Automated Correctness Proofs of Machine Code Programs for a Commercial Microprocessor[1]

Robert S. Boyer and *Yuan Yu*

Computer Sciences and Mathematics Departments
University of Texas at Austin
Austin, Texas 78712
telephone: (512) 471-9745
email: boyer@cs.utexas.edu or yuan@cs.utexas.edu

Abstract. We have formally specified a substantial subset of the MC68020, a widely used microprocessor built by Motorola, within the mathematical logic of the automated reasoning system Nqthm, i.e., the Boyer-Moore Theorem Prover [6]. Using this MC68020 specification, we have mechanically checked the correctness of MC68020 machine code programs for Euclid's GCD, Hoare's Quick Sort, binary search, and other well-known algorithms. The machine code for these examples was generated using the Gnu C and the Verdix Ada compilers. We have developed an extensive library of proven lemmas to facilitate automated reasoning about machine code programs. We describe a two stage methodology we use to do our machine code proofs.

Key words. Automated reasoning, Nqthm, Boyer-Moore Theorem Prover, formal program verification, object code, Gnu, C, Ada.

1 Introduction

One of the main reasons for our lack of confidence in computing systems is the lack of mathematical theories to forecast accurately the behavior of computing systems. The idea of providing a rigorous mathematical basis for programming dates back to the very beginning of computing. In the classic papers of von Neumann and Goldstine [12], which introduced the first "von Neumann machine," they described how to prove the correctness of machine code programs. Fifteen machine code programs are there specified, coded, and proved correct. Later, methods for proving the correctness of programs written in higher level programming languages were put forward by McCarthy, Floyd, Hoare and others.

A formal program verification is a mathematical proof that a program executed according to a certain model of computation meets some specification. Such a proof involves a formal specification of the computing model on which the program is intended to execute and a formal reasoning system on which the proof is based. Once a program is formally verified, people may place a very high degree of confidence in the correctness of the program. Because correctness proofs can be extremely tedious, it seems to be difficult for humans to check (most importantly, correctly) all the proof details. Automated reasoning systems have been successfully used to

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

assist humans to check the correctness of some computer programs (see, for example, the survey paper [4] and the more recent, large scale efforts [2]).

We have recently used the automated reasoning system Nqthm [6] to define formally a mathematical specification for the widely used Motorola MC68020 microprocessor and to verify mechanically the correctness of machine code generated by high-level programming language compilers for that microprocessor. We believe our work differs from almost all the other research in formal program verification, which almost exclusively has focused on proving the correctness of programs written in high level programming languages, and has not undertaken the goal of assuring that the software correctly executes upon hardware that is commonly used. In only a very few cases does research on formal program correctness come close to the hardware level, and, as far as we are aware, when it does, the hardware is either "on paper" or very novel. For earlier examples of Nqthm-checked program proofs that are rooted in an interpreter semantics approach, see the exemplary work of Bevier (on Kit), Hunt (on FM8502), Moore (on Piton), and Young (on Micro-Gypsy), all described in [2]. For related work, see also [10].

There are several motivations that led us to study program verification at the machine code level.

- Formal specification and verification at the processor level, e.g., for a compiler correctness proof, is ultimately a necessary ingredient in formal correctness proofs of programs if we take as our goal ensuring that programs are executed correctly on a particular processor.

- Some of the most sensitive programs in the world are currently "verified" at the machine code level anyway, even though written in higher level programming languages—compilers are simply not trusted by those to whom security matters most.

  - Many high level programming languages, such as those typically used in industrial practice, are not precisely specified. It is not easy, or even possible, to give the semantics of some programming features. For example, the `volatile` construct in C.
  - Some "industrial strength" compilers produce erroneous code. But given the sheer size of these compilers, it is unlikely that any will be proved correct in the foreseeable future.

