114

Automated Proofs of Object Code For a Widely Used Microprocessor

Yuan Yu

October 5, 1993

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Automated Proofs of Object Code For a Widely Used Microprocessor

Yuan Yu

October 5, 1993

Publication History
An earlier version appeared as technical report TR-93-09 from the Department of Computer Sciences of the University of Texas at Austin.

All the C code, taken from the Berkeley C String library, presented in chapter 7 is subject to the following copyright terms.

Author's Abstract

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

Contents

Chapter 1

Introduction

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

Correctness proofs can be extremely large and tedious, and it is therefore difficult for humans to check all the proof details, and ensure that they are correct. To reduce the chance of mistakes in such proofs, the idea of mechanically proving the correctness of computer programs has been extensively studied (See the survey in [8]). It seems possible that the use of formal, mathematical, mechanical methods for ensuring the reliability of computing systems will eventually be required in safety critical applications [41]. While there has been a large body of research on program proving, work has almost exclusively focused on programs written in high-level programming languages. The problem investigated here is the feasibility of mechanically verifying machine-code programs executed upon existing and widely used hardware.

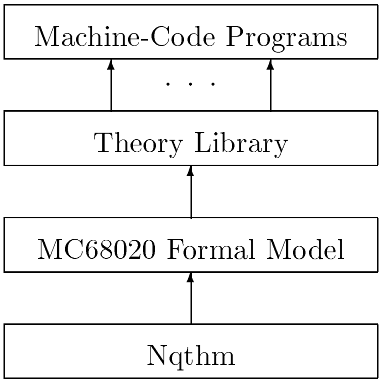

Figure 1.1: The Components of the Project

1.1 The Work

This report is about formally specifying and mechanically proving the correctness of machine-code programs using the automated reasoning system Nqthm, also known as the Boyer-Moore Theorem Proving System. On top of Nqthm, we formally defined a mathematical model of the MC68020, a widely used microprocessor built by Motorola, at the instruction-set level. We then proceeded to mechanize a mathematical theory tailored to machine-code reasoning. Finally, we studied the idea of mechanically verifying MC68020 object code produced by industrial strength optimizing high-level language compilers, such as the Gnu C or Verdix Ada compilers. We have successfully verified many such machine-code programs using Nqthm. An overall view of this work is shown in Figure 1.1.

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

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

[1] It is relevant to review Knuth's defense, in the Preface to *The Art of Computer Programming* [33], of his decision to present algorithms in assembler rather than in a higher-level language.

- Some of the most sensitive programs in the world are currently studied at the object-code level. For example, at several US Government agencies, including the DoD and the FAA, examiners look with great care at the machine code of critical systems, even though the systems were originally written in high-level programming languages. There are several good reasons for this.
    - Many high-level programming languages, especially those typically used in industrial practice, are not precisely specified. It is not easy, or even possible, to give the semantics of some programming language features, for example, the `volatile` type in C.
    - Some industrial strength compilers produce erroneous code. Until production compilers can be proved correct, we cannot rely on the code they produce. Validation at the machine-code level is the only alternative.
- Programs written in high-level languages may have assembly code embedded in them, in order to communicate with external devices. But no high-level formal language semantics we have seen has made clear the semantics of the embedding of assembler instructions.
- Real-time analysis is typically done at the machine-instruction level, because manufacturers often state how long an instruction takes to execute, but the definers of higher-level languages do not.

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

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

It is worth emphasizing that we are not advocating a return to programming in assembly or binary. Instead, our approach of studying the object code produced by high-level programming language compilers permits a programmer to continue to program in any high-level programming language while the correctness of the program is investigated at the machine-code level.

1.2 Related Work

There is a large body of literature on the topic of program proving. This section is by no means an exhaustive survey of the whole scientific field. Rather, we 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 [50] and Goldstine and von Neumann [19]. The latter paper discusses the specification and correctness proofs for fifteen programs at the machine-code level. Perhaps these were the earliest writings on program proving.

Methods for program proving have been advanced most notably by McCarthy [38], Floyd [18], and Hoare [22]. In the last twenty years, many research projects have focused on investigating the formal, mechanical verification of programs written in higher-level programming languages such as Pascal [24], Lisp [6], Fortran [5], and Gypsy [15]. Most of these projects are based on Floyd's inductive assertion method, and are therefore in the same spirit as the early mechanical verification work of King [32]. Our work differs from the 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 does research on formal, mechanical software verification address the correctness of programs at the machine-code level. To the best of our knowledge, Maurer [36] was the first person to address one of the major problems with machine-code verification — a machine-code program may modify itself. His solution was based on Floyd's inductive assertion method. The idea in Maurer's paper was to extend each verification condition with one additional assertion asserting the contents of the program segment. Hand proofs of a few very simple machine-code programs executed on toy hardware were given there. Maurer [35] later developed an IBM 370 assembly language verifier, and used it to verify some simple programs such as GCD.

Clutterbuck and Carré in [12] argued for the importance of the verification of low-level code, and, in a separate paper [25], 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, their work is also 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,[2] many restrictions had to be imposed on the assembly language, and complex annotations had to be inserted into the programs being verified.

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

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

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

The CLI short stack [3] is a small computing system consisting of a compiler, an assembler, a linker, and a gate-level design for a microprocessor that has been formally verified. In that work, Hunt proved the correctness

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

of a gate-level design for the FM8502 microprocessor;[3] Moore proved the correctness of a compiler for the assembly-level programming language Piton targeted to the verified FM8502; and Young proved the correctness of a code generator for the high-level programming language Micro-Gypsy targeted on the verified Piton. Their success inspired us to study the problem of specifying and verifying real programs executed on widely used hardware.

In contrast to our approach to machine-code proof, compiler verification attempts to establish the correctness of the compiler, so that we are ensured that the compiler always generates correct binary code. The first example of compiler proving seems to be the McCarthy and Painter [39] proof of a compiler for expression evaluation. They prove, by hand, the correctness of an expression compiler for an idealized machine using recursion induction. Mechanical proofs of slightly varied versions of the McCarthy-Painter expression compiler were later obtained by many researchers [7].

