (newb1 (if (equal pc #x4000000b) (nth (sub1 init) s1-list) b1))
           (newb2 (if (equal pc #x4000000c) (nth (sub1 init) s2-list) b2))
           (newl1 (if (equal pc #x40000028) temp l1))
           (newl2 (if (equal pc #x4000002c) temp l2))
           (newtemp (newtemp pc temp z))
           (newz (newz pc z s))
           (newc (newc pc c b1 b2)))
        (if (not (lessp total (length s1-list)))
            (append
             (quiz-history pc s b1 b2 (firstn init s1-list) s2-list l1 l2)
             (quiz-history newpc news newb1 newb2 (nthcdr init s1-list)
                           (nthcdr init s2-list) newl1 newl2))
          (append
           (quiz-history pc s b1 b2 (firstn init s1-list) s2-list l1 l2)
           (append
            (quiz-history newpc news newb1 newb2
                          (firstn (difference total init) (nthcdr init s1-list))
                          (nthcdr init s2-list) newl1 newl2)
            (quiz-program-operation
             newpc news newb1 newb2 (nthcdr total s1-list)
             (nthcdr total s2-list) newl1 newl2 newtemp newz newc))))))))))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable quiz-program-operation)))

(prove-lemma nthcdr-quiz-program-spec-map-list (rewrite)
  (equal (nthcdr n (quiz-program-spec-map-list h))
```

```
        (if (lessp (length h) n)
            0
          (quiz-program-spec-map-list (nthcdr n h))))
  ((enable nthcdr)
   (disable quiz-program-spec-map)
   (induct (nthcdr n h))))

(prove-lemma lessp-difference-hack (rewrite)
  (equal (lessp (difference a b) c)
         (if (lessp a b) (lessp 0 c) (lessp a (plus b c)))))

(prove-lemma nth-quiz-program-map-list (rewrite)
  (equal (nth n (quiz-program-spec-map-list h))
         (if (lessp n (length h))
             (quiz-program-spec-map (cadr (assoc 'pc (nth n h))))
           0))
  ((disable quiz-program-spec-map)
   (induct (nth n h))))

(prove-lemma car-quiz-program-map-list (rewrite)
  (equal (car (quiz-program-spec-map-list h))
         (if (listp h)
             (quiz-program-spec-map (cadr (assoc 'pc (car h))))
           0))
  ((disable quiz-program-spec-map)
   (expand (quiz-program-spec-map-list h))))

(prove-lemma length-quiz-program-operation (rewrite)
  (equal
   (length (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp z c))
   (length s1-list))
  ((induct (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp z c))
   (disable-theory t)
   (enable-theory ground-zero naturals)
   (enable length-append length-quiz-history length-nthcdr length-firstn lessp-quiz-i-quiz-t
           quiz-program-operation)
   (expand (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp z c))))

(prove-lemma listp-quiz-program-operation (rewrite)
  (equal
   (listp (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp z c))
   (listp s1-list))
  ((induct (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp z c))
   (disable-theory t)
   (enable-theory ground-zero naturals)
   (enable listp-append length-quiz-history listp-nthcdr listp-firstn lessp-quiz-i-quiz-t
           quiz-program-operation listp-quiz-history length-nlistp)
   (expand (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp z c))))

(prove-lemma lessp-subsumed2 (rewrite)
  (implies
   (and (lessp n1 x) (not (lessp n1 n2)))
   (equal (lessp x n2) f)))

(prove-lemma length-quiz-program-spec-map-list (rewrite)
  (equal
   (length (quiz-program-spec-map-list l))
   (length l)))

(prove-lemma listp-quiz-program-spec-map-list (rewrite)
  (equal
   (listp (quiz-program-spec-map-list l))
```

```
   (listp l)))

(prove-lemma difference-plus-add1-add1 (rewrite)
  (equal (difference (plus (add1 x) y) (add1 z))
         (difference (plus x y) z)))

(prove-lemma car-quiz-program-operation (rewrite)
  (equal (car (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp z c))
         (if (listp s1-list)
             (list
              (list 'pc pc)
              (list 's s)
              (list 'b1 b1)
              (list 'b2 b2)
              (list 'signal1 (car s1-list))
              (list 'signal2 (car s2-list))
              (list 'l1 l1)
              (list 'l2 l2))
           0))
  ((expand (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp z c))
   (disable-theory t)
   (enable car-append car-quiz-history listp-quiz-history listp-firstn car-firstn
           lessp-quiz-i-quiz-t listp-nthcdr length lessp-0-quiz-i lessp-0-quiz-t)
   (enable-theory ground-zero naturals)))

(prove-lemma update-counter-alist-n-update-counter-alist (rewrite)
  (equal (update-counter-alist-n (update-counter-alist c r) r n)
         (update-counter-alist (update-counter-alist-n c r n) r))
  ((induct (plus n n2))))

(prove-lemma update-counter-alist-n-open (rewrite)
  (and
  (implies
   (lessp 0 n)
   (equal
    (update-counter-alist-n c r n)
    (update-counter-alist (update-counter-alist-n c r (sub1 n)) r)))
  (implies
   (zerop n)
   (equal (update-counter-alist-n c r n) c))))




(defn sub1-counter-induction (n c)
  (if (zerop n) t
    (sub1-counter-induction (sub1 n) (update-counter-alist c nil))))

(prove-lemma counters-of-quiz-repeat (rewrite)
  (implies
   (lessp 0 n)
   (equal
    (counters-of-quiz c (repeat n e))
    (if (equal n 1) c
    (update-counter-alist-n c nil (sub1 n)))))
  ((induct (sub1-counter-induction n c))))

(prove-lemma counters-of-quiz-repeat-bridge (rewrite)
  (implies
   (lessp 0 n)
   (equal
    (counters-of-quiz c (cons e (repeat n e)))
```

```
    (if (zerop n) c (update-counter-alist-n c nil n))))
  ((induct (sub1-counter-induction n c))))

(prove-lemma counters-of-quiz-append (rewrite)
  (equal
   (counters-of-quiz c (append x y))
   (if (listp x)
       (if (listp y)
           (counters-of-quiz
            (update-counter-alist
             (counters-of-quiz c x)
             (if (and (equal (last x) 'update-l2) (equal (car y) 'read-b1))
                 '((c 9)) nil))
            y)
         (counters-of-quiz c x))
     (counters-of-quiz c y))))

(prove-lemma counters-of-quiz-small (rewrite)
  (equal
   (counters-of-quiz c (cons a nil))
   c))

(prove-lemma counters-of-quiz-small-2 (rewrite)
  (implies
   (equal (length l) 2)
   (equal
    (counters-of-quiz c l)
    (if (and (equal (car l) 'update-l2) (equal (cadr l) 'read-b1))
        (update-counter-alist c '((c 9)))
      (update-counter-alist c nil)))))

(prove-lemma assoc-counter-alist-backchain (rewrite)
  (and
   (implies
    (and
     (not (equal k 0))
     (not (member k l))
     (assoc k c))
    (equal (cadr (assoc k (update-counter-alist c l)))
           (add1 (cadr (assoc k c)))))
   (implies
    (member k l)
    (equal (cadr (assoc k (update-counter-alist c l))) 0))))

(prove-lemma repeat-1 (rewrite)
  (equal (repeat 1 e) (list e)))

(prove-lemma nthcdr-1 (rewrite)
  (equal (nthcdr 1 l) (cdr l)))

(prove-lemma lessp-1-length (rewrite)
  (equal (lessp 1 (length x)) (listp (cdr x))))

(prove-lemma lessp-length-1 (rewrite)
  (equal (lessp (length x) 1) (nlistp x)))

(prove-lemma program-embedded-satisfies-behavior-embedded-open (rewrite)
  (implies
   (and
    (quiz-program-correctness-invariants pc s b1 b2 temp z c (cadr (assoc '(c 9) counters)))
    (assoc '(c 9) counters)
    (list-of-zero-and-one s1-list)
```

