Automated Proofs of Object Code for a Widely Used Microprocessor[1]

Robert S. Boyer and *Yuan Yu* [2]

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712
telephone: (512) 471-9745
email: boyer@cs.utexas.edu or yuanyu@src.dec.com

Abstract. We have formally described a substantial subset of the MC68020, a widely used microprocessor built by Motorola, within the mathematical logic of the automated reasoning system Nqthm, a.k.a. the Boyer-Moore Theorem Prover [6]. Using this formal description, we have mechanically checked the correctness of MC68020 object code programs for binary search, Hoare's Quick Sort, twenty-one functions from the Berkeley Unix C string library, and other well-known algorithms. The object code for these examples was generated using the Gnu C, the Verdix Ada, and the Gnu Common Lisp compilers. We have mechanized a mathematical theory to facilitate automated reasoning about object code programs. We describe a two stage methodology we use to do our proofs.

Key words. Automated reasoning, Boyer-Moore logic, Nqthm, formal methods, machine code, program verification, C, Ada, Common Lisp.

1 Introduction

To search for mistakes in a computer program one can try to prove that the execution of the program according to a formal model of computation satisfies a formal specification. To reduce the chance of mistakes in such proofs, one can use an automated reasoning system. Thus far, the bulk of research in formal, mechanical program proving has focused on programs written in higher-level languages.

This paper describes how we have formally defined, within the logic of the automated reasoning system Nqthm [6], a substantial subset of the user model of the widely used Motorola MC68020 microprocessor. Using Nqthm, we have proved that some object code generated by "industrial strength" high-level programming language compilers, such as the Gnu C compiler, satisfies certain formal specifications.

[1]The work described here was supported in part by NSF Grant MIP-9017499.

[2]The current address for Yuan Yu is Digital Equipment Corporation, Systems Research Center, 130 Lytton Ave, Palo Alto, CA 94301.

Why study program proving at the object code level?

- If we take as our goal ensuring that programs are executed correctly on particular processors, then the semantics of machine code on those processors must be considered.[3]
- Some of the most critical programs in the world are currently studied at the object code level anyway, even though written in higher-level programming languages, and for several good reasons:
  - Many high-level programming languages, especially those typically used in industrial practice, are not precisely specified. It is not easy, or even possible, to give the semantics of some programming language features, for example, the `volatile` type in C.
  - Some "industrial strength" compilers produce erroneous code.
- Programs written in high-level languages may have assembly code embedded in them, in order to communicate with external devices. But no high-level formal language semantics we have seen has yet made clear the semantics of the embedding of assembler instructions.
- Real-time analysis is typically done at the machine instruction level because manufacturers often state how long an instruction takes to execute, but the definers of higher-level languages do not.

Our approach of proving theorems about object code rather than higher level programs addresses all these problems. For example, when we are proving theorems about object code, we have no need for a formal semantics of the higher-level language in which the program may have originally been written. Any mistakes in the object code introduced by the compiler can be revealed by studying the object code.

Our approach of studying the object code produced by higher-level language compilers permits a programmer to continue to code in any higher-level language. We are not advocating a return to coding in assembler or binary.

Throughout this paper, the word "formalize" means to write specifications in a formal logic. In our work, we write specifications in the logic of Nqthm.

2 The Automated Reasoning System Nqthm

We briefly review the automated reasoning system Nqthm, also known as "the Boyer-Moore Theorem Prover." Detailed knowledge of Nqthm is unnecessary

[3]It is relevant to review Knuth's defense, in the Preface to *The Art of Computer Programming* [18], of his decision to present algorithms in assembly code rather than in a higher-level language.

for those who are happy enough with the informal paraphrases of the formulas in the remainder of this paper. Nqthm is a Common Lisp program for proving mathematical theorems. Since *A Computational Logic* [5] was published in 1979, Nqthm has been used by several dozen users to check proofs of over 16,000 theorems from many areas of number theory, proof theory, and computer science. An extensive partial listing may be found in [6, pages 5–9]. See also [2]. For a thorough and precise description of the Nqthm logic, we refer the reader to the rigorous treatment in [6], especially Chapter 4, in which the logic is precisely defined. In the body of this paper, we use a conventional syntax rather than the official Lisp-like syntax of Nqthm.[4]

2.1 The Logic

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

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

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

- *The Shell Principle* allows the user to add axioms introducing "new" inductively defined "abstract data types." Nonnegative integer, ordered pairs, and symbols are axiomatized in the logic by adding shells:
  - *Nonnegative Integer.* The nonnegative integers are built from the constant 0 by successive applications of the constructor function 'add1'. The function 'numberp' recognizes nonnegative integers. The function 'sub1' returns the predecessor of a non-0 nonnegative integer. $x \in \mathbf{N}$ abbreviates numberp(x).

[4]The translation between the conventional syntax and the official Lisp-like syntax is discussed in [7].

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

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

The rules of inference of the logic are those of propositional logic and equality with the addition of mathematical induction.

2.2 The Theorem Prover

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

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

The commands to the theorem prover include those for defining new functions, proving lemmas, and adding shells. The following two commands are the most often used.

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

DEFINITION: fn-name $(x,\ y) = \mathit{body}$

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

  THEOREM: lemma-name
  statement

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

3 The MC68020 Instruction Set Specification

We have formalized most of the user programming model of the MC68020 microprocessor. The formal specification is intended to reflect as closely as possible the user's manual view of the MC68020 [22]. (A closely related instruction set will also be found on other Motorola MC68xxx microprocessors and microcontrollers, but we focused our formalization effort entirely on the MC68020.) We, at the present time, have avoided considering the supervisor level of the MC68020. Any exception caused by user programs simply halts our formalized machine. Before presenting our formal specification, we first give an informal description of the user programming model of the MC68020 and our formalism.

3.1 Formalizing the MC68020 Instruction Set Architecture

We have followed the state transition approach to formalizing the MC68020 microprocessor. We define the MC68020 as an abstract machine and the MC68020 instructions as operations on the states of the abstract machine. We specify the "semantics" of this abstract machine as a function in the Nqthm logic in the most straightforward way: fetch the current instruction in the current state, decode the instruction, perform the operation, and return a new machine state suitably altered.

Figure 1 provides an informal, two dimensional picture of the user "programming model" for the MC68020, as described in [22]. This model has 16 32-bit general-purpose registers (8 data registers, D0-D7, and 8 address registers, A0-A7), a 32-bit program counter PC, and an 8-bit condition code register, CCR.