- Programs written in high level languages may need to have assembly code embedded in them in order to communicate with the outside world. No high level language specification can make clear the semantics of the embedded assembly instructions. Furthermore, any sort of real-time analysis is (currently) best done at the machine code level.

It is our belief that success in verifying machine code programs will be a step towards incorporating formal verification into the "real" world of programming.

To familiarize the reader with the features of Nqthm, Section 2 is dedicated to a brief overview of the Nqthm logic and its theorem prover. In Section 3, we briefly describe our formal specification of the user instruction set of the MC68020. For more details of the specification, we refer the reader to the complete documentation of our specification in [7]. In Section 4, we discuss how we prove the correctness of object code generated from C and Ada programs.

2 The Automated Reasoning System Nqthm

The automated reasoning system Nqthm, also known as "the Boyer-Moore Theorem Prover," is a Common Lisp program for proving mathematical theorems. In the twelve years since *A Computational Logic* [5] was published, Nqthm has been used to specify and check the correctness of numerous theorems from many areas of mathematics and computing. An extensive partial listing may be found in [6, pages 5–9]. See also [2]. This section contains an extremely brief overview of Nqthm that may suffice for understanding the rest of this paper. 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.

The logic of Nqthm is a quantifier-free first order logic with equality. The syntax is similar to that of Pure Lisp or the lambda calculus, e.g., prefix notation with parentheses. Constants in the logic are functions with no arguments. The basic theory includes axioms defining (1) the Boolean constants `(TRUE)` and `(FALSE)`, abbreviated as `T` and `F`, respectively; (2) the equality function `(EQUAL X Y)`, axiomatized to return `T` or `F` according to whether `X` is `Y`; (3) the if-then-else function `(IF X Y Z)`, axiomatized to return `Z` if `X` is `F` and `Y` otherwise; and (4) the Boolean arithmetic functions `AND`, `OR`, `NOT`, `IMPLIES`, and `IFF`.

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 integers, ordered pairs, e.g., `(cons 1 2)`, and symbols, e.g., `'running`, are axiomatized in the logic by adding shells. The *Definition Principle* allows the user to define new functions in the logic. Among the functions in the logic are `PLUS`, `DIFFERENCE`, `LESSP`, `TIMES`, `QUOTIENT`, and `REMAINDER`, which correspond to $+, -, <, *, /$, and mod, respectively.

Nqthm is a mechanization of this 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 commands to the theorem prover include those for defining new functions, proving lemmas, and adding shells, etc. Two commands are the most often used. (1) `(defn` *fn-name* `(`*arg1* ... *argn*`)` *body*`)` defines a new function *fn-name* with n arguments *arg1*,..., *argn* and definition *body*. (2) `(prove-lemma` *lemma-name* `(`*lemma-type*`)` *statement*`)` initiates a proof attempt for conjecture *statement*. If the proof succeeds, the lemma is stored as a rule with type *lemma-type*, under the given name *lemma-name*. The behavior of the prover is influenced profoundly by the proof of appropriate lemmas.

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]. 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 a 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. 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 arcane 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. We summarize below those instructions formalized.