Polak's seems the most ambitious compiler verification effort [44]. Polak mechanically verified a compiler for a fairly substantial subset of Pascal. But his target machine was rather high-level and therefore unrealistic. In addition, it seemed he assumed a large collection of unproven lemmas which, in our opinion, should not be taken for granted.

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

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

Microcode verification is closely related to our work. Among the most significant reported work is the C/30 microcode verification using the State Delta Verification System(SDVS) [14]. A large majority of the C/30's instructions were proven to be correctly implemented by approximately 1000 MBB microinstructions. Hunt [51] and Cohn [13] are two major hardware design verification works involving microcode verification. Hunt reported

[3] FM8502 is a von Neumann machine designed by Warren Hunt. Its successor FM9001 [23] has been successfully fabricated.

some difficulties in microcode verification. We believe our techniques developed for machine-code verification would certainly contribute to microcode verification.

From a semantic point of view, our work is closely related to work on formal processor specification. A processor specification is a description of a computer architecture intended to provide a complete interface between processor and program. Intuitively, our formal MC68020 model is a processor specification that characterizes the behavior of MC68020 machine-code programs. Leonard [34] provides a comprehensive survey of work on architecture specification.

Most of the work on formal processor specification has adopted the operational approach; that is, the semantics of the machine is given by an abstract interpreter that describes how the state vector changes as the computation progresses [38]. Iverson proposed to use APL as an architecture specification language [26]. APL was later used to provide a complete formal description of the IBM system/360 architecture [16]. In the early 1970's, Bell and Newell introduced the specification language ISP [1]. ISP has been widely used to specify various computer architectures [46], most recently the SPARC [48]. ISP is one of the very few architecture specification languages that has achieved widespread use.

While the APL and ISP work was primarily motivated by providing a better notation for computer architecture description, McCarthy was perhaps the first person to connect the interpreter semantics with mathematical reasoning about programs. Our work is along the same lines as that of McCarthy [38, 37].

Processor specification has been intensively studied in the hardware verification communities where the main goal is the formal verification that a hardware design meets its architectural specification. Gordon [21] introduced LCF-LSM, and demonstrated its use in specifying and verifying Gordon's machine. Hunt used the Boyer-Moore logic to specify and verify the FM8502 microprocessor. Cohn [13] used the HOL system to specify and verify the Viper microprocessor. Most of the architectures studied in their work are either "on paper" or novel.

A group of researchers at Oxford have been working on formal processor specification using the formal specification language Z. They have specified the Motorola M6800 architecture [4], parts of the Motorola M68000 architecture [45], and the Inmos transputer architecture [17]. It seems that they have primarily focused on issues in formal specification. Little has been reported on any formal verification effort in their work.

1.3 Outline of the Report

The main product of this work is a powerful proof system built on top of Nqthm that allows us to reason about machine-code programs for the Motorola MC68020 microprocessor. To give the reader a clear picture of this project, we provide, in their verbatim form, our formal specification for the MC68020 microprocessor and our lemma library for machine-code reasoning in Appendix B. The complete script of all the program proofs presented here is given in [53]. The following is an outline of this report.

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

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

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

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

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

To demonstrate the usefulness of our system, we describe in Chapter 7 the formal verification of the Berkeley implementation of the ANSI/ISO C

String Library [52, 27].[4] Three programming errors were revealed in the process of our verification. Two were in the Berkeley Unix C string library.[5] The third was in Plauger's book *The Standard C Library* [43].[6]

The final chapter summarizes our main results, and considers the possible applications to our methodology. It also speculates on future research directions.

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

[5] One error was undetected when we reported it to the author Chris Torek [49]. It will be corrected for the release of BSD4.4. The other one was fixed about one year ago.

[6] P.J. Plauger had detected this error by the time we reported it to him [42].

Chapter 2

Formal Specification and Machine-Code Verification

This chapter gives a general characterization of our work before we dive into the technical details of the MC68020 formal model and machine-code program proving.

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

2.1 An Instruction-Set Specification of the MC68020

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

2.1.1 The Interpreter Semantics

An abstract finite state machine is defined by a specification of the machine state and a specification of a state transition relation on machine states. The machine state is specified as a vector of state components. The state transition relation is defined by an interpreter function acting on machine states.

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

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

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

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

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

DEFINITION:

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

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

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

This interpreter semantics describes the meaning of abstract machines in an intuitive and natural way. It can be easily understood by a wide range of computer professionals. For example, our MC68020 interpreter may be simply viewed as an architectural simulator for an MC68020 microprocessor defined in a formal logic.

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

2.1.2 The Specification

There are two main goals of our MC68020 formal specification. First, we provide a formal model to reflect as closely as possible the view of the MC68020 in the user's manual [40]. Second, this formal model should be amenable to automated reasoning. We wrote the specification with these two main goals in mind.

We have formalized most of the user programming model of the MC68020 microprocessor. However, we have not yet specified the supervisor level of the MC68020. Any exception caused by a user program simply halts our formalized machine. Figure 2.1 provides an informal, two dimensional picture of the user programming model for the MC68020, as described in [40]. This model has 16 32-bit general-purpose registers (8 data registers, D0-D7, and 8 address registers, A0-A7), a 32-bit program counter PC, and an 8-bit condition code register, CCR. The address register A7 is also used as the user stack pointer (USP). The 5 least significant bits in CCR are condition codes for carry, overflow, zero, negative, and extend. Our model is the only part of the state of an MC68020 that a user program can read or write under our formal semantics. Not present in our model are such arcane actualities as the instruction cache, memory management, and the supervisor stack.

Our specification consists of about 80% of all the user-available instructions and all eighteen MC68020 addressing modes. Most of the instructions we have left unspecified have some undefined effects on the machine state. For example, some of the condition codes of the instruction `CMP2` are described as "undefined" [40]. We have deliberately excluded such instructions from our specification. Fortunately, these instructions constitute only a small portion of the instruction set, and most of them are rarely used.[2] We summarize below the instructions we have formalized.