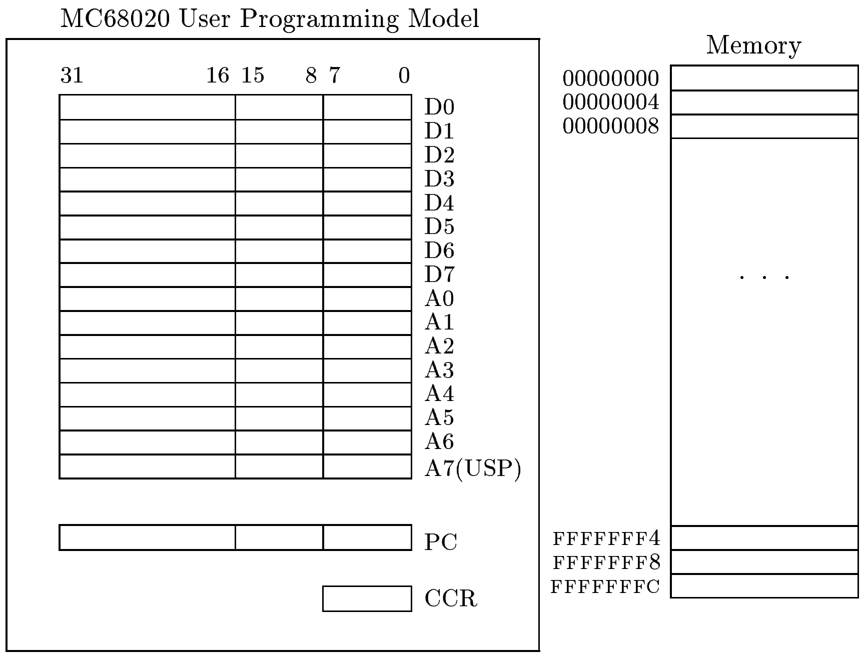

Figure 1: The User Visible Machine State

The address register A7 is also used as the user stack pointer (USP). The 5 least significant bits in CCR are condition codes for carry, overflow, zero, negative, and extend. This "model" is the only part of the state of an MC68020 that a user program can read or write under our formal semantics. Not present in this model are such processor actualities as the instruction cache, memory management, and the supervisor stack.

In our formalization, we have focused exclusively on the user available instructions of the MC68020 instruction set. Our specification consists of about 80% of all the user available instructions. Most of the instructions we have left unspecified have some *undefined* effects on the machine state. For example, some of the condition codes of the instruction `CMP2` are described as *undefined* in [22]. We have deliberately excluded these instructions in our specification. Fortunately, these instructions constitute only a small portion of the instruction set, and most of them are rarely used.[5] We summarize below those instructions formalized.

The instructions of the MC68020 instruction set are classified into ten categories according to their functions [22].

[5] We have not yet encountered such instructions in the programs we have studied.

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

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

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

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

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

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

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

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

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

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

We have formalized all eighteen MC68020 addressing modes. An addressing mode can specify a constant that is the operand, a register that contains the operand, or a location in memory where the operand is stored. For a complete description of the MC68020 addressing modes, we refer the reader to Motorola's MC68020 user's manual [22].

3.2 The Formal Instruction-Level Specification

Before we present some details of the formal specification, we first formally define the user visible state and its internal representation.

3.2.1 The User Visible State

The only type of object manipulated at the instruction level is the *bit vector*, which is represented as a nonnegative integer in our specification. For example, the content of the program counter is represented as a nonnegative integer with range between 0 and $2^{32} - 1$, inclusive. Each of the operations on bit vectors can then be formalized as an operation on nonnegative integers. Here are a few basic operations on bit vectors and their definitions in the Nqthm logic.

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

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

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

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

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

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

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

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

Intuitively, 'head' returns the bit vector of the first n low-order bits of x; 'tail' returns the bit vector obtained by discarding the first n low-order bits of x; 'bcar' and 'bcdr' are simply the special cases of 'head' and 'tail' with $n = 1$; 'bitn' returns the nth bit of the bit vector x; 'mbit' is simply a special case of 'bitn', returning the most significant bit of x; and 'app' returns the bit vector obtained by concatenating x and y.

A user visible state is represented as a list of length five, e.g., list (*status*, *regs*, *pc*, *ccr*, *mem*), where the contents of each of the five fields has the following interpretation:

- *status* is the machine status word, which is either the symbol `'running` or some error message if an exception occurs. This status field is not actually present in any MC68020. Rather, it is the artifice of our state formalization by which we indicate that an actual error has arisen or that an aspect of the MC68020 not defined in our formalization has been encountered during execution.
- *regs* is the register file, which is represented as a list of 16 nonnegative integers.
- *pc* is the program counter, which is represented as a nonnegative integer.

- *ccr* is the condition code register, which is represented as a nonnegative integer.

- *mem* is the memory, which is represented as a pair of binary trees. A binary representation for memory provides some efficiency for simulating MC68020 instructions. One of the binary trees is a formalization of memory protection—one may specify that any byte of memory is `'ram`, `'rom`, or `'unavailable`; the other binary tree holds the data, i.e., the actual bytes stored. As discussed elsewhere in this paper, we use the notion of read-only memory to deal with the issue of cache consistency. We also believe that it is unrealistic to assert the correctness of machine-code programs without carefully characterizing which parts of memory are read and written—few MC68020 chips are connected to a full 4 gigabytes of RAM. Memory protection issues are not specified in [22].

The functions 'mc-status', 'mc-rfile', 'mc-pc', 'mc-ccr' and 'mc-mem' are accessors to the machine status word, the register file, the program counter, the condition codes and the memory, respectively. We use the function 'mc-haltp' to check whether a machine state is illegal:

DEFINITION: $\text{mc-haltp}\,(s) = (\text{mc-status}\,(s) \neq \texttt{'running})$

In this work, function names, such as 'mc-haltp', which end in the letter "p" generally name predicates, i.e., functions that return 't' or 'f', in the Lisp tradition.

Individual registers are accessed with the following functions. read-rn (*oplen*, *rn*, *regs*) returns the content in the register indexed by *rn* in the register file *regs*. The functions 'read-dn' and 'read-an' are used to obtain the register contents of the address and data register files respectively.

DEFINITION:
$\text{read-dn}\,(oplen,\ dn,\ s) = \text{read-rn}\,(oplen,\ dn,\ \text{mc-rfile}\,(s))$

DEFINITION:
$\text{read-an}\,(oplen,\ an,\ s) = \text{read-rn}\,(oplen,\ \texttt{8} + an,\ \text{mc-rfile}\,(s))$

As a special case of 'read-an', 'read-sp' returns the stack pointer in the given state:

DEFINITION: $\text{SP} = \texttt{7}$

DEFINITION: $\text{L} = \texttt{32}$

DEFINITION: $\text{read-sp}\,(s) = \text{read-an}\,(\text{L},\ \text{SP},\ s)$

Memory contents are accessed primarily by 'read-mem'. read-mem (*addr*, *mem*, *k*) returns the nonnegative integer obtained by appending together the *k* consecutive bytes from the memory *mem* starting at location *addr*.

'rts-addr' returns the address of the current subroutine call, which is on the user stack pointed by the user stack pointer.