```
  (list-of-zero-and-one s2-list))
 (equal
  (quiz-program-spec-embedded-path-helper
   (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp z c)
   counters
   (quiz-program-spec-map-list
    (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp z c)))
  (let ((init (quiz-i pc z c)) (total (quiz-t pc z c)))
    (if (not (lessp init (length s1-list)))
        t
      (let
          ((newpc (newpc pc z c))
           (news (if (equal pc #x40000024) temp s))
           (newb1 (if (equal pc #x4000000b) (nth (sub1 init) s1-list) b1))
           (newb2 (if (equal pc #x4000000c) (nth (sub1 init) s2-list) b2))
           (newl1 (if (equal pc #x40000028) temp l1))
           (newl2 (if (equal pc #x4000002c) temp l2))
           (newtemp (newtemp pc temp z))
           (newz (newz pc z s))
           (newc (newc pc c b1 b2))
           (newcounters  (if (equal pc #x4000002c)
                             (update-counter-alist-n
                              (update-counter-alist
                               (update-counter-alist-n counters nil (sub1 init))
                               '((c 9)))
                              nil (difference total init))
                           (update-counter-alist-n counters nil total))))
        (quiz-program-spec-embedded-path-helper
         (quiz-program-operation
          newpc news newb1 newb2 (nthcdr total s1-list)
          (nthcdr total s2-list) newl1 newl2 newtemp newz newc)
         newcounters
         (quiz-program-spec-map-list
          (quiz-program-operation
           newpc news newb1 newb2 (nthcdr total s1-list)
           (nthcdr total s2-list) newl1 newl2 newtemp newz newc))))))))
((disable-theory t)
 (enable-theory ground-zero addition)
 (disable plus-add1-arg2 plus-add1-arg1 difference-add1-arg2)
 (enable open-quiz-program-operation quiz-program-spec-embedded-path-helper-single
         equal-sub1-0 lessp-length-1 firstn-0 firstn-1
         update-counter-alist-n-update-counter-alist lessp-sub1-x-x equal-length-small
         nth-quiz-program-map-list counters-of-quiz-append counters-of-quiz-repeat
         counters-of-quiz-repeat-bridge assoc-update-counter-alist-iff
         quiz-program-spec-embedded-nlistp nthcdr-quiz-program-spec-map-list
         lessp-difference-hack car-quiz-program-operation difference-add1-add1
         difference-plus-add1-add1 length-quiz-program-spec-map-list
         listp-quiz-program-spec-map-list counters-of-quiz-small counters-of-quiz-small-2
         lessp-plus-add1-add1 lessp-subsumed lessp-subsumed2 *1*nth *1*member length-nlistp
         length-quiz-history length-quiz-program-operation listp-quiz-program-operation
         length-firstn length-append length-cons repeat-0 nthcdr-1
         list-of-zero-and-one-means-nth list-of-zero-and-one-nthcdr update-counter-alist-n
         last-firstn car-append cdr-append car-nthcdr nth-small nth-append-better
         nth-repeat nth-append append-nlistp quiz-program-spec-embedded-open-2
         quiz-program-spec-embedded-path-helper-maplist-smaller
         quiz-program-spec-embedded-maplist-append-nlistp assoc-counter-alist-backchain
         update-counter-alist-n-iff update-counter-alist-n-open car-quiz-program-map-list
         newpc newc newz newtemp newpc car-quiz-history last-quiz-history firstn-append
         nthcdr-append listp-repeat last-repeat car-repeat cdr-repeat length-repeat
         firstn-repeat nthcdr-repeat length-nthcdr quiz-program-spec-map-list-quiz-history
         quiz-program-spec-map quiz-i quiz-t quiz-program-operation-b-buf2
         quiz-program-operation-b-buf1 quiz-program-correctness-invariants
```

```
          quiz-program-correctness-invariants-badpc quiz-program-spec-embedded-quiz-history
          quiz-program-spec-map-list-append quiz-program-spec-embedded-path-helper-append
          listp-firstn listp-nthcdr listp-append listp-quiz-history)))

;; Check if the invariants hold on the initial quiz loop instruction.
(prove-lemma quiz-correctness-of-initial-state nil
  (let ((s (fm9001-step
    (fm9001-step
     (fm9001-step
      (fm9001-step
       (fm9001-step
        (fm9001-step
         (fm9001-step
          (quiz-initial-state)
          (nat-to-v 15 4))
         (nat-to-v 15 4))
        (nat-to-v 15 4))
       (nat-to-v 15 4))
      (nat-to-v 15 4))
     (nat-to-v 15 4))
    (nat-to-v 15 4))))
    (quiz-program-correctness-invariants
     (v-to-nat (read-mem (nat-to-v 15 4) (regs (car s))))
     (v-to-nat (read-mem (nat-to-v 3 4) (regs (car s))))
     (v-to-nat (read-mem (nat-to-v 1 4) (regs (car s))))
     (v-to-nat (read-mem (nat-to-v 2 4) (regs (car s))))
     (v-to-nat (read-mem (nat-to-v 0 4) (regs (car s))))
     (z-flag (flags (car s)))
     (c-flag (flags (car s)))
     0)))

(prove-lemma quiz-correctness-invariant-preserved (rewrite)
  (implies
   (and
    (quiz-program-correctness-invariants pc s b1 b2 temp z c (cadr (assoc '(c 9) counters)))
    (assoc '(c 9) counters)
    (equal newcounters (if (equal pc #x4000002c)
                           (update-counter-alist-n
                            (update-counter-alist
                             (update-counter-alist-n counters nil (sub1 (quiz-i pc z c)))
                             '((c 9)))
                            nil (difference (quiz-t pc z c) (quiz-i pc z c)))
                         (update-counter-alist-n counters nil (quiz-t pc z c))))
    (equal news (if (equal pc #x40000024) temp s))
    (or (equal pc #x4000000b) (equal newb1 b1))
    (or (equal pc #x4000000c) (equal newb2 b2))
    (or (equal newb1 0) (equal newb1 1))
    (or (equal newb2 0) (equal newb2 1)))
   (quiz-program-correctness-invariants
    (newpc pc z c)
    news
    newb1
    newb2
    (newtemp pc temp z)
    (newz pc z s)
    (newc pc c b1 b2)
    (cadr (assoc '(c 9) newcounters))))
  ((disable-theory t)
   (enable-theory ground-zero)
   (expand (quiz-program-correctness-invariants
              pc s b1 b2 temp z c (cadr (assoc '(c 9) counters))))
   (disable quiz-program-correctness-invariants)
```

```
   (enable quiz-t quiz-program-correctness-invariants-constant newpc newtemp newz newc
           assoc-update-counter-alist-iff quiz-i update-counter-alist-n-open b-buf
           assoc-counter-alist-backchain)))


(prove-lemma invariant-button-normalize (rewrite)
  (implies
   (quiz-program-correctness-invariants pc s b1 b2 temp z c counters)
   (and
    (equal (equal b1 1) (not (equal b1 0)))
    (equal (equal b2 1) (not (equal b2 0)))))
  ((expand (quiz-program-correctness-invariants pc s b1 b2 temp z c counters))
   (disable-theory t)
   (enable-theory ground-zero)))

(prove-lemma update-counter-alist-n-iff-monotonic (rewrite)
  (implies
   (and
    (not (equal x 0))
    (not (assoc x (update-counter-alist-n c y n))))
   (not (assoc x c))))

(prove-lemma update-counter-alist-n-iff-monotonic2 (rewrite)
  (implies
   (and
    (not (equal x 0))
    (not (assoc x (update-counter-alist-n
                   (update-counter-alist
                    (update-counter-alist-n c r1 n1)
                    r2)
                   r3 n3))))
   (not (assoc x c))))

(disable update-counter-alist-n-iff-monotonic)
(disable update-counter-alist-n-iff-monotonic2)