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

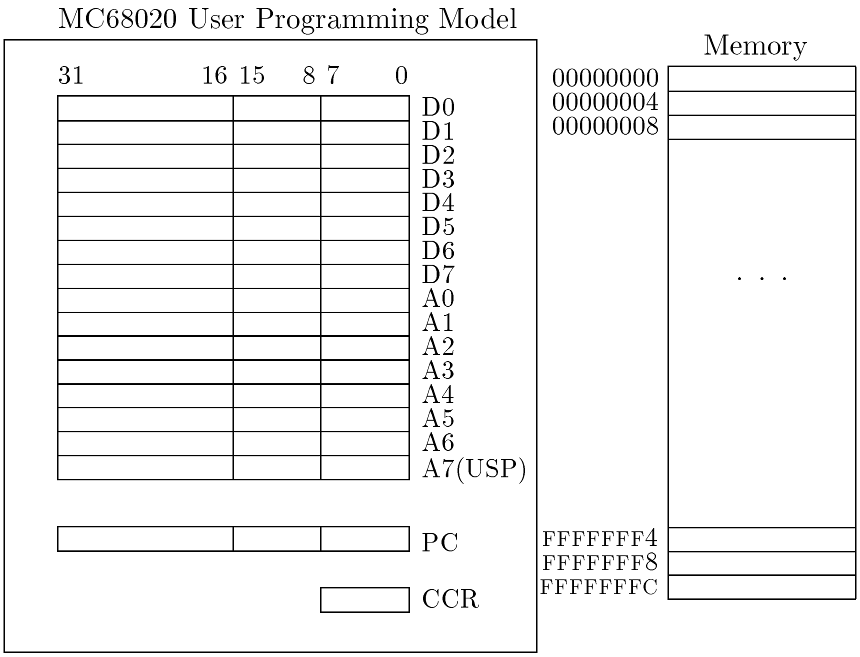

Figure 2.1: The User-Visible Machine State

The instructions of the MC68020 instruction set are classified into ten categories according to their functions [40].

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

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

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

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

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

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

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

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

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

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

Our formal specification is about $128,000$ bytes long, which takes up approximately 80 pages of text when printed. It consists of 569 function definitions in the Nqthm logic. The full text of this formal specification is given in [53]. The semantics of any machine-code program written in this subset of MC68020 instructions is given formally by our MC68020 model.

The complexity of this model is not particularly surprising to us. Rather, we believe the complexity is intrinsic for a CISC architecture like the MC68020.

2.2 Machine-Code Verification

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

2.2.1 Machine-Code Programs

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

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

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

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

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

```
0x22c4 <gcd+36>:        movel d0,d2
0x22c6 <gcd+38>:        bra 0x22b0 <gcd+16>
0x22c8 <gcd+40>:        divsll d2,d0,d3
0x22cc <gcd+44>:        movel d0,d3
0x22ce <gcd+46>:        bra 0x22b0 <gcd+16>
0x22d0 <gcd+48>:        movel d3,d0
0x22d2 <gcd+50>:        moveml fp@(-8),d2-d3
0x22d8 <gcd+56>:        unlk fp
0x22da <gcd+58>:        rts
(gdb) x/60ub gcd
<gcd>:     78     86     0      0      72     231    48     0
<gcd+8>:   36     46     0      8      38     46     0      12
<gcd+16>:  74     130    103    28     74     131    102    4
<gcd+24>:  32     2      96     22     182    130    108    8
<gcd+32>:  76     67     40     0      36     0      96     232
<gcd+40>:  76     66     56     0      38     0      96     224
<gcd+48>:  32     3      76     238    0      12     255    248
<gcd+56>:  78     94     78     117
(gdb) quit
rascal%
```

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

2.2.2 The Statement of Correctness

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

The Assumptions

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

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

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

Under these two assumptions, the program, when executed in an ideal environment, should behave the same as whatever the proved correctness theorem asserts. By “ideal execution environment”, we mean absence of power outages, hardware failures, and interference from the operating system, etc.

The Statement of the Correctness Theorem

The statement of the correctness theorem for a machine-code program should fully characterize the effects of the program’s execution 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 theorems about a program in other proofs. Our correctness theorem at the object-code level is more elaborate than one for a program written in a higher-level language. This is not particularly surprising. Our theorems assert more properties about a program than would higher-level program proving, because we have a more complicated model of the machine state.

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

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

Informally, the theorem says that, if the precondition $\textit{p-statep}\,(s)$ is satisfied, the properties specified by the relation *p-req* about the initial state s

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

and the resulting state *stepn*$(s,\ p\text{-}t(s))$, obtained by running the machine *p-t*(s) instructions from the initial state s, hold. Note that this theorem is completely based on the semantics given by *stepn*.

The precondition *p-statep* and the requirement *p-req* in the formula above both deserve further explanation.

The precondition *p-statep*(s) imposes certain conditions on the initial machine state to ensure the correct execution of the program. The conditions imposed in our formalism are given as follows, informally.

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

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

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

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

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

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

2.3 The Automated Reasoning System Nqthm

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

Nqthm is a Common Lisp program for proving mathematical theorems. Since *A Computational Logic* [7] was published in 1979, Nqthm has been widely used to check proofs of over 16,000 theorems from many areas of number theory, proof theory, and computer science. An extensive partial listing may be found in [9, pages 5–9]. In the body of this report, we use a conventional syntax rather than the official Lisp-like syntax of Nqthm. The translation between the conventional syntax and the official Lisp-like syntax is discussed in [11], and given in Appendix A.