DEFINITION:
rts-addr (s) = read-mem (read-an (`32`, `7`, s), mc-mem (s), `4`)

We now list some constants and very simple functions which are used in the subsequent sections, but whose definitions may be happily skipped on first reading:

DEFINITION: WSZ = `2`

DEFINITION: PC-SIGNAL = `'pc_outside_rom`

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

DEFINITION: B1 = `1`

DEFINITION: B0 = `0`

DEFINITION: b0p (x) = (x = B0)

DEFINITION:
```
b-not (x)
=   if b0p (x) then B1
    else B0 endif
```

DEFINITION:
```
b-and (x, y)
=   if b0p (x) then B0
    elseif b0p (y) then B0
    else B1 endif
```

DEFINITION:
```
b-or (x, y)
=   if b0p (x)
    then if b0p (y) then B0
         else B1 endif
    else B1 endif
```

DEFINITION:
```
fix-bit (c)
=   if b0p (c) then B0
    else B1 endif
```

DEFINITION: nat-rangep $(nat,\ n) = (nat < \exp(2,\ n))$

DEFINITION: bcs (c) = fix-bit (c)

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

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

DEFINITION:
iread-dn $(oplen,\ dn,\ s)$ = nat-to-int (read-dn $(oplen,\ dn,\ s)$, $oplen$)

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

DEFINITION: mod32-eq $(x,\ y)$ = (head $(x,\ 32)$ = head $(y,\ 32)$)

3.2.2 The Specification

Because of space limitations, we necessarily omit some of the details of our M68020 definition. The complete description may be found in [7].

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

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

```
DEFINITION:
stepn $(s,\ n)$
=   if mc-haltp $(s) \vee (n \simeq 0)$ then $s$
    else stepn (stepi $(s)$, $n - 1$) endif
```

'stepi' calls 'execute-ins' to compute the new machine state from the current state s by executing the current instruction if the program counter is aligned on a word boundary, as required by the MC68020, and points to readable memory, as is checked by the function 'pc-word-readp'. The function 'add' is defined below.

'stepn' executes n instructions by calling the single stepper 'stepi'. But 'stepn' halts prematurely if the status field of s ceases to be `'running`. The function 'halt' sets the status field of its second argument to be its first argument.

Roughly speaking, 'execute-ins' decodes the current instruction according to the opcode and jumps to the specification of the instruction identified. 'current-ins' obtains the current instruction. 'update-pc' updates the program counter in the current machine state to point to the next instruction.

In formalizing each individual instruction, we always proceed by the following four steps that are explained with our specification of the `ADD` instruction.

Addressing Modes. We first specify which addressing modes are available to the instruction. For each instruction, an addressing mode predicate is defined. For example, as a part of the `ADD` instruction, we define a function 'add-addr-modep1' to express the constraint that all the addressing modes are available to the `ADD` instruction except that a byte operation is not allowed in address register direct mode.[6]

Operation. We then define the operation performed by each of the instructions. The operation of the `ADD` instruction is:

$$\textsc{Definition:}\quad \text{add}\,(n,\, x,\, y) = \text{head}\,(x + y,\, n)$$

which is simply modulo addition: $(x+y) \bmod 2^n$.

Condition Codes. We then specify the new values of the five condition codes returned by each of the instructions. In our example, we formalize the five condition codes by the following four functions, paraphrasing the description given in Table 3-11 of the MC68020 manual [22]. For the ADD instruction, the X condition is the same as the C condition. The function *add-cvznx* creates the new condition code register.

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

```
DEFINITION:
add-v (n, sopd, dopd)
=   let result be add (n, sopd, dopd)
```

[6]This is only one of the two cases in the `ADD` instruction; please refer to [22] and [7] for more details.

```
in
b-or (b-and (b-and (mbit (sopd, n), mbit (dopd, n)),
             b-not (mbit (result, n))),
      b-and (b-and (b-not (mbit (sopd, n)), b-not (mbit (dopd, n))),
             mbit (result, n))) endlet
```

DEFINITION:

```
add-z (oplen, sopd, dopd)
=  if add (oplen, dopd, sopd) = 0  then B1
   else B0 endif
```

DEFINITION:

```
add-n (oplen, sopd, dopd) = mbit (add (oplen, dopd, sopd), oplen)
```

The auxilliary function 'cvznx' does the arithmetic to pack the fields into a bit vector.

DEFINITION:

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

DEFINITION:

```
add-cvznx (oplen, sopd, dopd)
=  cvznx (add-c (oplen, sopd, dopd),
          add-v (oplen, sopd, dopd),
          add-z (oplen, sopd, dopd),
          add-n (oplen, sopd, dopd),
          add-c (oplen, sopd, dopd))
```

Next Machine State. Finally, we define a main function for each of the instructions, a function that specifies the new machine state produced by the effects of the execution of that instruction. In our example, 'add-effect' returns the effects of the ADD instruction and 'add-ins1' returns the new machine state.[7]

DEFINITION:

```
add-effect (oplen, sopd, dopd)
=  cons (add (oplen, dopd, sopd), add-cvznx (oplen, sopd, dopd))
```

[7]In the definition of 'add-ins1', the ampersand character in the identifier 's&addr' has the same status as an ordinary alphabetic character in an Nqthm identifier.

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

Roughly speaking, 'add-ins1' checks whether *ins* is a legal `ADD` instruction, and, if so, proceeds to calculate the effective addresses, fetch the source and destination operands (with the functions 'operand' and 'read-dn'), perform the specific operation, and finally update the condition codes and store the results by calling the function 'd-mapping'. It may return an error state with a signal indicating the type of error encountered. The auxilliary function 'mc-instate' calculates a pair representing an internal state in the execution; in the case of an illegal instruction the pair's 'car' will be a halt signal.

Altogether, our formal specification consists of 569 function definitions. It takes up approximately 80 pages of text. About two thirds of the specification is devoted to the formalization of individual instructions.

Rather than giving more details for any particular instruction, all of which have been fully documented in [7], we instead focus on some of the interesting issues that have come up in the specification.

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

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

Condition Code Computation. Ideally, we would specify the condition codes in a way most natural to the "user." But in order to assure full compliance with the MC68020 specification [22], we have followed the syntactical definition described in Table 3-11 of [22]. For instance, we define the carry bit of the `SUB` instruction as follows:

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

To paraphrase this, the carry bit is set to $(Sm \wedge \overline{Dm}) \vee (Rm \wedge \overline{Dm}) \vee (Sm \wedge Rm)$, where Sm, Dm, and Rm denote the most significant bit of source, destination and result, respectively. This characterization is perhaps not the way the programmer views the carry bit of a `SUB` (subtraction) instruction! One of the problems we have to deal with in the verification phase is to eliminate these "semantic gaps."

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

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

- It is possible to "test" the specification's correctness by executing it on specific data and comparing the result with the behavior of an actual MC68020. While testing does not find all bugs, it does find some, and that helps us gain confidence in our formal model. Kenneth Albin of Computational Logic devoted several weeks to testing our formal MC68020 model against an MC68020, the processor of a Sun-3, on approximately 40,000 instructions, a substantial portion of the approximately 45,000 opcode combinations covered by the specification. Albin uncovered approximately a dozen errors in the formal specification, which we have corrected. It is interesting to note that all of the errors he found were in instructions, or modes of instructions, not exercised in any of the many machine-code programs verified. Most of the errors Albin found were of a typographic character, e.g., a left-for-right swap.

- By giving the MC68020 semantics entirely with definitions instead of with an *ad hoc* collection of axioms, we are guaranteed that the specification is consistent, relative to the consistency of elementary number theory.

- The executability of our formal model supports a fast means of symbolic manipulation in some cases during program proving, viz., when an expression is encountered that is variable free.