(defn program-embedded-satisfies-induct (pc s b1 b2 s1-list s2-list l1 l2 temp z c counters)
  (let ((init (quiz-i pc z c)) (total (quiz-t pc z c)))
      (let
          ((newpc (newpc pc z c))
           (news (if (equal pc #x40000024) temp s))
           (newb1 (if (equal pc #x4000000b) (nth (sub1 init) s1-list) b1))
           (newb2 (if (equal pc #x4000000c) (nth (sub1 init) s2-list) b2))
           (newl1 (if (equal pc #x40000028) temp l1))
           (newl2 (if (equal pc #x4000002c) temp l2))
           (newtemp (newtemp pc temp z))
           (newz (newz pc z s))
           (newc (newc pc c b1 b2))
           (newcounters  (if (equal pc #x4000002c)
                               (update-counter-alist-n
                                (update-counter-alist
                                 (update-counter-alist-n counters nil (sub1 init))
                                 '((c 9)))
                                nil (difference total init))
                             (update-counter-alist-n counters nil total))))
        (if (not (lessp total (length s1-list)))
            t
            (program-embedded-satisfies-induct
             newpc news newb1 newb2 (nthcdr total s1-list)
             (nthcdr total s2-list) newl1 newl2 newtemp newz newc newcounters))))
  ((lessp (length s1-list))))
```

```
(prove-lemma program-embedded-satisfies-behavior-embedded (rewrite)
  (implies
   (and
    (quiz-program-correctness-invariants pc s b1 b2 temp z c (cadr (assoc '(c 9) counters)))
    (assoc '(c 9) counters)
    (list-of-zero-and-one s1-list)
    (list-of-zero-and-one s2-list))
   (quiz-program-spec-embedded-path-helper
    (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp z c)
    counters
    (quiz-program-spec-map-list
     (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp z c))))
   ((induct (program-embedded-satisfies-induct
              pc s b1 b2 s1-list s2-list l1 l2 temp z c counters))
    (disable-theory t)
    (enable-theory ground-zero naturals)
    (enable program-embedded-satisfies-behavior-embedded-open list-of-zero-and-one-nthcdr
            listp-quiz-program-operation quiz-correctness-invariant-preserved
            update-counter-alist-n-iff-monotonic update-counter-alist-n-iff-monotonic2
            invariant-button-normalize list-of-zero-and-one-means-nth
            quiz-program-spec-embedded-nlistp listp-nthcdr)))


(prove-lemma satisfiesp-with-path-assoc-same-reverse (rewrite)
     (implies (and (all-cars-litatoms trans)
                   (all-transitions-litatoms b))
              (equal (satisfiesp-with-path-helper pred trans h counters b path)
                     (satisfiesp-with-path-assoc-helper pred trans h counters b path)))
     ((use (satisfiesp-with-path-assoc-same-helper (trans1 trans) (trans2 trans)))))

(disable SATISFIESP-WITH-PATH-ASSOC-SAME-reverse)

(prove-lemma satisfiesp-with-path-helper-quiz-program-operation nil
  (implies
   (and
    (quiz-program-correctness-invariants pc s b1 b2 temp z c (cadr (assoc '(c 9) counters)))
    (assoc '(c 9) counters)
    (list-of-zero-and-one s1-list)
    (list-of-zero-and-one s2-list)
    (equal bs (cadr (quiz-program-spec)))
    (equal p (quiz-program-spec-map-list
              (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp z c)))
    (no-litatom-cars counters)
    (all-cars-litatoms (state-trans-list (assoc (car p) bs)))
    (all-transitions-litatoms (cadr (quiz-program-spec))))
   (satisfiesp-with-path-helper
    (state-pred (assoc (car p) bs))
    (state-trans-list (assoc (car p) bs))
    (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp z c)
    counters
    (cadr (quiz-program-spec))
    p))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable program-embedded-satisfies-behavior-embedded
           satisfiesp-with-path-assoc-same-reverse quiz-program-spec-s-w-p-a-embedded)))


(prove-lemma satisfiesp-with-path-helper-means-satisfiesp-helper-better (rewrite)
  (implies
   (and
    (satisfiesp-with-path-helper p tr h c b path)
```

```
    (equal i (car path)))
   (satisfiesp-helper i p tr h c b))
  ((disable-theory t)
   (enable-theory ground-zero)
   (expand (satisfiesp-eff-helper (car path) p tr h c b))
   (enable state-name state-pred state-trans-list trans-to trans-resets trans-pred
           satisfiesp-helper satisfiesp-eff-helper satisfiesp-eff-helper-equiv
           satisfiesp-with-path-helper car-assoc)))

(prove-lemma quiz-program-operation-nlistp (rewrite)
  (implies
   (not (listp s1-list))
   (equal
    (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp z c)
    nil)))

(prove-lemma satisfiesp-quiz-program-operation nil
  (implies
   (and
    (quiz-program-correctness-invariants pc s b1 b2 temp z c 0)
    (equal pc #x4000000b)
    (list-of-zero-and-one s1-list)
    (list-of-zero-and-one s2-list))
   (satisfiesp
    (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp z c)
    (quiz-program-spec)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (use (satisfiesp-with-path-helper-quiz-program-operation
         (counters (make-list-alist
                    (counters-of-state-list (behavior-state-list (quiz-program-spec)))))
         (p (quiz-program-spec-map-list
             (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp z c)))
         (bs (cadr (quiz-program-spec)))))
   (enable *1*quiz-program-spec *1*behavior-initial *1*state-name
           quiz-program-operation-nlistp satisfiesp-nlistp satisfiesp
           satisfiesp-with-path-helper-means-satisfiesp-helper-better
           car-quiz-program-map-list listp-quiz-program-operation car-quiz-program-operation
           *1*state-trans-list *1*state-pred *1*quiz-program-spec-map *1*all-cars-litatoms
           *1*all-transitions-litatoms *1*no-litatom-cars *1*behavior-state-list
           *1*make-list-alist *1*counters-of-state-list)))

))
```

E.5 "quiz-final.events"

```
(proveall "quiz-final"
'(
#|
 Copyright (C) 1995 by Matthew Wilding and Computational Logic, Inc.
 All Rights Reserved.

 This script is hereby placed in the public domain, and therefore unlimited
 editing and redistribution is permitted.

 NO WARRANTY

 Matthew Wilding and Computational Logic, Inc. PROVIDES ABSOLUTELY NO
```

```
This file contains the events that lead to the proof of the final
quiz-show correctness theorem.

|#

;We load a library containing the quiz-show lemmas
(note-lib "quiz-correct")

(compile-uncompiled-defns "temp")

(defn initialized-quiz-state ()
  (fm9001-step
   (fm9001-step
    (fm9001-step
     (fm9001-step
      (fm9001-step
       (fm9001-step
        (fm9001-step
         (quiz-initial-state)
         (nat-to-v 15 4))
        (nat-to-v 15 4))
       (nat-to-v 15 4))
      (nat-to-v 15 4))
     (nat-to-v 15 4))
    (nat-to-v 15 4))
   (nat-to-v 15 4)))

(prove-lemma variable-not-overflowed-from-list-of-zero-and-one (rewrite)
  (implies
   (list-of-zero-and-one
       (list-of-cadrs (list-of-cars (extract-from-history (list (list a x)) h))))
   (variable-not-overflowed-in-historyp x h)))

(defn good-initial-fm9001-quiz-statep (s)
    (let
      ((mem (cadr s))
       (regs (caar s))
       (pc (v-to-nat (read-mem (nat-to-v 15 4) (caar s)))))

      (and
       ;; The quiz program is loaded

       (equal (read-mem (nat-to-v #x4000000b 32) mem) (nth 11 (quiz-prog)))
       (equal (read-mem (nat-to-v #x4000000c 32) mem) (nth 12 (quiz-prog)))
       (equal (read-mem (nat-to-v #x4000000d 32) mem) (nth 13 (quiz-prog)))
       (equal (read-mem (nat-to-v #x4000000e 32) mem) (nth 14 (quiz-prog)))
```