2.3.1 The Logic

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

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

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

- *The Shell Principle* allows the user to add axioms introducing new inductively defined abstract data types. Natural numbers, symbols, and ordered pairs are axiomatized in the logic by adding shells:
  - *Natural Numbers.* The nonnegative integers are built from the constant `0` by successive applications of the constructor function *add1*. The function *numberp* recognizes natural numbers. The function *sub1* returns the predecessor of a non-`0` natural number. $x \in \mathbf{N}$ abbreviates $numberp(x)$.
  - *Symbols.* The data type of symbols, for example, **`'running`**, is built using the primitive constructor *pack* and `0`-terminated lists of ASCII codes. The symbol **`'nil`**, also abbreviated **nil**, is used to represent the empty list.
  - *Ordered Pairs.* Given two arbitrary objects, the function *cons* builds an ordered pair of these two objects. The function *listp* recognizes ordered pairs. The functions *car* and *cdr* return the first and second component of such an ordered pair. Lists of arbitrary length are constructed with nested pairs. Thus $list(arg_1, \ldots, arg_n)$ is an abbreviation for $cons(arg_1, ..., cons(arg_n, \mathbf{nil}))$.
- *The Definitional Principle* allows the user to define new functions in the logic. For recursive functions, there must be an ordinal measure of the arguments that can be proved to decrease in each recursion, which, intuitively, guarantees that one and only one function satisfies the definition. Many functions are added as part of the basic theory by this definitional principle. For example, we define for the natural numbers these familiar operations: $i + j$, $i - j$, $i < j$, $i * j$, $i \div j$, i **mod** j, and $exp\,(i,\, j)$. $i \simeq \mathtt{0}$ returns **f** if and only if i is a positive integer.
- *The Constraint Principle* allows the user to introduce and constrain new function symbols in the logic, rather than completely define them. To avoid introducing any new inconsistency into the logic, the user is required to prove that the proposed constraints are satisfiable by providing some already defined "witness" functions for the new function symbols.

The rules of inference of the logic consist of:

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

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

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

2.3.2 The Theorem Prover

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

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

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

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

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

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

  THEOREM: *lemma-name*
  statement

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

CONSERVATIVE AXIOM: *name*
axiom

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

  THEOREM: *lemma-name*
  statement

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

  DEFINITION: *fn-name* (*args*) ⇔ *form*

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

2.3.3 An Interactive Enhancement to Nqthm

While our work is completely built on top of Nqthm, we have found Kaufmann's PC-Nqthm system [28] a valuable tool for debugging Nqthm proofs. This system is fully integrated with Nqthm. Thus, the user can give commands at a low level (such as deleting a hypothesis) or at a high level (such as calling Nqthm).

As with a variety of proof-checking systems, PC-Nqthm is goal-directed; a proof is completed when the main goal and all subgoals have been proved. A notion of *macro commands* lets the user create compound commands, in the same spirit of the tactics and tacticals of LCF [20]. An interactive proof is complete when all goals have been proved. It is PC-Nqthm's low-level features that help us understand when and why a goal fails.

[5] This is an extension to Nqthm by Matt Kaufmann, which is not documented in [9]. See [29] for details.

Chapter 3

The MC68020 Instruction Set Specification

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

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

The entire MC68020 formal specification is given in [53].

3.1 Basic Concepts

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

3.1.1 Natural Numbers

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

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

DEFINITION:
$log\,(b,\ x)$
= if $(b \simeq 0) \lor (b = 1)$ then $0$
    elseif $x < b$ then $0$
    else $1 + log\,(b,\ x \div b)$ endif
```

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

3.1.2 Integer Arithmetic

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

Since the meanings of these functions are quite intuitive, we will not give their definitions here. The reader may find their definitions in [53].

3.1.3 Bit Vector Arithmetic

Bit vectors are represented as natural numbers in our formalism. For example, the content of the program counter is represented as a nonnegative integer with range between 0 and $2^{32} - 1$, inclusive. Each of the operations on bit vectors is therefore formalized as some sort of operation on nonnegative integers. The decision to use natural number representation was not easy to make. Finite lists, for example, seemed an equally good representation for bit vectors, and have been used successfully in hardware design verification [51]. In fact, we tried to use the finite list representation in our early version of the MC68020 specification. But we soon found it awkward for machine-code program proving. The choice of representation should take into account the often much more difficult task of automated reasoning.

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

The following are the definitions of the basic bit field manipulation operations.

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

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

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

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

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

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

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

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

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

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

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

DEFINITION: $\mathit{lognot}(n, x) = ((\mathit{exp}(2, n) - \mathit{head}(x, n)) - 1)$

```
DEFINITION:
$\mathit{logand}(x, y)$
=  if $(x \simeq 0) \lor (y \simeq 0)$ then 0
   else $\mathit{b\text{-}and}(\mathit{bcar}(x), \mathit{bcar}(y))$
         $+ \; (2 * \mathit{logand}(\mathit{bcdr}(x), \mathit{bcdr}(y)))$ endif
```

```
DEFINITION:
$\mathit{logor}(x, y)$
=  if $x \simeq 0$ then $\mathit{fix}(y)$
   elseif $y \simeq 0$ then $\mathit{fix}(x)$
   else $\mathit{b\text{-}or}(\mathit{bcar}(x), \mathit{bcar}(y))$
         $+ \; (2 * \mathit{logor}(\mathit{bcdr}(x), \mathit{bcdr}(y)))$ endif
```