- As with most uses of Nqthm, we rely heavily on symbolic execution in our proofs. Our formal model was fine-tuned to increase the likelihood that the Nqthm simplifier would perform symbolic execution.

4 Object Code Proofs

Among the possible applications of the MC68020 formal specification, we are currently primarily concerned with studying the verification of specific object code programs. To date we have successfully verified many small object code programs generated from their C, Ada, and Common Lisp counterparts with the Gnu C compiler, the Verdix Ada compiler, and the Gnu Common Lisp Compiler. This section describes our work in this direction.

Our formal model gives a semantics for MC68020 machine-code programs. From a strictly formal perspective, this semantics is all that is needed to prove the correctness of MC68020 machine-code programs. Indeed, some theorems about explicitly given machine states can be proved by direct execution. But practically speaking, to get Nqthm to check the correctness of MC68020 machine-code programs, it is necessary first to develop a library of Nqthm-checked lemmas, a library that describes some of the useful mathematical properties of our formal semantics. In this section, we first describe our lemma library.

4.1 A Library of Lemmas

The development of lemmas is the key to success in any use of an interactive theorem proving system, certainly of Nqthm. Lemmas are saved as derived inference rules that affect the future behavior of the system. The quality of the lemmas often determines the success of the entire proof effort. Our approach to developing a lemma library can be roughly viewed as "bottom-up." We carefully study each of the concepts involved, in the hope of proving a set of lemmas that fully characterizes these concepts. In general, the library is intended to be the mechanization of a basic theory of formal reasoning about object code programs which will have utility in the verification of many different programs.

We have invested more time creating our lemma database than on any other aspect of this multi-year project. It takes a lot of effort to formulate the lemmas in such a way that the theorem prover can find them at the "right time" and apply them automatically. This investment has paid off, because many proofs involve the application of a large collection of lemmas. Each of the lemmas is proved by Nqthm before it is admitted into the system. Allowing users of theorem provers to assert without proof the lemmas they think correct seems, with some historical experience, a pretty sure way to render the system inconsistent. A persistent experience of Nqthm users, many with deep previous training in mathematics, is the high likelihood of their making mistakes in the formulation of lemmas, mistakes that are caught in the attempt to have Nqthm check them. Therefore, the proofs we describe are based solely on the definitions of our MC68020 model, and not on any nondefinitional axioms added during the project.

Currently, our library of lemmas consists of approximately 1500 lemmas, about 120 pages of text. The full lemma library is presented in [30]. We give a few examples of typical lemmas below. Our experiments with the library, the topic of the next section, have been very satisfactory. Next, we briefly review some of the important issues we have dealt with in our development of the library.

4.1.1 Arithmetic

All the bit vector operations are defined with nonnegative integer arithmetic; hence theorems about bit vectors are merely theorems about nonnegative integer arithmetic. We have focused on reasoning about these operations: $x + y$, $x * y$, $x - y$, $x \textbf{ mod } y$, $x \div y$, and $\exp(x, y)$. During the development, we have been greatly benefited from an integer library developed at Computational Logic, Inc.

Due to the fixed size of operations at the machine-code level, it is inevitable that we study modulo arithmetic. Our purpose here is to establish a set of proof rules to support modulo arithmetic reasoning at a relatively high level. For example, recall that $\text{add}(x, y, n)$ is defined as $(x + y) \bmod 2^n$. One of the rules for modulo addition is associativity:

THEOREM: add-associativity
$\text{add}(n, \text{add}(n, x, y), z) = \text{add}(n, x, \text{add}(n, y, z))$

It is worth noting that some meta lemmas about modulo arithmetic have been proved by Nqthm and incorporated into our theory, which demonstrates the usefulness of the Nqthm's meta extension mechanism [3, 6]. This meta extension technique permits the proof of the correctness of simplification procedures, which once proved are then embedded into Nqthm's simplification machinery. A simple example of such a meta lemma is the statement that identical terms may be cancelled from opposite sides of an additive arithmetic equation. Such meta lemmas permit the user a degree of control over Nqthm's simplification procedure that can be difficult, if not possible, to achieve by the mere addition of ordinary lemmas.

4.1.2 Alternative Interpretations

In order to formalize the MC68020 microprocessor as accurately as possible, we followed Motorola's description of the MC68020 very literally while writing our formal specification. But it is sometimes the case that the descriptions Motorola provides are not the most useful mathematical characterizations of operations to use when it is time to prove the correctness of particular programs. An important type of lemma in our library is the kind which expresses in a more useful or intuitive fashion the semantics of an operation. For example, we have previously discussed the rather syntactic formulation of the changes to the condition codes given in the Motorola manual. The following lemma establishes, roughly speaking, that the carry bit after a subtraction instruction is set iff $y < x$. 'nat-to-uint' gives the unsigned integer interpretation of a bit vector, while 'nat-to-int' gives the signed integer interpretation.

THEOREM: sub-bcs&cc
$(\text{nat-rangep}(x, n) \land \text{nat-rangep}(y, n) \land (n \not\simeq 0))$
$\rightarrow \quad (\text{bcs}(\text{sub-c}(n, x, y))$
$\qquad = \quad$ **if** $\text{nat-to-uint}(y) < \text{nat-to-uint}(x)$ **then** 1
$\qquad\qquad$ **else** 0 **endif**)

In a similar vein, the following lemma characterizes in arithmetic terms (using exponentiation) the effects of arithmetic shifting, which is defined in the specification by "bit movement."

THEOREM: asl-int
$(\text{nat-rangep}(x, n) \land \text{int-rangep}(\text{nat-to-int}(x, n), n - s))$
$\rightarrow \quad (\text{nat-to-int}(\text{asl}(n, x, s), n) = \text{itimes}(\text{nat-to-int}(x, n), \exp(2, s)))$

Roughly, this lemma says that shifting x left s bits equals multiplying x by 2^s, if there is no overflow. 'int-rangep' returns **t**, if $((-\exp(2, n-1)) \leq \mathit{int}) \land (\mathit{int} < \exp(2, n-1))$, and returns **f**, otherwise. 'itimes' multiplies integers.

4.1.3 Machine State Management

Machine state management is probably the most difficult part of the library. It mainly concerns general theorems about the machine state and its components. In proofs of programs, machine states are the objects the theorem prover has to reason about and the user has to inspect when the proof fails. The machine state is often very complex and unmanageable. By developing carefully a set of lemmas for each of the components of the machine state, we are able to gain some level of abstraction that helps the theorem prover focus on the relevant part of the proof and helps the user understand the proof script, in particular, when the proof attempt fails.

For example, one of the lemmas about memory "tells" the prover how to read the byte at memory location x.

```
THEOREM: byte-read-write
 byte-read ($x$, byte-write ($v$, $y$, $mem$))
 =   if mod32-eq ($x$, $y$)
     then if nat-rangep ($v$, 8)  then fix ($v$)
          else head ($v$, 8) endif
     else byte-read ($x$, $mem$) endif
```

Roughly, this says that the result of reading at location x after writing v at location y is either v or the previous contents of x, according to whether x is equal to y or not.