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

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`

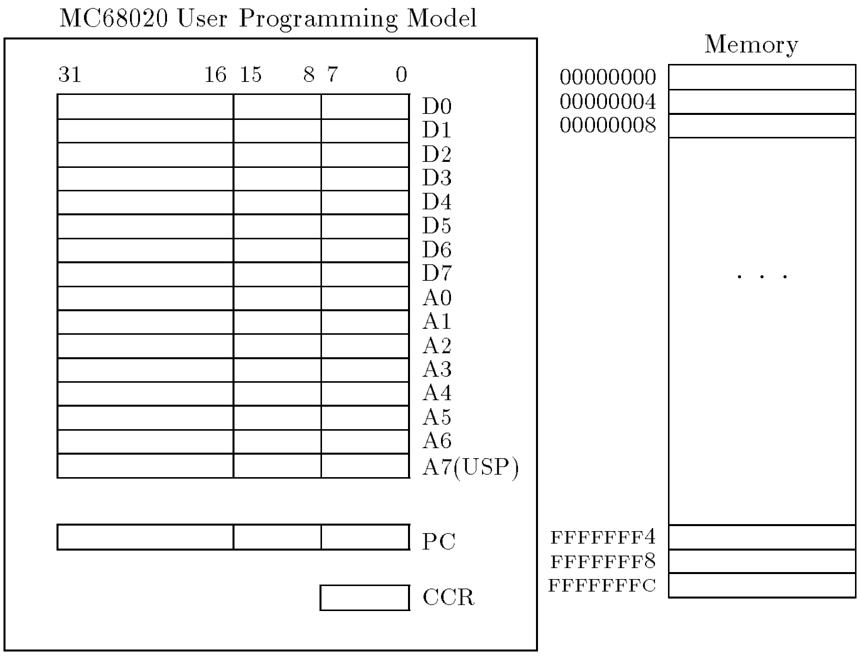

Figure 1: The User Visible Machine State

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 value 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 some sort of operation on nonnegative integers. Here are a few basic operations on bit vectors and their definitions in the Nqthm logic.

```
(defn head (x n)
  (remainder x (exp 2 n)))

(defn tail (x n)
  (quotient x (exp 2 n)))

(defn bits (x i j)
  (head (tail x i) (plus 1 (difference j i))))

(defn app (n x y)
  (plus (head x n) (times y (exp 2 n))))
```

Intuitively, `head` returns the bit vector of the first `n` bits of `x`; `tail` returns the bit vector obtained by discarding the first `n` bits of `x`; `bits` returns the bit vector consisting of bits `i` through `j` of `x`; `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 content 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].

`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.

3.2.2 The Specification

The top-level loop of our specification is defined by a pair of functions, the "single-stepper" `stepi` and the "stepper" `stepn`:

```
(defn stepi (s)                    ; the single stepper.
  (if (evenp (mc-pc s))
      (if (pc-word-readp (mc-pc s) (mc-mem s))
          (execute-ins (current-ins (mc-pc s) s)
                       (update-pc (add (l) (mc-pc s) (wsz)) s))
        (halt (pc-signal) s))
    (halt (pc-odd-signal) s)))

(defn stepn (s n)                  ; executes n instructions.
  (if (or (mc-haltp s) (zerop n))
      s
    (stepn (stepi s) (sub1 n))))
```

`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. `stepn` simply executes `n` instructions by calling the single stepper `stepi`.

Roughly speaking, `execute-ins` decodes the current instruction according to the opcode and jumps to the specification of the instruction identified. We formalize each individual instruction with one main function and a few auxiliary functions, which calculate the effective addresses, fetch the operands, perform the specific operation, update the condition codes, and store the results.

Altogether, our formal specification is about 128,000 bytes long, which takes up approximately 80 pages of text. It consists of 569 function definitions. About two thirds of the specification is devoted to the formalization of individual instructions.

In addition to using Nqthm 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, 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!)
- 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.

Rather than describing the details of any particular instruction specification, 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, and 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:

```
(defn sub-c (n sopd dopd)
  (let ((result (sub n sopd dopd)))
    (b-or (b-or (b-and (bitn sopd (sub1 n))
                       (b-not (bitn dopd (sub1 n))))
                (b-and (bitn result (sub1 n))
                       (b-not (bitn dopd (sub1 n)))))
          (b-and (bitn sopd (sub1 n)) (bitn result (sub1 n))))))
```

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 user 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 pain to formalize completely.

4 Machine Code Verification

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

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 machine code programs which will have utility in the verification of many different programs.