```
DEFINITION:
$\mathit{logeor}(x, y)$
=  if $(x \simeq 0) \land (y \simeq 0)$ then 0
   else $\mathit{b\text{-}eor}(\mathit{bcar}(x), \mathit{bcar}(y))$
         $+ \; (2 * \mathit{logeor}(\mathit{bcdr}(x), \mathit{bcdr}(y)))$ endif
```

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

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

DEFINITION: $\mathit{adder}(n, c, x, y) = \mathit{head}(c + x + y, n)$

DEFINITION: $\mathit{add}(n, x, y) = \mathit{head}(x + y, n)$

DEFINITION:
$\mathit{subtractor}(n, c, x, y) = \mathit{adder}(n, \mathit{b\text{-}not}(c), y, \mathit{lognot}(n, x))$

DEFINITION:
$\mathit{sub}(n, x, y) = \mathit{head}(y + (\mathit{exp}(2, n) - \mathit{head}(x, n)), n)$

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

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

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

Definition: $\mathit{lsl}(\mathit{len}, x, \mathit{cnt}) = \mathit{head}(x * \mathit{exp}(2, \mathit{cnt}), \mathit{len})$

Definition: $\mathit{asl}(\mathit{len}, x, \mathit{cnt}) = \mathit{head}(x * \mathit{exp}(2, \mathit{cnt}), \mathit{len})$

Definition: $\mathit{lsr}(x, \mathit{cnt}) = \mathit{tail}(x, \mathit{cnt})$

Definition:

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

Definition:

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

Definition:

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

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

3.2 The User-Visible State

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

Definition:
$\mathit{mc\text{-}state}(\mathit{status}, \mathit{regs}, \mathit{pc}, \mathit{ccr}, \mathit{mem}) = \mathit{list}(\mathit{status}, \mathit{regs}, \mathit{pc}, \mathit{ccr}, \mathit{mem})$

Definition: $\mathit{mc\text{-}status}(s) = \mathit{car}(s)$

Definition: $\mathit{mc\text{-}rfile}(s) = \mathit{cadr}(s)$

Definition: $\mathit{mc\text{-}pc}(s) = \mathit{head}(\mathit{caddr}(s), \textsc{l})$

Definition: $\mathit{mc\text{-}ccr}(s) = \mathit{head}(\mathit{cadddr}(s), \textsc{b})$

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

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

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

3.2.1 The Processor Status Word

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

DEFINITION: READ-SIGNAL = `'read_unavailable_memory`

DEFINITION:
WRITE-SIGNAL = `'write_rom_or_unavailable_memory`

DEFINITION:
RESERVED-SIGNAL
= `'motorola_reserved_for_future_development`

DEFINITION: PC-SIGNAL = `'pc_outside_rom`

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

DEFINITION:
MODE-SIGNAL
= `'illegal_addressing_mode_in_current_instruction`

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

3.2.2 The Register File

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

DEFINITION:
$\mathit{read\text{-}rn}(\mathit{oplen},\ \mathit{rn},\ \mathit{regs}) = \mathit{head}(\mathit{get\text{-}nth}(\mathit{rn},\ \mathit{regs}),\ \mathit{oplen})$

DEFINITION:
$\mathit{write\text{-}rn}(\mathit{oplen},\ \mathit{value},\ \mathit{rn},\ \mathit{regs})$
$=\ \ \mathit{put\text{-}nth}(\mathit{replace}(\mathit{oplen},\ \mathit{value},\ \mathit{get\text{-}nth}(\mathit{rn},\ \mathit{regs})),\ \mathit{rn},\ \mathit{regs})$

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

3.2.3 The Program Counter

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

3.2.4 The Condition Code Register

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

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

DEFINITION: $\mathit{set\text{-}cvznx}(\mathit{cvznx},\ \mathit{ccr}) = \mathit{replace}(5,\ \mathit{cvznx},\ \mathit{ccr})$

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

3.2.5 The Memory

The memory is represented as a pair of binary trees. A binary representation for memory provides some efficiency for simulating MC68020 instructions. One of the binary trees is a formalization of memory protection—we may specify that any byte of memory is `'ram`, `'rom`, or `'unavailable`; the other binary tree holds the data, for example, the actual bytes stored. As discussed elsewhere in this chapter, we use the notion of read-only memory to deal with the issue of cache consistency. We also believe that it is unrealistic to assert the correctness of machine-code programs without carefully characterizing which parts of memory are read and written—few MC68020 chips are connected to a full 4 gigabytes of RAM. Memory protection issues are not specified in the MC68020 user's manual [40].

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

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

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

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

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

3.3 Internal States and Effective Address Calculation

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

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

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

The function *effec-addr* formalizes the effective address calculation. All the eighteen MC68020 addressing modes have been specified. An addressing mode can specify a constant that is the operand, a register that contains the operand, or a location in memory where the operand is stored. For the informal description and the formal definition, please refer to [40] and [53].

3.4 The Specification of the SUB Instruction

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

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

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

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

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

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

Roughly speaking, *execute-ins* decodes the current instruction according to the opcode and jumps to the specification of the instruction identified. The first argument of *execute-ins* should be the first word (operation word) of the current instruction, and the second argument should be an internal state with the program counter incremented by 2. The very top-level operation decoding is given by Table 3-14 in [40].

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

```
DEFINITION:
sub-ins1 (oplen, ins, s)
=  if sub-addr-modep1 (oplen, ins)
   then let s&addr  be  mc-instate (oplen, ins, s)
        in
        if mc-haltp (car (s&addr))  then car (s&addr)
```

[1] This is only one of the two cases in the SUB instruction; please refer to [40] and [11] for more details.