4.2 Correctness Proofs

We turn now to the most interesting part of our project—the correctness proof of object code generated from programs written in higher-level languages. In this section, we will explain our approach with the correctness proof of some MC68020 object code that computes the greatest common divisor of two non-negative integers by Euclid's algorithm. One may find the complete script of our GCD proof in [30].

To obtain our object code, we start with the following C program.

```
int gcd(int a, int b)
{
  while (a != 0){
    if (b == 0) return (a);
    if (a > b)
      a = a % b;
    else b = b % a;
  };
  return (b);
}
```

We next run this C program through the Gnu C compiler, load the object code into memory, and use the Gnu debugger to obtain the object code both in symbolic format (for human consumption, only) and in numeric format (for Nqthm's consumption). The symbolic format is:

```
gcd:            linkw a6,#0
                moveml d2-d3,sp@-
                movel a6@(8),d2
                movel a6@(12),d3
gcd+16:         tstl d2
                beq 0x22f6 <gcd+48>
                tstl d3
                bne 0x22e2 <gcd+28>
                movel d2,d0
                bra 0x22f8 <gcd+50>
gcd+28:         cmpl d2,d3
                bge 0x22ee <gcd+40>
                divsll d3,d0,d2
                movel d0,d2
                bra 0x22d6 <gcd+16>
gcd+40:         divsll d2,d0,d3
                movel d0,d3
                bra 0x22d6 <gcd+16>
gcd+48:         movel d3,d0
gcd+50:         moveml a6@(-8),d2-d3
                unlk a6
                rts
```

The numeric format, expressed as a list of nonnegative integers, is given by this Nqthm function:

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

The above list of numbers (bytes) is the object subject to proof.

4.2.1 The Correctness Statement

The correctness statement for the foregoing GCD program should fully characterize the effects of the execution of the program on the machine state. The most important requirement of the correctness statement is that it be "context-free" and "universally" applicable so that we can reuse the theorems in the proofs of other programs that use this program as a subprogram. Our correctness theorem at the object code level is more elaborate than it would be for a program written in a higher-level language. This is not particularly surprising to us, since our proofs assert more properties about a program than would a higher-level specification.

In general, the correctness of object code in our formalism means:

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

In our example, the correctness of our GCD program is given by the following theorem, which formalizes exactly what we have described above.

THEOREM: gcd-correctness
let sn **be** stepn (s, gcd-t (a, b))
in
gcd-statep (s, a, b)
$\rightarrow$ ((mc-status (sn) = `'running`)
  $\wedge$ (mc-pc (sn) = rts-addr (s))
  $\wedge$ (read-rn (32, 14, mc-rfile (sn))
    = read-rn (32, 14, mc-rfile (s)))
  $\wedge$ (read-rn (32, 15, mc-rfile (sn))
    = add (32, read-an (32, 7, s), 4))
  $\wedge$ ((d2-7a2-5p (rn) $\wedge$ ($oplen \leq 32$))
    $\rightarrow$ (read-rn ($oplen$, rn, mc-rfile (sn))
      = read-rn ($oplen$, rn, mc-rfile (s))))
  $\wedge$ (disjoint (x, k, sub (32, 12, read-sp (s)), 24)
    $\rightarrow$ (read-mem (x, mc-mem (sn), k)
      = read-mem (x, mc-mem (s), k)))
  $\wedge$ (iread-dn (32, 0, sn) = gcd (a, b))) **endlet**

where gcd-statep (s, a, b), given below, is the hypothesis that specifies the assumptions on the initial state.

DEFINITION:
gcd-statep (s, a, b)

$$\begin{aligned}
= \quad & ((\text{mc-status}\,(s) = \texttt{'running}) \\
& \wedge \quad \text{evenp}\,(\text{mc-pc}\,(s)) \\
& \wedge \quad \text{rom-addrp}\,(\text{mc-pc}\,(s),\ \text{mc-mem}\,(s),\ 60) \\
& \wedge \quad \text{mcode-addrp}\,(\text{mc-pc}\,(s),\ \text{mc-mem}\,(s),\ \textsc{gcd-code}) \\
& \wedge \quad \text{ram-addrp}\,(\text{sub}\,(32,\ 12,\ \text{read-sp}\,(s)),\ \text{mc-mem}\,(s),\ 24) \\
& \wedge \quad (a = \text{iread-mem}\,(\text{add}\,(32,\ \text{read-sp}\,(s),\ 4),\ \text{mc-mem}\,(s),\ 4)) \\
& \wedge \quad (b = \text{iread-mem}\,(\text{add}\,(32,\ \text{read-sp}\,(s),\ 8),\ \text{mc-mem}\,(s),\ 4)) \\
& \wedge \quad (a \in \mathbf{N}) \\
& \wedge \quad (b \in \mathbf{N}))
\end{aligned}$$

gcd-statep $(s,\ a,\ b)$, roughly speaking, asserts:

- The machine state s is in the user mode.
- The program counter of s is even.
- The 60 consecutive bytes in the memory of s, starting from the address pointed to by the program counter of s, store the GCD program given above by GCD-CODE, in ROM.
- There are 24 bytes available on the stack.
- The integers a and b are on the stack, and both are nonnegative.

Informally, the theorem 'gcd-correctness' states that if s is as characterized by gcd-statep $(s,\ a,\ b)$, then there is an integer n, given by the expression gcd-t $(a,\ b)$, which tells us how many instructions to run the MC68020 starting with s before the GCD program returns, such that after running s for n steps the resulting state s' has these properties:

- s' is still `'running`, i.e., no errors occurred.
- The pc of s' points to the return address on the top of the stack in s.
- Register A6, which is used by the `LINK` instruction, is unchanged.
- Register A7, the stack pointer, has been incremented by 4.
- All of the registers of s' have the same values as those of s, except D0, D1, A0, A1, A6, and A7.
- Every memory location of s' has the same value as it did in s, except for the 12 bytes on either side of the stack pointer of s.
- Register D0 of s' contains gcd $(a,\ b)$.

The function 'gcd' is formally defined as a recursive function, patterned after Euclid's algorithm. Lemmas later proved about 'gcd' establish that it does in fact return the greatest common divisor of its two arguments.

The completeness of detail that is necessary when proving the correctness of object code programs is especially obvious when one verifies programs involving subroutine calls and recursions, such as we have done with Quick Sort.

4.2.2 The Proof

In our approach to the verification of specific object code programs, there are always two independent phases: one deals with the correctness of the underlying algorithm and the other deals with the correctness of its implementation. Success in separating the two issues and tackling each of them in isolation makes the correctness proof easier. Therefore, our correctness proofs are always divided into two steps:

1. We formalize the underlying algorithm as a function in the Nqthm logic and prove the equivalence of the algorithm with the result of running the MC68020 specification on the given object code. What we establish in this step is that the implementation does implement the algorithm. Note that this says nothing about the correctness of the algorithm.

2. We prove that the algorithm, formalized as an Nqthm function, is correct. Note here we do not need to deal with any MC68020 related specifics in this step. We can therefore focus completely on the mathematics of the algorithm, and fully enjoy many of the mathematical laws that are not available at the processor level.