```
(equal (read-mem (nat-to-v #x4000000f 32) mem) (nth 15 (quiz-prog)))
(equal (read-mem (nat-to-v #x40000010 32) mem) (nth 16 (quiz-prog)))
(equal (read-mem (nat-to-v #x40000011 32) mem) (nth 17 (quiz-prog)))
(equal (read-mem (nat-to-v #x40000012 32) mem) (nth 18 (quiz-prog)))
(equal (read-mem (nat-to-v #x40000013 32) mem) (nth 19 (quiz-prog)))
(equal (read-mem (nat-to-v #x40000014 32) mem) (nth 20 (quiz-prog)))
(equal (read-mem (nat-to-v #x40000015 32) mem) (nth 21 (quiz-prog)))
(equal (read-mem (nat-to-v #x40000016 32) mem) (nth 22 (quiz-prog)))
(equal (read-mem (nat-to-v #x40000017 32) mem) (nth 23 (quiz-prog)))
(equal (read-mem (nat-to-v #x40000018 32) mem) (nth 24 (quiz-prog)))
(equal (read-mem (nat-to-v #x40000019 32) mem) (nth 25 (quiz-prog)))
(equal (read-mem (nat-to-v #x4000001a 32) mem) (nth 26 (quiz-prog)))
(equal (read-mem (nat-to-v #x4000001b 32) mem) (nth 27 (quiz-prog)))
(equal (read-mem (nat-to-v #x4000001c 32) mem) (nth 28 (quiz-prog)))
(equal (read-mem (nat-to-v #x4000001d 32) mem) (nth 29 (quiz-prog)))
(equal (read-mem (nat-to-v #x4000001e 32) mem) (nth 30 (quiz-prog)))
(equal (read-mem (nat-to-v #x4000001f 32) mem) (nth 31 (quiz-prog)))
(equal (read-mem (nat-to-v #x40000020 32) mem) (nth 32 (quiz-prog)))
(equal (read-mem (nat-to-v #x40000021 32) mem) (nth 33 (quiz-prog)))
(equal (read-mem (nat-to-v #x40000022 32) mem) (nth 34 (quiz-prog)))
(equal (read-mem (nat-to-v #x40000023 32) mem) (nth 35 (quiz-prog)))
(equal (read-mem (nat-to-v #x40000024 32) mem) (nth 36 (quiz-prog)))
(equal (read-mem (nat-to-v #x40000025 32) mem) (nth 37 (quiz-prog)))
(equal (read-mem (nat-to-v #x40000026 32) mem) (nth 38 (quiz-prog)))
(equal (read-mem (nat-to-v #x40000027 32) mem) (nth 39 (quiz-prog)))
(equal (read-mem (nat-to-v #x40000028 32) mem) (nth 40 (quiz-prog)))
(equal (read-mem (nat-to-v #x40000029 32) mem) (nth 41 (quiz-prog)))
(equal (read-mem (nat-to-v #x4000002a 32) mem) (nth 42 (quiz-prog)))
(equal (read-mem (nat-to-v #x4000002b 32) mem) (nth 43 (quiz-prog)))
(equal (read-mem (nat-to-v #x4000002c 32) mem) (nth 44 (quiz-prog)))
(equal (read-mem (nat-to-v #x4000002d 32) mem) (nth 45 (quiz-prog)))
(equal (read-mem (nat-to-v #x4000002e 32) mem) (nth 46 (quiz-prog)))

;; The memory-mapped input locations are 32-bit RAM
(equal (length (read-mem (nat-to-v #xc0000000 32) mem)) 32)
(ramp-mem (nat-to-v #xc0000000 32) mem)
(equal (length (read-mem (nat-to-v #xc0000002 32) mem)) 32)
(ramp-mem (nat-to-v #xc0000002 32) mem)

;; The memory-mapped output locations are RAM
(ramp-mem (nat-to-v #xc0000001 32) mem)
(ramp-mem (nat-to-v #xc0000003 32) mem)

;; The registers we use are 32-bit RAM
(ramp-mem (nat-to-v 0 4) regs)
(ramp-mem (nat-to-v 1 4) regs)
(ramp-mem (nat-to-v 2 4) regs)
(ramp-mem (nat-to-v 3 4) regs)
(ramp-mem (nat-to-v 15 4) regs)
(equal (length (read-mem (nat-to-v 0 4) regs)) 32)
(equal (length (read-mem (nat-to-v 1 4) regs)) 32)
(equal (length (read-mem (nat-to-v 2 4) regs)) 32)
(equal (length (read-mem (nat-to-v 3 4) regs)) 32)
(equal (length (read-mem (nat-to-v 15 4) regs)) 32)

;; Some registers have memory-mapped I/O device addresses
(equal (read-mem (nat-to-v 4 4) regs) (nat-to-v #xc0000000 32))
(equal (read-mem (nat-to-v 5 4) regs) (nat-to-v #xc0000002 32))
(equal (read-mem (nat-to-v 6 4) regs) (nat-to-v #xc0000001 32))
(equal (read-mem (nat-to-v 7 4) regs) (nat-to-v #xc0000003 32))

;; pc is at beginning of loop
```

```
       (equal (v-to-nat (read-mem (nat-to-v 15 4) regs)) #x4000000b)

       ;; registers are initialed to 0
       (equal (v-to-nat (read-mem (nat-to-v 3 4) regs)) 0)
       (equal (v-to-nat (read-mem (nat-to-v 1 4) regs)) 0)
       (equal (v-to-nat (read-mem (nat-to-v 2 4) regs)) 0)

       ;; lights are off
       (equal (v-to-nat (read-mem (nat-to-v #xc0000001 32) (cadr s))) 1)
       (equal (v-to-nat (read-mem (nat-to-v #xc0000003 32) (cadr s))) 1))))

(prove-lemma fm9001-rt-history-satisfies-program-spec nil
  (implies
   (and
    (good-initial-fm9001-quiz-statep s)
    (list-of-zero-and-one
     (list-of-cadrs
      (list-of-cars
       (extract-from-history '((#xc0000000 signal1)) h))))
    (list-of-zero-and-one
     (list-of-cadrs
      (list-of-cars
       (extract-from-history '((#xc0000002 signal2)) h)))))
   (satisfiesp
    (fm9001-rt-history-repeat-helper
     s 3
     (extract-from-history '((#xc0000000 signal1) (#xc0000002 signal2)) h)
     '(((15) pc) ((3) s) ((1) b1) ((2) b2) (#xc0000000 signal1) (#xc0000002 signal2)
       (#xc0000001 l1) (#xc0000003 l2)))
    (quiz-program-spec)))
  ((use (satisfiesp-quiz-program-operation
         (pc (v-to-nat (read-mem (nat-to-v 15 4) (regs (car s)))))
         (s (v-to-nat (read-mem (nat-to-v 3 4) (regs (car s)))) )
         (b1 (v-to-nat (read-mem (nat-to-v 1 4) (regs (car s)))) )
         (b2 (v-to-nat (read-mem (nat-to-v 2 4) (regs (car s)))) )
         (s1-list
          (list-of-cadrs (list-of-cars (extract-from-history '((#xc0000000 signal1)) h))))
         (s2-list
          (list-of-cadrs (list-of-cars (extract-from-history '((#xc0000002 signal2)) h))))
         (l1 (v-to-nat (read-mem (nat-to-v #xc0000001 32) (cadr s))))
         (l2 (v-to-nat (read-mem (nat-to-v #xc0000003 32) (cadr s))))
         (temp (v-to-nat (read-mem (nat-to-v 0 4) (regs (car s)))))
         (z (z-flag (flags (car s))))
         (c (c-flag (flags (car s))))))
   (disable-theory t)
   (enable *1*nth nth-small good-initial-fm9001-quiz-statep
           quiz-program-correctness-invariants quiz-program-functioning-invariants
           fm9001-repeat-helper-quiz-embedded
           variable-not-overflowed-from-list-of-zero-and-one
           *1*quiz-program-functioning-invariants *1*quiz-program-correctness-invariants
           *1*quiz-prog regs *1*v-to-nat *1*read-mem *1*nat-to-v *1*regs *1*flags *1*z-flag
           *1*c-flag)
   (enable-theory ground-zero)))