```
          else d-mapping (oplen,
                          sub-effect (oplen,
                                      operand (oplen,
                                               cdr (s&addr),
                                               s),
                                      read-dn (oplen,
                                               d_rn (ins),
                                               s)),
                          d_rn (ins),
                          car (s&addr)) endif endlet
      else halt (MODE-SIGNAL, s) endif
```

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

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

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

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

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

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

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

The function *cvznx* puts together the five new condition codes of the SUB instruction, which are formalized by the following four functions, paraphrasing the description given in Table 3-11 of the MC68020 manual [40]. The X flag is the same as the C flag.

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

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

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

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

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

3.5 Discussion

Having described the MC68020 specification in the preceding sections, we conclude this chapter with some of the interesting issues that have come up in the specification.

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

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

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

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

Ideally, we would specify the condition codes in a way most natural to the user. But in order to assure full compliance with the MC68020 specification, we have followed the syntactical definition described in Table 3-11 of [40]. For instance, the definition of *sub-c* is perhaps not the way the programmer views the carry bit of a SUB (subtraction) instruction. One of the problems we have to deal with in the verification phase is to prove an abstraction theorem that relates these different views. This problem has been addressed in the lemma library.

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

In addition to using the Nqthm prover to prove general theorems about the correctness of MC68020 programs under the semantics provided by *stepn*, 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. By doing so, we acquired some degree of confidence in our formal model. Ken Albin at Computational Logic has been working on a testing suite for the MC68020 specification.

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

- The executability of the formal model provides in some cases a fast means of symbolic manipulation during program proving.

Chapter 4

The Mechanization of Machine-Code Reasoning

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

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

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

Our approach to developing a lemma library can be roughly viewed as "bottom-up". We carefully study each of the concepts involved in the problem domain, in the hope of proving a set of lemmas that fully characterizes those concepts. Our presentation of the library in this chapter also follows this general approach. We will address some of the important issues we have dealt with in developing the library. The Nqthm script of the entire lemma library is in [53].

4.1 Integer Arithmetic

Integer arithmetic is the basic theory of our program proving work. In our work, we have at least one more reason to develop a powerful sublibrary for integer arithmetic: all the bit vector operations are formalized with nonnegative integer arithmetic; hence theorems about bit vectors are merely theorems about nonnegative integers. Most of the lemmas in this sublibrary are concerned with these basic arithmetic functions: $x + y$, $x - y$, $x * y$, x **mod** y, $x \div y$, $exp(x,\ y)$, $log(b,\ x)$, and $x < y$. During the development, we have greatly benefited from an integer library [30] developed at Computational Logic, Inc.

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

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

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

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

4.2 Bit Vector Arithmetic

Since we model the MC68020 at the machine-code level, it is inevitable that we must study the mathematical properties of bit vector operations. The purpose here is to establish a set of proof rules to support bit vector arithmetic reasoning at a relatively high level of abstraction. Reducing bit vector reasoning to integer arithmetic reasoning all the time is practically intractable.

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

Just to exhibit the lemmas of this class, we give the following two simple lemmas taken from the library.

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

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

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

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

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

4.3 Interpretations of Bit Vector Operations

At the machine-code level, mathematical functions are modeled by bit vector operations. It is therefore necessary to establish the correspondence between the real mathematical functions and their bit vector "implementations". We addressed this issue by using interpretation lemmas in our lemma library.

Basically, there are two kinds of lemmas we have considered: unsigned and signed integer interpretations. We have proved the interpretation lemmas for the basic unsigned and signed integer operations supported by the MC68020 instruction set. In this section, we explain the basic ideas using the two interpretation lemmas for addition.

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

DEFINITION: $\textit{nat-to-uint}\,(x) = \textit{fix}\,(x)$

DEFINITION: $\textit{uint-to-nat}\,(x) = \textit{fix}\,(x)$

DEFINITION:
$$\begin{aligned}&\textit{nat-to-int}\,(x,\ n)\\ =\ &\textbf{if}\ x < \textit{exp}\,(2,\ n-1)\ \ \textbf{then}\ \textit{fix}\,(x)\\ &\textbf{else}\ -\ (\textit{exp}\,(2,\ n) - x)\ \textbf{endif}\end{aligned}$$

DEFINITION:
$$\begin{aligned}&\textit{int-to-nat}\,(x,\ \textit{size})\\ =\ &\textbf{if}\ \textit{negativep}\,(x)\ \ \textbf{then}\ \textit{exp}\,(2,\ \textit{size}) - \textit{negative-guts}\,(x)\\ &\textbf{else}\ \textit{fix}\,(x)\ \textbf{endif}\end{aligned}$$

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

Now, let us consider the interpretations for the bit vector operation *add* whose definition was given in section 3.1.3. As we know from two's complement addition, the function *add* can be viewed as either unsigned or signed integer addition, depending on how we interpret the two bit vector inputs. Intuitively speaking, the lemma *add-uint* establishes the relation between *add* and *plus*, if the unsigned integer interpretation is taken; the lemma *add-int* establishes the relation between *add* and *iplus*, if the signed integer interpretation is taken.

THEOREM: *add-uint*
$$\begin{aligned}&(\textit{nat-rangep}\,(x,\ n) \land \textit{nat-rangep}\,(y,\ n))\\ \Rightarrow\ \ &(\textit{nat-to-uint}\,(\textit{add}\,(n,\ x,\ y))\\ &\quad =\ \textbf{if}\ (\textit{nat-to-uint}\,(x) + \textit{nat-to-uint}\,(y)) < \textit{exp}\,(2,\ n)\\ &\quad\quad\ \textbf{then}\ \textit{nat-to-uint}\,(x) + \textit{nat-to-uint}\,(y)\\ &\quad\quad\ \textbf{else}\ (\textit{nat-to-uint}\,(x) + \textit{nat-to-uint}\,(y))\\ &\quad\quad\quad\quad -\ \ \textit{exp}\,(2,\ n)\ \textbf{endif})\end{aligned}$$

```
THEOREM: add-int
 (nat-rangep (x, n) ∧ nat-rangep (y, n))
 ⇒  (nat-to-int (add (n, x, y), n)
        =  if int-rangep (iplus (nat-to-int (x, n), nat-to-int (y, n)), n)
           then iplus (nat-to-int (x, n), nat-to-int (y, n))
           elseif negativep (nat-to-int (x, n))
           then iplus (nat-to-int (x, n),
                        iplus (nat-to-int (y, n), exp (2, n)))
           else iplus (nat-to-int (x, n),
                       iplus (nat-to-int (y, n), − exp (2, n))) endif)
```

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

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

4.4 Machine-State Management

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

Intuitively, one might think of these lemmas as some kind of Hoare rules [22] for machine-code program proving. But these lemmas are rather complicated and delicate because of the complexity of the MC68020 architecture.

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

In this section, we briefly discuss the lemmas for the register file and the memory. The lemmas for the other state components are quite straightforward.

4.4.1 The Register File

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

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

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

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

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

4.4.2 The Memory

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

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

Definition:
read-write-mem-end (x, *value*, y, *mem*, m, n)
= *read-mem* (x, *write-mem* (*value*, y, *mem*, m), n)

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

Theorem: *read-write-mem2*
uint-rangep (n, 32)
$\Rightarrow$ (*read-mem* (x, *write-mem* (*value*, x, *mem*, n), n) = *head* (*value*, $8 * n$))

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

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

4.5 Interpretations of Condition Codes

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

In the preceding chapter, we gave the definition of the condition codes of the SUB instruction. But this definition is often not the most useful mathematical characterization of the condition codes to use when it comes

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

Table 4.1: The Bcc Condition Codes

to program proving. We therefore need to prove a set of lemmas that provides the mathematical meaning of condition codes used in program proving. Table 4.1 shows all the condition codes that can be specified in the Bcc instruction. The following lemmas characterize the most useful semantics of these condition codes for program proving.

Theorem: *sub-bls*
$(\mathit{nat\text{-}rangep}(x, n) \land \mathit{nat\text{-}rangep}(y, n) \land (n \not\simeq 0))$
$\Rightarrow (\mathit{bls}(\mathit{sub\text{-}c}(n, x, y), \mathit{sub\text{-}z}(n, x, y))$
$= \textbf{if } \mathit{nat\text{-}to\text{-}uint}(x) < \mathit{nat\text{-}to\text{-}uint}(y) \textbf{ then } 0$
$\textbf{else } 1 \textbf{ endif})$

Theorem: *sub-beq-uint*
$(\mathit{nat\text{-}rangep}(x, n) \land \mathit{nat\text{-}rangep}(y, n))$
$\Rightarrow (\mathit{beq}(\mathit{sub\text{-}z}(n, x, y))$
$= \textbf{if } \mathit{nat\text{-}to\text{-}uint}(x) = \mathit{nat\text{-}to\text{-}uint}(y) \textbf{ then } 1$
$\textbf{else } 0 \textbf{ endif})$

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

Theorem: *sub-bvs&vc*
$(\mathit{nat\text{-}rangep}(x, n) \land \mathit{nat\text{-}rangep}(y, n) \land (n \not\simeq 0))$
$\Rightarrow (\mathit{bvs}(\mathit{sub\text{-}v}(n, x, y))$
$= \textbf{if } \mathit{int\text{-}rangep}(\mathit{idifference}(\mathit{nat\text{-}to\text{-}int}(y, n), \mathit{nat\text{-}to\text{-}int}(x, n)),$
$n) \textbf{ then } 0$
$\textbf{else } 1 \textbf{ endif})$

Theorem: *sub-bmi*
$(\mathit{nat\text{-}rangep}(x, n) \land \mathit{nat\text{-}rangep}(y, n) \land (n \not\simeq 0))$
$\Rightarrow (\mathit{bmi}(\mathit{sub\text{-}n}(n, x, y))$