Illustrating the first step. Thus, for the GCD example, we formalize Euclid's algorithm in Nqthm as follows:

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

The first step is to prove that the functional behavior of the object code is equivalent to the above function, which is one of the many properties proved by 'gcd-correctness'.

The work involved in this step depends on the complexity of the flow of the control of the program. For each loop in the program, we need to prove two intermediate lemmas that correspond to the base case and inductive case of an inductive proof. These intermediate lemmas are quite analogous to the verification conditions in proof by the Floyd method[11]. The lemmas differ from typical verification conditions largely because of very detailed attention to the number of machine instructions being executed. Since we do not consider the mathematical properties of the program here, the typical invariant proofs found in Floyd/Hoare method are delayed to the second step. For straight-line programs, there is no need to introduce any intermediate lemmas.

In our GCD example, the most crucial lemma towards the first step is stating and checking with Nqthm that we can "go around the loop." The following is part of this "go around the loop" lemma:

$$(\textit{gcd-s0p}\,(s,\ a,\ b) \wedge (a \not\simeq 0) \wedge (b \not\simeq 0) \wedge (b < a))$$
$$\Rightarrow\ \ \textit{gcd-s0p}\,(\textit{stepn}\,(s,\ 9),\ a\ \mathbf{mod}\ b,\ b)$$

which states that if the state s satisfies a certain invariant condition 'gcd-s0p' with respect to a and b, and if $a < b$, then after 9 machine instructions, the machine will be in a new state that satisfies the same invariant condition with respect to a mod b and b. gcd-s0p$\,(s,\ a,\ b)$ requires, among other things, that GCD-CODE be loaded in ROM memory starting at an appropriate place, namely 16 locations before the program counter of s, and that a and b be in registers 2 and 3 respectively. Given the extensive effort that was put into formulating the lemma library, and the consequent ability of the Nqthm simplifier to execute machine-code programs symbolically, it is generally easy to state and check such lemmas once one gets used to counting instructions and attending to registers and memory locations instead of symbolically named variables. So routine is the formulation and proving of the lemmas in this step of verification that we feel there is some hope for automation here, much as verification condition generation has been automated.

Illustrating the second step. The second step, in our GCD example, is to prove that the function 'gcd' does indeed compute the greatest common divisor of a and b. This is stated with the following two theorems:

- gcd$\,(a,\ b)$ is a common divisor of a and b.

  THEOREM: gcd-is-cd
  $$((a\ \mathbf{mod}\ \mathrm{gcd}\,(a,\ b)) = 0) \wedge ((b\ \mathbf{mod}\ \mathrm{gcd}\,(a,\ b)) = 0)$$

- gcd$\,(a,\ b)$ is the greatest, i.e., it is no less than any common divisor of a and b.

  THEOREM: gcd-the-greatest
  $$((a \not\simeq 0) \wedge (b \not\simeq 0) \wedge ((a\ \mathbf{mod}\ x) = 0) \wedge ((b\ \mathbf{mod}\ x) = 0))$$
  $$\rightarrow\ \ (\mathrm{gcd}\,(a,\ b) \not< x)$$

To get Nqthm to check such theorems, one must, in general, discover, state, and mechanically check a number of intermediate lemmas that capture all the key ideas in the underlying mathematics. For example, in the case of the proof of the immediately preceding theorem, we need the following intermediate lemma:

THEOREM: remainder-remainder
$$((b\ \mathbf{mod}\ c) = 0) \rightarrow (((a\ \mathbf{mod}\ b)\ \mathbf{mod}\ c) = (a\ \mathbf{mod}\ c))$$

This lemma in turn was dependent upon some previously proven results in our lemma library, such as,

THEOREM: remainder-times
$$(((x * y)\ \mathbf{mod}\ y) = 0) \wedge (((y * x)\ \mathbf{mod}\ y) = 0)$$

and

THEOREM: remainder-quotient-elim
$((y \not\simeq 0) \land (x \in \mathbf{N})) \rightarrow (((x \ \mathbf{mod}\ y) + (y * (x \div y))) = x)$

4.2.3 Timing Analysis for GCD

The function 'gcd-t', which was used above in the theorem 'gcd-correctness', returns the exact number of MC68020 instructions executed by the GCD program. The definition of 'gcd-t' is:

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

DEFINITION: $\text{gcd-t}(a, b) = (4 + \text{gcd-t1}(a, b))$

Using the definition of 'gcd-t', we have mechanically proved that the number of instructions executed by the GCD program is at most 598.

THEOREM: gcd-t-ubound
$((a < \exp(2, 31)) \land (b < \exp(2, 31))) \rightarrow (\text{gcd-t}(a, b) \leq 580)$

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

5 The Results

Using the techniques described here, we have managed to verify mechanically the object code produced by the Gnu C compiler for hundreds of lines of C, including some of the C code in Kernighan and Ritchie's book [17], in particular binary search and Quick Sort.

We have also tried to verify the Berkeley Unix C string library. Twenty-one functions out of twenty-two functions specified in the ISO standard [29, 16] have been mechanically verified. The function `strerror`, though mathematically trivial, is the only one left out because of the need to formalize IO, which we have not treated in our work. In the process of verifying the Berkeley C string library, three C programming errors were revealed. Two were in the Berkeley Unix C string library. One error was undetected when we reported it to the author [27], and was to have been corrected for the release of BSD4.4. The second error had already been fixed by the author [27] when we reported it. A third error was in Plauger's book *The Standard C Library* [24]. This error had been detected by the author by the time we reported it to him [23]. These errors and all the C string library proofs are discussed in detail in Yuan Yu's dissertation [30].

Primarily to provide concrete evidence that this work is easily applicable to many other languages other than C, we have also mechanically verified the object code produced by the Verdix Ada compiler for an integer square root algorithm. Furthermore, we have mechanically verified the object code produced by the Gnu Common Lisp compiler for a "fixnum" GCD program.

Via a few examples, we have also shown how it is possible to check mechanically the correctness of machine-code programs that call other, previously verified, machine-code programs as subroutines. Our verification of Quick Sort, for example, illustrates subroutine calling, even recursion. Another example is the strstr function in the Berkeley Unix C string library which calls the strlen and strncmp functions. Handling subroutines raises such obvious issues as the correct passage of the return program counter on the stack and the location of the code for the called subroutine, both where it is located in memory and how that location is indicated by the calling subroutine. In addition to subroutines, we have studied some other interesting C language issues in the context of machine-code programs for the MC68020, such as functional parameters, computed jumps, and embedded assembly code. For a detailed discussion of these issues, we refer interested readers to Chapter 6 of [30].

6 Future Directions

As a next step, we plan to apply the mathematics so developed to another computer architecture, say, Alpha or Sparc. We believe we can do it but dealing with the nondeterminism introduced via instructions such as "delayed branch," which may leave the program counter in an indeterminate state during some instructions, will be a challenge.

A greater challenge is to specify and prove programs involving supervisor mode, especially interrupt handling.

The formal specification we have already developed allows us to investigate:

- The correctness of some moderate sized piece of software that is in critical use. One good example is the verification of microcontroller programs, an important subject that has been largely ignored by the formal verification community. The MC68332 microcontroller contains a processor with an instruction set quite close to that of the MC68020.