We have invested a vast amount of time creating our lemma database, probably the most time spent in the entire project. Currently, our library of lemmas is about $180,000$ bytes, or 100 pages, long. 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 functions: `plus`, `difference`, `times`, `remainder`, `quotient` and `exp`. 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 MC68020 machine 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, addition is defined here as $(x + y) \bmod 2^n$:

```
(defn add (n x y) (head (plus x y) n))
```

One of the rules for modulo addition is associativity:

```
(prove-lemma add-associativity (rewrite)
     (equal (add n (add n x y) z) (add n x (add n y z))))
```

4.1.2 Alternative Interpretations

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 machine code programs. An important type of lemma in our library is the kind which expresses in a more useful or intuitive fashion the semantics of a machine code 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 $y < x$.

```
(prove-lemma sub-bcs&cc (rewrite)
    (implies (and (nat-rangep x n)
                  (nat-rangep y n)
                  (not (zerop n)))
             (equal (bcs (sub-c n x y))
                    (if (lessp (nat-view y) (nat-view x))
                         1 0))))
```

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." Roughly, this lemma says that shifting x left s bits equals multiplying x by 2^s, if there is no overflow.

```
(prove-lemma asl-int (rewrite)
     (implies (and (nat-rangep x n)
                   (int-rangep (nat-to-int x n) (difference n s)))
              (equal (nat-to-int (asl n x s) n)
                     (itimes (nat-to-int x n) (exp 2 s)))))
```

4.1.3 Memory Management

Memory management is probably the most difficult part of the library. It mainly concerns general theorems about the machine state and its components. For example, the following lemma "tells" the prover how to read the byte at memory location x.

```
(prove-lemma byte-read-write (rewrite)
     (equal (byte-read x (byte-write v y mem))
```

```
        (if (mod32-eq x y)
            (if (nat-rangep v 8) (fix v) (head v 8))
          (byte-read x mem))))
```

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 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 higher level languages. In this section, we will explain our approach with the correctness proof of a machine code program that computes the greatest common divisor of two nonnegative integers by Euclid's algorithm. To obtain our machine code program, 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 machine 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:

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

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. In our formalism, the correctness of a machine code program 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 three theorems, which formalize exactly what we have described above.

```
(prove-lemma gcd-correctness (rewrite)
  (implies (gcd-statep s a b)
           (and (equal (mc-status (stepn s (gcd-t a b))) 'running)
                (equal (mc-pc (stepn s (gcd-t a b))) (rts-addr s))
                (equal (iread-dn 32 0 (stepn s (gcd-t a b))) (gcd a b))
                (equal (read-an 32 6 (stepn s (gcd-t a b)))
                       (read-an 32 6 s))
                (equal (read-an 32 7 (stepn s (gcd-t a b)))
```

```
                      (add 32 (read-an 32 7 s) 4)))))

(prove-lemma gcd-rfile (rewrite)
   (implies (and (gcd-statep s a b)
                 (d2-7a2-5p rn)
                 (leq oplen 32))
            (equal (read-rn oplen rn (mc-rfile (stepn s (gcd-t a b))))
                   (read-rn oplen rn (mc-rfile s)))))

(prove-lemma gcd-mem (rewrite)
   (implies (and (gcd-statep s a b)
                 (disjoint x k (sub 32 12 (read-sp s)) 24))
            (equal (read-mem x (mc-mem (stepn s (gcd-t a b))) k)
                   (read-mem x (mc-mem s) k))))
```

`(gcd-statep s a b)`, which is the hypothesis that specifies the assumptions on the initial state, asserts, roughly speaking:

- 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)`.
- There is "enough" space 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 D0 of s' contains `(gcd a b)`, the greatest common divisor of a and b.
- Register A6, which is used by the LINK instruction, is unchanged.
- Register A7, the stack pointer, has been incremented by 4.

The theorem `gcd-rfile` further asserts that all of the registers of s' have the same values as those of s, except D0, D1, A0, A1, A6, and A7. Finally, the theorem `gcd-mem` asserts that 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. (`read-rn` and `read-mem` are the primary functions for reading the contents of the registers and memory.)