(prove-lemma satisfiesp-means-state-pred-satisfied (rewrite)
  (implies
   (and
    (listp h)
    (satisfiesp-helper name pred tr h c bs))
   (my-eval pred (append c (car h)) nil)))

(prove-lemma satisfiesp-helper-nlistp (rewrite)
```

```
  (implies
   (nlistp tr)
   (equal (satisfiesp-helper name pred tr h c bs)
          (or (nlistp h)
              (and (listp h) (nlistp (cdr h))
                   (and (assoc name bs)
                        (my-eval pred
                                 (append c (car h))
                                 nil)))))))

(prove-lemma satisfiedp-helper-bad-trans-name (rewrite)
  (implies
   (and
    (listp tr)
    (not (assoc (caar tr) bs))
    (not (assoc 0 bs)))
   (equal (satisfiesp-helper name pred tr h c bs)
          (satisfiesp-helper name pred (cdr tr) h c bs))))


(DEFN STATE-PREDS-SATISFIEDP
 (NAME PRED TRANS H COUNTERS B-STATES)
 (IF
  (LISTP H)
  (IF
   (LISTP (CDR H))
   (IF
    (LISTP TRANS)
    (AND
     (ASSOC NAME B-STATES)
     (MY-EVAL PRED
              (APPEND COUNTERS (CAR H))
              NIL)
     (IF
      (STATE-PREDS-SATISFIEDP (STATE-NAME (ASSOC (TRANS-TO (CAR TRANS))
                                                 B-STATES))
                              (STATE-PRED (ASSOC (TRANS-TO (CAR TRANS))
                                                 B-STATES))
                              (STATE-TRANS-LIST (ASSOC (TRANS-TO (CAR TRANS))
                                                       B-STATES))
                              (CDR H)
                              (UPDATE-COUNTER-ALIST COUNTERS
                                                    (TRANS-RESETS (CAR TRANS)))
                              B-STATES)
      T
      (STATE-PREDS-SATISFIEDP NAME PRED
                              (CDR TRANS)
                              H COUNTERS B-STATES)))
    F)
   (AND (ASSOC NAME B-STATES)
        (MY-EVAL PRED
                 (APPEND COUNTERS (CAR H))
                 NIL)))
  T)
 ((ORD-LESSP (CONS (ADD1 (LENGTH H))
                   (LENGTH TRANS)))))

(prove-lemma satisfiesp-helper-button1-means-list-of-zero-and-one (rewrite)
  (implies
   (and
    (satisfiesp-helper name pred tr h c bs)
    (equal bs (cadr (button1-spec)))
```

```
    (assoc name bs)
    (equal pred (state-pred (assoc name bs)))
    (not (assoc 'signal1 c))
    (assoc '(c 1) c))
   (list-of-zero-and-one
    (list-of-cadrs
     (list-of-cars
      (extract-from-history (list (list x 'signal1)) h)))))
  ((induct (state-preds-satisfiedp name pred tr h c bs))
   (use (satisfiesp-means-state-pred-satisfied))
   (disable-theory t)
   (expand (satisfiesp-helper name pred tr h c bs))
   (enable-theory ground-zero)
   (enable button1-spec state-trans-list trans-to state-name state-pred
           state-preds-satisfiedp extract-from-history list-of-cars list-of-cadrs
           satisfiesp-helper-nlistp satisfiedp-helper-bad-trans-name list-of-zero-and-one
           assoc-append extract-from-assignment my-eval assoc-append
           assoc-update-counter-alist-iff trans-resets)))

(prove-lemma satisfiesp-helper-button2-means-list-of-zero-and-one (rewrite)
  (implies
   (and
    (satisfiesp-helper name pred tr h c bs)
    (equal bs (cadr (button2-spec)))
    (assoc name bs)
    (equal pred (state-pred (assoc name bs)))
    (not (assoc 'signal2 c))
    (assoc '(c 3) c))
   (list-of-zero-and-one
    (list-of-cadrs
     (list-of-cars
      (extract-from-history (list (list x 'signal2)) h)))))
  ((induct (state-preds-satisfiedp name pred tr h c bs))
   (use (satisfiesp-means-state-pred-satisfied))
   (disable-theory t)
   (expand (satisfiesp-helper name pred tr h c bs))
   (enable-theory ground-zero)
   (enable button2-spec state-trans-list trans-to state-name state-pred
           state-preds-satisfiedp extract-from-history list-of-cars list-of-cadrs
           satisfiesp-helper-nlistp satisfiedp-helper-bad-trans-name list-of-zero-and-one
           assoc-append extract-from-assignment my-eval assoc-append
           assoc-update-counter-alist-iff trans-resets)))

(prove-lemma satisfiesp-button1-means-list-of-zero-and-one (rewrite)
  (implies
   (satisfiesp h (button1-spec))
   (list-of-zero-and-one
    (list-of-cadrs
     (list-of-cars
      (extract-from-history (list (list x 'signal1)) h))))))

(prove-lemma satisfiesp-button2-means-list-of-zero-and-one (rewrite)
  (implies
   (satisfiesp h (button2-spec))
   (list-of-zero-and-one
    (list-of-cadrs
     (list-of-cars
      (extract-from-history (list (list x 'signal2)) h))))))


(prove-lemma satisfiesp-button1-means-variable-not-overflow (rewrite)
  (implies
```

```
   (satisfiesp h (button1-spec))
   (variable-not-overflowed-in-historyp 'signal1 h))
  ((use (variable-not-overflowed-from-list-of-zero-and-one
         (x 'signal1)))))

(prove-lemma satisfiesp-button2-means-variable-not-overflow (rewrite)
  (implies
   (satisfiesp h (button2-spec))
   (variable-not-overflowed-in-historyp 'signal2 h))
  ((use (variable-not-overflowed-from-list-of-zero-and-one
         (x 'signal2)))))

(prove-lemma behavior-state-list-quiz-program-spec (rewrite)
  (equal
   (behavior-state-list (quiz-program-spec))
   '((read-b1
      (lessp (c 9) 50)
      ((read-b1 ()
         (and (equal (old b1) b1) (and (equal (old b2) b2) (and (equal (old s) s)
              (and (equal (old l1) l1) (equal (old l2) l2))))))
       (read-b2 ()
         (and (equal (old signal1) b1) (and (equal (old b2) b2) (and (equal (old s) s)
              (and (equal (old l1) l1) (equal (old l2) l2))))))))
     (read-b2
      (lessp (c 9) 100)
      ((read-b2 ()
         (and (equal (old b1) b1) (and (equal (old b2) b2) (and (equal (old s) s)
              (and (equal (old l1) l1) (equal (old l2) l2))))))
       (update-state ()
         (and (equal (old b1) b1) (and (equal (old signal2) b2) (and (equal (old s) s)
              (and (equal (old l1) l1) (equal (old l2) l2))))))))
     (update-state
      (lessp (c 9) 300)
      ((update-state ()
         (and (equal (old b1) b1) (and (equal (old b2) b2) (and (equal (old s) s)
              (and (equal (old l1) l1) (equal (old l2) l2))))))
       (update-l1 ()
         (and (equal (old b1) b1) (and (equal (old b2) b2)
              (and (equal s (if (or (equal (old s) 0) (equal (old s) 1))
                                 (if (equal b1 1) 1 (if (equal b2 1) 2 0))
                               (if (equal b2 1) 2 (if (equal b1 1) 1 0))))
              (and (equal (old l1) l1) (equal (old l2) l2))))))))
     (update-l1
      (lessp (c 9) 350)
      ((update-l1 ()
         (and (equal (old b1) b1) (and (equal (old b2) b2) (and (equal (old s) s)
              (and (equal (old l1) l1) (equal (old l2) l2))))))
       (update-l2 ()
         (and (equal (old b1) b1) (and (equal (old b2) b2) (and (equal s (old s))
              (and (equal l1 (if (equal s 1) 0 1)) (equal (old l2) l2))))))))
     (update-l2
      (lessp (c 9) 400)
      ((update-l2 ()
         (and (equal (old b1) b1) (and (equal (old b2) b2) (and (equal (old s) s)
              (and (equal (old l1) l1) (equal (old l2) l2))))))
       (read-b1 ((c 9))
         (and (equal (old b1) b1) (and (equal (old b2) b2) (and (equal s (old s))
              (and (equal (old l1) l1) (equal l2 (if (equal s 2) 0 1))))))))))))