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

- The correctness of high-level programming language compilers. Even though compiler verification may have little practical impact in the near future, it is a research area with many interesting problems.
- The correctness of some lower level software, e.g., software for cache and memory management.

We believe that success in any of these directions would be a major contribution to formal methods. One of the authors has applied the same approach to prove properties of Alpha PALcode (Privileged Architecture Library)[26].

7 Related Work

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

Our work has built on the work of many others. Of historic interest is the early work of Turing [28] and Goldstine and von Neumann [12]. The careful proof of machine-code programs is coincident with the foundations of the von Neumann machine, first presented in [12]. In those classic papers of von Neumann and Goldstine, we find discussed the specification and correctness proofs for fifteen programs at the machine-code level. Proving the correctness of programs written in the assembly language MIX is a main feature of Knuth's *magnum opus* [18]. The informal, hand proof of object code produced by "industrial strength" compilers is not unheard of today for especially critical programs.

Methods for program proving have been advanced most notably by McCarthy [21], Floyd [11], and Hoare [13]. In the last twenty years, many research projects have investigated the formal, mechanical verification of programs written in higher-level languages. One can mention as examples the initial work of Floyd and King; London and Musser's group at ISI; Luckham's Pascal and Ana groups at Stanford; Good's Gypsy group at the University of Texas and Computational Logic; Milner and Gordon's LCF and HOL groups in Stanford, Edinburgh, and Cambridge; Huet's Coq group at INRIA; Craigen and Pase's Never group at ORA; Burstall and Plotkin's group in Edinburgh; Constable's NuPrl group at Cornell; and several projects led by Levitt, Neumann, and Rushby at SRI. Most of these projects are based on Floyd's inductive assertion method. For a survey of many of these projects, see [4]. Our work differs from this previous work in that we address the correctness of programs at the machine-code level executed on a widely used processor.

In only a very few cases, however, does research on formal, mechanical software correctness address the machine-code level for actually fabricated processors. To the best of our knowledge, Maurer [20, 19] was the first to consider the verification, with an automated reasoning system, of machine-code programs for a fabricated microprocessor. Subsequently, Clutterbuck and Carré[8] argued for the importance of the verification of low-level code, and, in a separate paper [15], reported their effort to analyze and verify the LUCOL assembly code modules used in the fuel control unit of the Rolls-Royce RB211-524G jet engine designed for Boeing 747-400. Like most work on software verification, this work is based upon the use of a Floyd-style verification condition generator. The problem of assembler correctness was not addressed in their work. Since the semantics of assembly language is normally rather complicated,[8] many restrictions had to be imposed on the assembly language, and complex annotations had to be inserted into the programs being verified.

Our work, in contrast, is based on an explicit, formal, operational semantics, i.e., a definition in the logic of the automated reasoning system being used. One advantage of such a semantics is that it is not restricted to considering only programs that can be analyzed into preestablished patterns of loops or variables to help the verification condition generator produce conjectures to verify. Our approach can be used to address the correctness of *any* machine-code program that uses only instructions in the subset defined by our formal model. Our proofs are completely based on this formal model. Simplicity greatly increases our confidence in our formal models and formal proofs.

The first example we know of formal, mechanical verification of binary programs based on such an operational semantics for a von Neumann machine is the work of Bevier [1], also reported in [2]. In proving the correctness of a small operating system kernel, Bevier proves the correctness of several hundred lines of machine code produced by his own assembler for a von Neumann style machine of his own design.

In contrast to our approach to machine-code proof, compiler verification attempts to establish the correctness of the compiler, so that we are ensured that the compiler always generates correct binary code. Polak's work [25] seems the most ambitious compiler verification effort. Polak mechanically verified a compiler for a fairly substantial subset of Pascal. Moore's Piton and Young's Micro-Gypsy [2], two components of the CLI short stack, are major compiler verification efforts targeted on a more realistic von Neumann architecture—the verified and fabricated FM9001.

Even with such fairly encouraging results, it seems to us that the formal, mechanical verification of 'industrial strength' compilers may be a considerable distance off into the future because of the sheer complexity of those compilers.

Microcode verification is closely related to our work. Among the most significant reported work is the C/30 microcode verification using the State Delta

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

Verification System(SDVS) [10]. A large majority of the C/30's instructions were proven to be correctly implemented by approximately 1000 MBB microinstructions.

Microcode verification is often part of some larger hardware verification effort. Hunt [14] and Cohn [9] are two major hardware design verification projects involving microcode verification. Some of our techniques developed here can be applied to microcode verification.

8 Remarks

It has been asked "Why not dispense with the high-level language implementation and code directly in assembler?" Certainly, we can use, and have used, our approach to verify programs coded directly in machine code. Some examples are presented in [30]. However, since almost all programs are now written in higher-level languages, we believe it is useful to demonstrate that it is feasible, at least in some instances, to address directly the correctness of the actual "bits" generated by "industrial strength" compilers without first obtaining both a formalization of the semantics of the higher-level language and a proof of the correctness of a serious compiler for such a language. Given that the informal specification of C is book length, and given that the sources for, say, the Gnu C Compiler exceeds 20 megabytes, such compiler proofs can be rather expensive.

It might also be asked whether a compiler proof for a higher-level language would make verification of "bits" easier. It is certainly the case that the verification of algorithms written in higher-level languages, using the Floyd [11] approach, is somewhat easier than the method we have described because in the Floyd approach one is relieved of attention to such details as instruction counting, the identification of variables with specific machine locations, and, especially, the layout of code in memory. However, our experience is that it is not greatly more expensive, in the verification of a specific algorithm, to address the "bits" directly as opposed to treating the algorithm as coded in a higher-level language. The hard work in either case is similar: writing the formal specifications, finding the key loop invariants, and checking that the invariants are indeed preserved during execution. Furthermore, by addressing machine code directly, one can obtain exact instruction-count and stack-utilization information, which have relevance to real-time and real-space analysis, issues quite important for critical software.

Certainly, the specific compiler used may affect the intellectual "distance" between the expression of an algorithm in a higher-level language and the machine-code representation. A very clever compiler could, in principle, make verification with our approach very difficult. However, in the examples we have studied, we have found that even when using the highest optimization settings, the correspondence between the high-level expression and the machine code was sufficiently obvious as not to be a source of difficulty.

9 Conclusions

We believe that the work reported here is the first instance of the formal verification, by an automated reasoning system, of the binary code for software produced by "industrial strength" higher-level compilers targeting a widely used microprocessor whose semantics was formalized with an operational semantics in the logic of the reasoning system used.

We are optimistic that this verification technique can be applied to many programs on many different microprocessors and for many different higher-level language compilers.

In our opinion, the major scientific obstacle to the formal verification of small, single-process programs is obtaining formal specifications of what a given program is supposed to do.

One important lesson we learned, or confirmed, is that formal specifications should be developed together with their intended applications because the form and details of a specification so influence the resulting proof obligations. Our MC68020 specification has been greatly influenced by the need for reasoning about machine-code programs.