The completeness of detail that is necessary when proving the correctness of machine code programs is especially obvious when one verifies recursive machine code programs, such as we have done with Quick Sort.

4.2.2 The Proof

In our approach to the verification of specific machine 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 machine 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 we do not need to deal with any MC68020 related specifics in this step. So, we can focus completely on the mathematical properties of the algorithm.

Thus, for the previous example, we formalize the Euclid's algorithm in Nqthm as follows:

```
(defn gcd (a b)
  (if (zerop a) (fix b)
  (if (zerop b) a
  (if (lessp b a) (gcd (remainder a b) b)
                  (gcd a (remainder b a))))))
```

The first step is to prove that the functional behavior of the machine code is equivalent to the above function. The second step is to prove that the above function does indeed compute the greatest common divisor of a and b, which is proved by the following two theorems:

```
; (gcd a b) is a common divisor of a and b.
(prove-lemma gcd-is-cd (rewrite)
             (and (equal (remainder a (gcd a b)) 0)
                  (equal (remainder b (gcd a b)) 0)))

; (gcd a b) is the greatest, i.e., it divides any common divisor of a
; and b.
(prove-lemma gcd-the-greatest (rewrite)
             (implies (and (not (zerop a))
                           (not (zerop b))
                           (equal (remainder a x) 0)
                           (equal (remainder b x) 0))
                      (not (lessp (gcd a b) x))))
```

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:

```
(defn gcd-t1 (a b)
  (if (zerop a) 6
  (if (zerop b) 9
  (if (lessp b a) (plus 9 (gcd-t1 (remainder a b) b))
                  (plus 9 (gcd-t1 a (remainder b a)))))))

(defn gcd-t (a b) (plus 4 (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.

To study the real-time bounds of programs, we need to incorporate time information for each individual instruction, which seems to us a quite natural extension to our specification.

5 Conclusions

Our experience with machine code verification has been encouraging. We have managed to verify, with Nqthm, the object code produced by the Gnu C compiler for some of the C functions in Kernighan and Ritchie's book [17], such as binary search, Quick Sort, and also some C library functions, e.g., strlen and strcpy. We have also mechanically verified the object code produced by the Verdix Ada compiler for an integer square root algorithm. This success leads us to believe that it is completely feasible to verify the object code of moderate pieces of software in a reasonable time span. To scale up, some automated tools must be developed to assist the user in the proof. At the present time, we have not implemented any such tools. But we have been considering the implementation of something like a verification condition generator for machine code reasoning.

Our MC68020 specification has been greatly influenced by the need for reasoning about machine code. This consideration has complicated, to some extent, our specification task.

Building the library of lemmas consumes most of our energy. It is still under development as we are increasing our ability to handle more and more programming language constructs and data types. It is interesting to see whether we can apply the mathematics so developed to another computer architecture, say, RISC; we believe we can. To ensure that any change is indeed an improvement, we have followed the "proveall" discipline described in [6], i.e., the practice of making sure that after we make changes to our specification or lemmas, we can still prove the most important of our previous results.

Much can be said about future directions for such research. To specify supervisor mode, especially interrupt handling, is both desirable and challenging, but we have not yet attended to it. The formal specification we have already developed will let us investigate: (a) the correctness of some moderate sized piece of software that is in critical use; (b) the real-time execution bounds of programs; (c) the correctness of high level programming language compilers; and (d) 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 reasoning.

It can be expected that the full details of this project, including the complete MC68020 specification, lemma library, and programs proved will appear in subsequent publications, for example in the Ph. D. dissertation of Yuan Yu.

6 Acknowledgements

We would like to thank Bill Bevier, Don Good, Warren Hunt, Matt Kaufmann, J Moore, and Bill Schelter for their many constructive suggestions and discussions. Special thanks to Fay Goytowski for her meticulous reading of the MC68020 specification, which revealed a dozen or so errors. 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.

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.