(prove-lemma value-from-extension-of-quiz-program-operation (rewrite)
  (implies
   (and
```

```
    (extension-historyp
     (quiz-program-operation pc s b1 b2 s1-list s2-list l1 l2 temp z c)
     h)
    (listp s1-list))
   (and
    (equal (cadr (assoc 'l1 (car h))) l1)
    (equal (cadr (assoc 'l2 (car h))) l2)
    (equal (cadr (assoc 's (car h))) s))))

(prove-lemma extension-historyp-fm9001-quiz-hack (rewrite)
  (implies
   (and
    (listp h)
    (variable-not-overflowed-in-historyp 'signal1 h)
    (variable-not-overflowed-in-historyp 'signal2 h)
    (extension-historyp
     (fm9001-rt-history-repeat-helper
      s 3
      (extract-from-history '((#xc0000000 signal1) (#xc0000002 signal2)) h)
      '(((15) pc) ((3) s) ((1) b1) ((2) b2) (#xc0000000 signal1) (#xc0000002 signal2)
        (#xc0000001 l1) (#xc0000003 l2)))
     h)
    (good-initial-fm9001-quiz-statep s))
   (and
    (equal (cadr (assoc 'l2 (car h))) 1)
    (equal (cadr (assoc 'l1 (car h))) 1)
    (equal (cadr (assoc 's (car h))) 0)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable fm9001-repeat-helper-quiz-embedded quiz-program-functioning-invariants regs
           quiz-program-correctness-invariants *1*nat-to-v *1*nth *1*read-mem nth-small
           good-initial-fm9001-quiz-statep listp-list-of-cars listp-list-of-cadrs
           listp-extract-from-history value-from-extension-of-quiz-program-operation)))

(prove-lemma good-initial-fm9001-quiz-statep-means-not-read-write (rewrite)
  (implies
   (good-initial-fm9001-quiz-statep s)
   (not (read-write-statep s))))

;; combine each of the main lemmas into almost-final theorem
(prove-lemma quiz-system-correct-exposed nil
  (implies
   (and
    (listp h)
    (good-initial-fm9001-quiz-statep s)
    (satisfiesp h (button1-spec))
    (satisfiesp h (button2-spec))
    (extension-historyp
     (fm9001-rt-history-repeat-helper
      s 3
      (extract-from-history '((#xc0000000 signal1) (#xc0000002 signal2)) h)
      '(((15) pc) ((3) s) ((1) b1) ((2) b2) (#xc0000000 signal1) (#xc0000002 signal2)
        (#xc0000001 l1) (#xc0000003 l2)))
     h))
   (satisfiesp h (quiz-spec)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable extension-satisfiesp *1*subset *1*all-variables-in-state-list *1*list-of-cadrs
           good-initial-fm9001-quiz-statep-means-not-read-write
           satisfiesp-button1-means-list-of-zero-and-one
           satisfiesp-button2-means-list-of-zero-and-one listp-extract-from-history
           bound-in-history-fm9001-rt-history-repeat-helper
```

```
          extension-historyp-fm9001-quiz-hack satisfiesp-button1-means-variable-not-overflow
          satisfiesp-button2-means-variable-not-overflow
          behavior-state-list-quiz-program-spec)
   (use (abstract-quiz-system-correctness)
        (fm9001-rt-history-satisfies-program-spec))))

;; convert to official rt FM9001 model
(prove-lemma quiz-system-correct-v2 nil
  (implies
   (and
    (good-initial-fm9001-quiz-statep s)
    (satisfiesp h (button1-spec))
    (satisfiesp h (button2-spec))
    (extension-historyp
     (fm9001-rt-history
      s 3
      (extract-from-history '((#xc0000000 signal1) (#xc0000002 signal2)) h)
      '(((15) pc) ((3) s) ((1) b1) ((2) b2) (#xc0000000 signal1) (#xc0000002 signal2)
        (#xc0000001 l1) (#xc0000003 l2)))
     h))
   (satisfiesp h (quiz-spec)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable fm9001-rt-repeat-helper-equiv fm9001-rt-history satisfiesp satisfiesp-helper)
   (use (quiz-system-correct-exposed))))

(prove-lemma extension-historyp-repeat-with-inputs-copy-unnecc2 (rewrite)
  (implies
   (and
    (not (member-v-iff (nat-to-v (car a) 32) (list-nat-to-v (list-of-cars b) 32)))
    (variable-not-overflowed-in-historyp (cadr a) h)
    (ramp-mem (nat-to-v (car a) 32) (cadr s))
    (nlistp (car a)))
  (equal
   (extension-historyp
    (repeat-with-inputs s (extract-from-history (cons x (cons a b)) h) (cons a c))
    h)
   (extension-historyp
    (repeat-with-inputs s (extract-from-history (cons x (cons a b)) h) c)
    h)))
  ((induct (length h))
   (disable-theory t)
   (enable-theory ground-zero)
   (enable prove-boolp bv2p nat-to-v length-nat-to-v bvp-nat-to-v v-iff=equal ramp-mem
          v-to-nat-of-nat-to-v equal-iff
          when-the-mantissa-isnt-0-then-neither-is-the-exponential update-state-with-inputs
          extract-locs repeat-with-inputs extension-assignmentp extension-historyp
          extract-from-assignment extract-from-history list-of-cars list-nat-to-v
          member-v-iff update-state-with-inputs-write-mem read-mem-write-mem-rewrite
          variable-not-overflowed-in-historyp list-of-cars-extract-from-assignment
          ramp-mem-update-state-with-inputs)))


(prove-lemma extension-history-fm9001-repeat-helper-copy-unnecc2 (rewrite)
  (implies
   (and
    (not (member-v-iff (nat-to-v (car a) 32) (list-nat-to-v (list-of-cars b) 32)))
    (variable-not-overflowed-in-historyp (cadr a) h)
    (ramp-mem (nat-to-v (car a) 32) (cadr s))
    (nlistp (car a))
    (equal i (extract-from-history (cons x (cons a b)) h)))
   (equal
```

```
    (extension-historyp
     (fm9001-rt-history-repeat-helper s m i (cons a c))
     h)
    (extension-historyp
     (fm9001-rt-history-repeat-helper s m i c)
     h)))
  ((induct (fm9001-rt-history-repeat-helper-and-h-induct s m i h))
   (disable-theory t)
   (enable-theory ground-zero naturals)
   (expand
    (fm9001-rt-history-repeat-helper
     s m (extract-from-history (cons x (cons a b)) h) (cons a c))
    (fm9001-rt-history-repeat-helper
     s m (extract-from-history (cons x (cons a b)) h) c))
   (enable equal-append-append extension-historyp-append length-extract-from-history
           nthcdr-nthcdr nthcdr-firstn length-repeat-with-inputs lessp-total-time-init-time
           ramp-mem-update-state-with-inputs ramp-mem-fm9001-step length-nthcdr
           nthcdr-extract-from-history firstn-extract-from-history
           variable-not-overflowed-in-historyp-firstn
           variable-not-overflowed-in-nthcdr-history nthcdr-extract-from-history
           length-firstn extension-historyp-repeat-with-inputs-copy-unnecc2)))