```
    = if int-rangep (idifference (nat-to-int (y, n), nat-to-int (x, n)),
                     n)
      then if ilessp (nat-to-int (y, n), nat-to-int (x, n)) then 1
           else 0 endif
      elseif ilessp (nat-to-int (y, n), nat-to-int (x, n)) then 0
      else 1 endif)
```

```
THEOREM: sub-bge
(nat-rangep (x, n) ∧ nat-rangep (y, n) ∧ (n ≄ 0))
⇒ (bge (sub-v (n, x, y), sub-n (n, x, y))
    = if ilessp (nat-to-int (y, n), nat-to-int (x, n)) then 0
      else 1 endif)
```

```
THEOREM: sub-bgt
(nat-rangep (x, n) ∧ nat-rangep (y, n) ∧ (n ≄ 0))
⇒ (bgt (sub-v (n, x, y), sub-z (n, x, y), sub-n (n, x, y))
    = if ilessp (nat-to-int (x, n), nat-to-int (y, n)) then 1
      else 0 endif)
```

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

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

4.6 The Interpreter Lemmas

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

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

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

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

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

Typically, the proof of the lemmas of this class is very shallow mathematically, but tedious and painful practically. Because of the complexity of the MC68020 model, this kind of proof often ends up splitting into a huge number of cases, and some lemmas have to be provided to control the case analysis. We believe this problem is intrinsic: any theorem prover has to visit every corner of the interpreter in order to prove a single fact about the interpreter.

Chapter 5

Machine-Code Program Proving

Among the possible applications of our MC68020 formal specification, we are at this time primarily concerned with studying the verification of specific object code programs. This chapter, together with the next two chapters, addresses the problem of program proving at the machine-code level, which is the central theme of this work. We illustrate our verification approach with some examples which provide evidence that this work can be applied to some moderately sized real applications. By presenting these examples, we provide the reader with a fair account of the difficulty of formalizing and proving machine-code programs.

In this chapter, we investigate the formal correctness proofs of the object code of five small programs written in high-level programming languages. The first one is the C function `gcd` already given in Chapter 2. The second is an ADA program `isqrt` that computes the integer square root using Newton's method. The third and the fourth are slightly modified versions of binary search and quick sort taken from *The C Programming Language* [31]. The last one is a C program that implements the Boyer-Moore Majority Voting algorithm. The object code of these programs is generated by Gnu C or Verdix Ada compilers, as explained in Chapter 2.

Proving programs has sharpened our understanding of the MC68020 model and the mathematics for machine-code reasoning. These five examples have been particularly beneficial to us. We feel that a detailed discussion of them would be equally beneficial to those verificationists who happen to attempt these examples on their own verification system.

This chapter contains six sections. The first describes our approach to machine-code program proving. The remaining five sections are devoted to the five examples. For each example, we discuss the formalization, the proof, and some other important issues such as the time analysis and memory space bounds of the program. We advise the reader to go through the section 5.2 first since many concepts are introduced there and not repeated in the other sections. The complete proof script for the five examples described in this chapter is given in [53].

5.1 The Approach

In Chapter 2, we briefly described our approach to machine-code program proving. This section provides a more rigorous mathematical treatment of our program-proving methodology.

5.1.1 The Formulation

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

- a predicate *p-statep* (s) that characterizes the preconditions on the initial state s where the program starts.

- a time function *p-t* (s) that defines the number of instructions needed to complete the computation.

- a set of mathematical functions *p-f1* (s), *p-f2* (s), ..., *p-fn* (s) that specifies the intended functional behavior of the program.

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

P-1. The resulting machine state is "normal"; for example, the processor status word is equal to `'running`.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

P-7. The functional behavior of the given program `p` is equivalent to the mathematical functions $p\text{-}f1\,(s)$, $p\text{-}f2\,(s)$, $\ldots$, $p\text{-}fn\,(s)$.

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

The equivalence relation *p-sem-eq* normally takes the form of a conjunct of equalities.

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

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

5.1.2 The Proof

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

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

2. We attempt to prove the theorem **P-8**, which establishes the correctness of the algorithm according to some specification. Note here we do not need to deal with any specifics related to MC68020 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.

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

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

5.2 Greatest Common Divisor

The first example continues the discussion started in Section 2.2.1. There we explained how we generated the machine code to be verified. Here, we will show the correctness proof of that machine code.

5.2.1 The Formalization

According to our approach, we need to formalize in the Nqthm logic the preconditions, the time function, and the functional behavior of the GCD machine-code program.

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

DEFINITION:
GCD-CODE

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

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