Building our library of lemmas consumed most of our time in this endeavor—many months of work. However, given the previous development of the lemma library, it would take one of us about two to three hours to complete a proof such as the GCD example presented in this paper, assuming that we start with a complete understanding of the correctness of the underlying informal algorithm. This time estimate has been repeatedly confirmed by experience in verifying small examples similar in complexity to our GCD example. As a general rule of thumb, it seems that a very highly skilled Nqthm user, given a clear understanding of a proof in elementary, discrete mathematics, can get Nqthm to check the proof in a period of time no greater than ten times the amount of time it takes to write up such a proof at the level of an advanced undergraduate mathematics textbook.

The library is still under development as we increase our ability to handle more and more programming language constructs and data types and other microprocessors. So far, we have dealt with many language features in our proofs:

- various data types: several sizes of signed and unsigned integers, boolean, characters, strings, arrays, structures, and static variables.
- subroutine: macro, recursion, standard subroutine calling, and functional parameters.
- various control structures: if-then-else, loop, goto, and case.
- pointer manipulation.
- higher-level language programs with embedded assembly code.

In pursuit of the objective of ensuring that any change to our collection of lemmas is indeed an improvement, we have fruitfully followed the *proveall* discipline described in [6], i.e., the practice of making sure that after we make changes, we can still prove the most important of our previous results.

10 Acknowledgments

We would like to thank Bill Bevier, Paul Eggert, Don Good, Warren Hunt, Matt Kaufmann, J Moore, and Bill Schelter for their many constructive suggestions and discussions. Special thanks to Fay Goytowski Saydjari for her meticulous reading of the MC68020 specification, which revealed a dozen or so errors. Further special thanks to Kenneth Albin, for his extensive testing of the specification against an actual MC68020 chip, which also revealed about a dozen errors in our specification. The general style of Nqthm formalization used in this MC68020 specification is the product of over a decade of study by the authors of Nqthm and their students. Especially influential was the FM8502 and Piton work [2]. The development of Nqthm was primarily supported by NSF, ONR, and DARPA.

The sources for Nqthm together with the formal text for our MC68020 specification and all theorems mentioned are available in machine readable form via the Internet as either ftp://ftp.cli.com/pub/nqthm/nqthm-1992/nqthm-1992.tar.Z or the file ftp://ftp.cs.utexas.edu/pub/boyer/nqthm-1992.tar.Z, or by asking the authors.

Submitted in May, 1992. First reviews received in December, 1994. Revision submitted in February 1995.

References

[1] William Bevier. *A Verified Operating System Kernel.* PhD thesis, University of Texas at Austin, 1987.

[2] William Bevier, Warren Hunt, J Strother Moore, and William Young. Special issue on system verification. *Journal of Automated Reasoning*, 5(4), 1989.

[3] R. S. Boyer and J S. Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. In R. S. Boyer and J S. Moore, editors, *The Correctness Problem in Computer Science*, pages 103–184. Academic Press, London, 1981.

[4] Robert S. Boyer and J S. Moore. Program verification. *Journal of Automated Reasoning*, 1(1):17–23, 1985.

[5] Robert S. Boyer and J Strother Moore. *A Computational Logic.* Academic Press, New York, 1979.

[6] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook.* Academic Press, 1988. For sources and examples, see ftp://ftp.cli.com/pub/nqthm/nqthm-1992/nqthm-1992.tar.Z or ftp://ftp.cs.utexas.edu/pub/boyer/nqthm-1992.tar.Z.

[7] Robert S. Boyer and Yuan Yu. A formal specification of some user mode instructions for the Motorola 68020. Technical Report TR-92-04, Computer Sciences Department, University of Texas at Austin, 1992. See ftp://ftp.cs.utexas.edu/pub/techreports/tr92-04.ps.Z. Alternatively, see examples/yu/mc20-1.ps in ftp://ftp.cs.utexas.edu/pub/boyer/nqthm-1992.tar.Z.

[8] D.L. Clutterbuck and B.A. Carré. The verification of low-level code. *IEE Software Engineering Journal*, May 1988.

[9] Avra Cohn. A proof of correctness of the Viper microprocessor: The first level. Technical Report 104, University of Cambridge, January 1987.

[10] J. V. Cook. Verification of the C/30 microcode using the State Delta Verification System (SDVS). In *13th National Computer Security Conference*, volume 1, pages 20–31, 1990.

[11] Robert W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics, American Mathematical Society*, pages 19–32, Providence, Rhode Island, 1967.

[12] Herman H. Goldstine and John von Neumann. Planning and coding problems for an electronic computing instrument. In *John von Neumann, Collected Works*, volume V, pages 34–235. Pergamon Press, Oxford, 1961.

[13] C.A.R. Hoare. An axiomatic basis for computer programming. *The Communication of ACM*, 12(10):576–583, 1969.

[14] Warren A. Hunt. *FM8501: A Verified Microprocessor.* PhD thesis, University of Texas at Austin, 1985.

[15] I.M. O'Neill, et al. The formal verification of safety-critical assembly code. In *Safety of Computer Control System 1988.* Pergamon Press, November 1988.

[16] ISO Committee JTC1/SC22/WG14. *ISO/IEC Standard 9899:1990.* International Standards Organization, Geneva, 1990.

[17] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition.* Prentice Hall, Englewood Cliff, New Jersey, 1988.

[18] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, Reading, Massachusetts, 1981.

[19] W. D. Maurer. An IBM 370 assembly language verifier. In *Proceedings of the 16th Annual Technical Symposium on Systems and Software: Operational Reliability and Performance Assurance.* ACM, June 1974.

[20] W. D. Maurer. Some correctness principles for machine language program and microprocessors. In *Proceedings of the Seventh Annual Workshop on Microprogramming*, Palo Alto, CA, 1974.

[21] John McCarthy. Towards a mathematical science of computation. In *Proceedings of IFIP Congress*, pages 21–28, 1962.

[22] Motorola, Inc. *MC68020 32-bit Microprocessor User's Manual.* Prentice Hall, New Jersey, 1989.

[23] P. J. Plauger. Private communication.

[24] P. J. Plauger. *The Standard C Library.* Prentice Hall, New Jersey, 1992.

[25] Wolfgang Polak. *Compiler Specification and Verification.* Springer-Verlag, Berlin, 1981.

[26] Richard L. Sites. *Alpha Architecture Reference Manual.* Digital Press, Bedford, Mass., 1992.

[27] Chris Torek. Private communication.

[28] Alan M. Turing. On checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69. Univ. Math. Laboratory, Cambridge, 1949.

[29] The ANSI Committee X3J11. *ANSI Standard X3.159-1989.* American National Standards Institute, New York, 1989.

[30] Yuan Yu. *Automated Proofs of Object Code For a Widely Used Microprocessor.* PhD thesis, University of Texas at Austin, 1992. For the dissertation text, see ftp://ftp.cs.utexas.edu/pub/techreports/tr93-09.ps.Z. See the files ./examples/yu/* in ftp://ftp.cs.utexas.edu/pub/boyer/nqthm-1992.tar.Z for replayable proof scripts of all the definitions and theorems mentioned in this paper and in the thesis.