(prove-lemma extension-history-fm9001-copy-unnecc2 (rewrite)
 (implies
  (and (not (member-v-iff (nat-to-v (car a) 32)
                          (list-nat-to-v (list-of-cars b) 32)))
       (variable-not-overflowed-in-historyp (cadr a)
                                            h)
       (ramp-mem (nat-to-v (car a) 32) (cadr s))
       (nlistp (car a)))
  (equal
     (extension-historyp (fm9001-rt-history s m
                                            (extract-from-history (cons x (cons a b)) h)
                                            (cons a c))
                         h)
     (extension-historyp (fm9001-rt-history s m
                                            (extract-from-history (cons x (cons a b)) h)
                                            c)
                         h)))
 ((use (extension-history-fm9001-repeat-helper-copy-unnecc2
        (i (extract-from-history (cons x (cons a b)) h))))
  (disable-theory t)
  (enable fm9001-rt-repeat-helper-equiv fm9001-rt-history)
  (enable-theory ground-zero)))


(prove-lemma ramp-initial-quiz-state (rewrite)
  (implies
   (good-initial-fm9001-quiz-statep s)
  (and
   (ramp-mem (list f f f f f f f f f f f f f f f f f f f f f f f f f f f f f f t t)
             (cadr s))
   (ramp-mem (list f t f f f f f f f f f f f f f f f f f f f f f f f f f f f f t t)
             (cadr s)))))

(prove-lemma quiz-system-correct-ver3 nil
  (implies
   (and
    (listp h)
    (good-initial-fm9001-quiz-statep s)
    (satisfiesp h (button1-spec))
```

```
    (satisfiesp h (button2-spec))
    (extension-historyp
     (fm9001-rt-history
      s 3
      (extract-from-history '((#xc0000000 signal1) (#xc0000002 signal2)) h)
      '((#xc0000001 l1) (#xc0000003 l2)))
     h))
   (satisfiesp h (quiz-spec)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (use (quiz-system-correct-v2
         (h (zipper-lists
             (fm9001-rt-history
              s
              3
              (extract-from-history '((#xc0000000 signal1) (#xc0000002 signal2)) h)
              '(((15) pc) ((3) s) ((1) b1) ((2) b2)))
             h)))
        (permutation-means-extension-fm9001-rt-history-same
         (m 3)
         (i (extract-from-history '((#xc0000000 signal1) (#xc0000002 signal2)) h))
         (x '(((15) pc) ((3) s) ((1) b1) ((2) b2) (#xc0000000 signal1) (#xc0000002 signal2)
              (#xc0000001 l1) (#xc0000003 l2)))
         (y '(((15) pc) ((3) s) ((1) b1) ((2) b2) (#xc0000000 signal1) (#xc0000002 signal2)
              (#xc0000001 l1) (#xc0000003 l2)))
         (z (zipper-lists
             (fm9001-rt-history
              s
              3
              (extract-from-history '((#xc0000000 signal1) (#xc0000002 signal2)) h)
              '(((15) pc) ((3) s) ((1) b1) ((2) b2)))
             h))))
   (enable listp-zipper-lists firstn-too-big member-append subset-cons satisfiesp-nlistp
           ramp-initial-quiz-state not-lessp-length-from-extension-historyp-fm9001
           state-trans-list behavior-initial behavior-state-list make-list-alist
           counters-of-state-list extension-history-fm9001-copy-unnecc
           extension-history-fm9001-copy-unnecc2 satisfiesp-nlistp *1*intersection
           extension-historyp-extract-from-history-ziplist intersection-single-element
           *1*subset equal-length-from-extension-historyp-fm9001 satisfiesp-zipper-lists
           button1-spec button2-spec quiz-spec trans-pred trans-resets state-name
           all-variables-in-term counters-of-trans counters-of-term state-pred
           counters-of-translist counters-of-state set-union all-variables-in-translist
           *1*all-variables-in-state-list extension-historyp list-of-cadrs set-difference
           all-bound-in-history-fm9001-rt-history listp-extract-from-history
           satisfiesp-button1-means-variable-not-overflow
           satisfiesp-button2-means-variable-not-overflow
           good-initial-fm9001-quiz-statep-means-not-read-write listp-fm9001-rt-history
           extract-from-history-zipper-lists extract-from-history-make-properp
           intersection-nil all-litatoms no-dup-cadrs *1*nat-to-v *1*list-of-cars
           *1*list-nat-to-v *1*member-v-iff *1*v-iff)))


(prove-lemma quiz-system-correct nil
  (implies
   (and
    (good-initial-fm9001-quiz-statep s)
    (satisfiesp h (button1-spec))
    (satisfiesp h (button2-spec))
    (extension-historyp
     (fm9001-rt-history
      s 3
      (extract-from-history '((#xc0000000 signal1) (#xc0000002 signal2)) h)
```

```
      '((#xc0000001 l1) (#xc0000003 l2)))
     h))
   (satisfiesp h (quiz-spec)))
  ((use (quiz-system-correct-ver3))
   (disable-theory t)
   (enable-theory ground-zero)
   (enable satisfiesp-nlistp)))

(prove-lemma good-initial-fm9001-quiz-statep-reasonable nil
  (good-initial-fm9001-quiz-statep (initialized-quiz-state))
  ((enable *1*good-initial-fm9001-quiz-statep *1*initialized-quiz-state)
   (disable-theory t)))

))
```

Index of Nqthm Definitions and Theorems

BIBLIOGRAPHY

[1] Robert L. Akers, Donald I. Good, Lawrence M. Smith, and William D. Young. Report on Micro-Gypsy. Technical Report 34, Computational Logic, Inc., November 1988.

[2] Ken Albin and Warren Hunt. The FM9001 single-board computer. Technical Report 90, Computational Logic, Inc., 1993.

[3] Rajeev Alur and David Dill. Automata for modeling real-time systems. In M.S. Paterson, editor, *Automata, Languages and Programming – 17th International Colloquium*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer-Verlag, 1990.

[4] AMD. *PALASM User's Guide*. Advanced Micro Devices, 1990.

[5] William R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1368–81, November 1989. Also published as CLI Technical Report 28.

[6] William R. Bevier, Warren A. Hunt Jr., J Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.

[7] W.R. Bevier. A library for hardware verification. Internal Note 57, Computational Logic, Inc., June 1988.

[8] W.W. Bledsoe. The Sup-Inf method in Presberger arithmetic. Technical Report ATP-18, Mathematics Department, University of Texas at Austin, 1975.

[9] Charles L. Bouton. Nim, a game with a complete mathematical theory. In *Annals of Mathematics*, volume 3, 1901-02.

[10] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, Boston, 1988.

[11] R. S. Boyer and J S. Moore. *Chapter 14 of A Computational Logic Handbook, 2nd edition*. Academic Press, Boston, to be published. (Distributed in Nqthm-1992 release available from Computational Logic, Inc.).

[12] Robert S. Boyer, M. W. Green, and J S. Moore. The use of a formal simulator to verify a simple real-time control program. In *Beauty is Our Business*, pages 54–66. Springer-Verlag, 1990.

[13] Robert S. Boyer and Yuan Yu. Automated correctness proofs of machine code programs for a commercial microprocessor. In D. Kapur, editor, *Automated Deduction – CADE-11*, number 607 in Lecture Notes in Computer Science, pages 416–430. Springer-Verlag, 1992.

[14] Bishop C. Brock, Warren A. Hunt Jr., Matt Kaufmann, Art Flatau, Ann Siebert, and William D. Young. The formal specification and verification of the FM9001 microprocessor. Technical Report 86, Computational Logic, Inc., July 1993.

[15] Moyer Chen. TAL - a language for timing analysis. unpublished technical report, University of Texas, 1987.

[16] Nancy Day. A model checker for statecharts (linking CASE tools with formal methods. Technical Report 93-35, Department of Computer Science, University of British Columbia, October 1993.

[17] Colin Fidge, Peter Kearney, and Mark Utting. Formal specification and interactive proof of a simple real-time scheduler. Technical Report 94-11, Software Verification Research Centre, The University of Queensland, April 1994.

[18] Arthur Flatau. A verified implementation of an applicative language with dynamic storage allocation. Technical Report 83, Computational Logic, Inc., November 1992.

[19] Martin Gardner. *Mathematical Puzzles and Diversions.* Simon and Schuster, New York, 1959.

[20] D.M. Goldschlag. Mechanizing unity. In *Proceedings of IFIP TC2 Working Conference on Programming Concepts and Methods.* Elsevier Science Publishers, 1990.

[21] Donald I. Good and William D. Young. Mathematical methods for digital systems development. Technical Report 67, Computational Logic, Inc., August 1991.

[22] Mike Gordon. A mechanized Hoare logic of state transitions. Technical report, University of Cambridge Computer Laboratory, June 1993.

[23] Mike Gordon. Overview of HOL. In *Towards Verified Systems.* 1993. draft chapter, to appear.

[24] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete Mathematics.* Addison Wesley, 1988.

[25] Wolfgang A. Halang and Alexander D. Stoyenko. *Constructing Predictable Real Time Systems*. Kluwer Academic Press, 1991.

[26] David Harel. Statecharts: A visual formalism for complex systems. Technical report, Dept. of Applied Mathematics, The Weizmann Institute of Science, July 1986.

[27] Warren A. Hunt and Bishop Brock. A formal HDL and its use in the FM9001 verification. *Proceedings of the Royal Society*, April 1992.

[28] F. Jahanian and A. Mok. Safety analysis of timing properties in real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, September 1986.

[29] Farnam Jahanian and Aloysius K. Mok. Modechart: A specification language for real-time systems. *IEEE Transactions on Software Engineering*, 20(12):933–947, December 1994.

[30] Matt Kaufmann. A user's manual for an interactive enhancement to the Boyer-Moore theorem prover. Technical Report 19, Computational Logic, Inc., May 1988.

[31] M.J. Kaufmann. An integer library for Nqthm. Internal Note 182, Computational Logic, Inc., 1990.

[32] M.J. Kaufmann and Paolo Pecchiari. Interaction with the Boyer-Moore and theorem prover: A tutorial study using the arithmetic-geometric mean theorem. Technical Report 100, Computational Logic, Inc., August 1994. (to appear in *Journal of Automated Reasoning*).

[33] Peter Kearney and Mark Utting. A layered real-time specification of a RISC processor. In Costas Courcoubetis, editor, *Computer-aided Verification – CAV '93*, volume 697 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[34] C.L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, 1973.

[35] Al Mok, Private Communication, March 1993.

[36] Al Mok. SARTOR - a design environment for real-time systems. In *Proceedings Compsac*, October 1985.

[37] Aloysius K. Mok, Prasanna Amerasinghe, Moyer Chen, and Karntorn Tantisirivat. Evaluating tight execution time bounds of programs by annotations. In *Proceedings of IEEE Real-Time Operating Systems and Software Workshop*, May 1989.

[38] J Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4):493–518, December 1989. Also published as CLI Technical Report 30.

[39] John B. Peatman. *Design with Microcontrollers*. McGraw-Hill, 1988.

[40] Gustav Pospischil, Peter Puschner, Alexander Vrchoticky, and Ralph Zainlinger. Developing real-time tasks with predictable timing. *IEEE Software*, pages 35–44, September 1992.

[41] Anne Rose, Maunel A. Perez, and Paul Clements. Modechart toolset user's guide. Technical report, Center for Computer High Assurance Systems, US Navy Research Laboratory.

[42] John Rushby. Critical system properties: Survey and taxonomy. Technical Report CSL-93-01, Computer Science Laboratory, SRI International, May 1993.

[43] Larry Smith. Programming a PAL. Internal note, Computational Logic, Inc., 1990.

[44] M.K. Smith, Donald I. Good, and Benedetto L. DiVito. *Using the Gypsy Methodology.* Computational Logic Inc., 1986. Revised January 1988.

[45] John A. Stankovic, Marco Spuri, and Marco Di Natale. Implications of classical scheduling results for real-time systems. Technical report, University of Massachusetts, April 1993.

[46] R. Tol. A small real-time kernel proven correct. *Proceedings of the Real-Time Systems Symposium*, pages 227–230, 1992.

[47] John Ursoleo. Software performs debouncing for large array of switches. *Electronic Design*, October 3 1994.

[48] M. Utting and P. Kearney. Pipeline specification of a mips r3000 cpu. Technical Report 93-25, Software Verification Research Cenre, The University of Queensland, February 1994.

[49] M. Utting and K. Whitwell. Ergo user manual. Technical Report 93-19, Software Verification Research Cenre, The University of Queensland, February 1994.

[50] Larry Wall and Randal L. Schwartz. *Programming PERL.* O'Reilly and Associates, Inc., California, 1990.

[51] Matthew Wilding. A mechanically-checked correctness proof of a floating-point search program. Technical Report 56, Computational Logic, Inc., May 1990.

[52] Matthew Wilding. Events for the proof of rational facts needed in a distributed algorithm correctness proof. Internal Note 222, Computational Logic, Inc., March 1991.

[53] Matthew Wilding. Proving Matijasevich's lemma with a default arithmetic strategy. *Journal of Automated Reasoning*, 7(3), September 1991.

[54] Matthew Wilding. A verified nim strategy. Internal Note 249, Computational Logic, Inc., November 1991.

[55] Matthew Wilding. Using the FM9001 single-board computer. Internal Note 260, Computational Logic, Inc., 1992.

[56] Matthew Wilding. A mechanically verified application for a mechanically verified environment. In Costas Courcoubetis, editor, *Computer-aided Verification – CAV '93*, volume 697 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[57] Matthew Wilding. A detailed processor model for verification of real-time applications. In *Proceedings of the 2nd IFAC Workshop on Safety and Reliability*, IFAC Control Notes, 1995.

[58] W.D. Young W.R. Bevier. The proof of correctness of a fault-tolerant circuit design. In *Proceedings of the Second International Working Conference on Dependable Computing for Critical Applications*, pages 107–114. IFIP, February 1991.

[59] W.D. Young. A verified code generator for a subset of Gypsy. Technical Report 33, Computational Logic, Inc., 1988. Ph.D. Thesis, University of Texas at Austin.

[60] William D. Young. Verifying a simple real-time system with Nqthm. Internal Note 288, Computational Logic, Inc., 1993.

[61] Zheng Yuhua and Zhou Chaochen. A formal proof of the deadline driven scheduler. *Formal Techniques in Real-Time and Fault-Tolerant Systems, Third International Symposium*, pages 756–775, 1994.

Vita

Matthew Michael Wilding was born in Washington, D.C., on May 26, 1962, the son of Marcella Gibbons Wilding and James Anthony Wilding. He received a Bachelor of Science in Computer Science from Virginia Tech in 1984, having spent several semesters as a cooperative education student at The BDM Corporation in McLean, Virginia. After graduation he began full-time work as a defense analyst at BDM, and left in 1985 to attend the University of Texas at Austin. He taught computer programming at the University of Texas in 1987 and 1988. In 1988 he received a Master of Science in Computer Sciences, and began work at Computational Logic, Inc, in Austin. He is the author of several published papers and numerous technical reports.

In 1992 he married Martha Roebuck Wilding of Austin, and in 1993 they were blessed with a son, James Robert "Jake" Wilding.

Permanent address: 1404 Norwalk #101,
Austin, Texas, 78703

This dissertation was typeset with LaTeX by the author.