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

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

- s is in the user mode, for example, the processor status word is equal to `'running`.

- The program counter of s is even. The function *evenp* (x) asserts that x is even.

- The program GCD-CODE is stored in the 60 consecutive bytes in the memory, starting from the address pointed to by the program counter. The function *mcode-addrp* $(x,$ *mem*, *code*$)$ asserts that *code* is stored in the memory *mem* starting from the address x. The function *rom-addrp* $(x,$ *mem*, $n)$ asserts that the memory segment $[x, x+1, \ldots, (x+(n-1))]$ is ROM.

- The 24 bytes from $sp - 12$ to $sp + 12$ are RAM. The function *ram-addrp* $(x,$ *mem*, $n)$ asserts that the memory segment $[x, x+1, \ldots, (x+(n-1))]$ is RAM.

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

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

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

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

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

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

5.2.2 The Proof

We follow strictly the two-step proof outlined in Section 5.1.2.

In the first step, we prove the following theorem that is a conjunct of seven formulas corresponding exactly to the theorems **P-1** to **P-7**.

$$\begin{array}{l}
\text{Theorem: } \textit{gcd-correctness}\\
\textbf{let } \textit{sn} \textbf{ be } \textit{stepn}(s,\ \textit{gcd-t}(a,\ b))\\
\textbf{in}\\
\textit{gcd-statep}(s,\ a,\ b)\\
\Rightarrow\ ((\textit{mc-status}(\textit{sn}) = \texttt{'running})\\
\quad\quad \wedge\ (\textit{mc-pc}(\textit{sn}) = \textit{rts-addr}(s))\\
\quad\quad \wedge\ (\textit{read-rn}(\texttt{32},\ \texttt{14},\ \textit{mc-rfile}(\textit{sn}))\\
\quad\quad\quad\quad = \ \textit{read-rn}(\texttt{32},\ \texttt{14},\ \textit{mc-rfile}(s)))\\
\quad\quad \wedge\ (\textit{read-rn}(\texttt{32},\ \texttt{15},\ \textit{mc-rfile}(\textit{sn}))\\
\quad\quad\quad\quad = \ \textit{add}(\texttt{32},\ \textit{read-an}(\texttt{32},\ \texttt{7},\ s),\ \texttt{4}))\\
\quad\quad \wedge\ ((\textit{d2-7a2-5p}(\textit{rn}) \wedge (\textit{oplen} \leq \texttt{32}))\\
\quad\quad\quad\quad \Rightarrow\ (\textit{read-rn}(\textit{oplen},\ \textit{rn},\ \textit{mc-rfile}(\textit{sn}))\\
\quad\quad\quad\quad\quad\quad = \ \textit{read-rn}(\textit{oplen},\ \textit{rn},\ \textit{mc-rfile}(s))))\\
\quad\quad \wedge\ (\textit{disjoint}(x,\ k,\ \textit{sub}(\texttt{32},\ \texttt{12},\ \textit{read-sp}(s)),\ \texttt{24})\\
\quad\quad\quad\quad \Rightarrow\ (\textit{read-mem}(x,\ \textit{mc-mem}(\textit{sn}),\ k)\\
\quad\quad\quad\quad\quad\quad = \ \textit{read-mem}(x,\ \textit{mc-mem}(s),\ k)))\\
\quad\quad \wedge\ (\textit{iread-dn}(\texttt{32},\ \texttt{0},\ \textit{sn}) = \textit{gcd}(a,\ b)))\ \textbf{endlet}
\end{array}$$

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

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

$$\begin{array}{l}
\text{Theorem: } \textit{gcd-is-cd}\\
((a \textbf{ mod } \textit{gcd}(a,\ b)) = \texttt{0}) \wedge ((b \textbf{ mod } \textit{gcd}(a,\ b)) = \texttt{0})
\end{array}$$

$$\begin{array}{l}
\text{Theorem: } \textit{gcd-the-greatest}\\
((a \not\simeq \texttt{0}) \wedge (b \not\simeq \texttt{0}) \wedge ((a \textbf{ mod } x) = \texttt{0}) \wedge ((b \textbf{ mod } x) = \texttt{0}))\\
\Rightarrow\ (\textit{gcd}(a,\ b) \not< x)
\end{array}$$

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

5.2.3 A Simple Timing Analysis

The fact that the function *gcd-t*(a, b) returns the exact number of MC68020 instructions executed by the GCD program allows us to obtain the timing constraints of the GCD program by studying the mathematical properties of this *gcd-t* function. This approach is how we analyze the real-time bounds of machine-code programs.

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

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

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

The theorem *gcd-t-ubound* is just an immediate consequence of *gcd-t-ub*.

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

5.3 Integer Square Root

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

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

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

   1  function isqrt (i:integer) return integer is
   00000: link.w      a6, #-04
2     j : integer := (i / 2);
   00004: move.l      d2, d1
   00006: bge.b       06    -> 0e
   00008: addi.l      #01, d1
   0000e: asr.l       #01, d1
3  begin
4     while not ((i / j) >= j) loop
   00010: move.l      d2, d0
   00012: divsl.l     d1, d0:d0
   00016: trapv
   00018: cmp.l       d0, d1
   0001a: ble.b       01c    -> 038
5         j := (j + (i / j)) / 2;
   0001c: add.l       d1, d0
   0001e: trapv
   00020: move.l      d0, d1
   00022: bge.b       06    -> 02a
   00024: addi.l      #01, d1
   0002a: asr.l       #01, d1
4     while not ((i / j) >= j) loop
   0002c: move.l      d2, d0
   0002e: divsl.l     d1, d0:d0
   00032: trapv
6  end loop;
   00034: cmp.l       d0, d1
   00036: bgt.b       -01c    -> 01c
7  return j;
   00038: move.l      d1, d0
   0003a: unlk        a6
   0003c: rts
8 end isqrt;
```

5.3.1 The Formalization

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

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

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

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

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

```
DEFINITION:
isqrt1-t (i, j)
=  if $j \simeq 0$ then 0
   elseif $(i \div j) < j$
   then splus (10, isqrt1-t (i, $(j + (i \div j)) \div 2$))
   else 8 endif
```

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

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

```
DEFINITION:
isqrt1 (i, j)
=  if $j \simeq 0$ then fix (i)
   elseif $(i \div j) < j$ then isqrt1 (i, $(j + (i \div j)) \div 2$)
   else fix (j) endif
```

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

5.3.2 The Proof

We follow strictly the two-step proof outlined in Section 5.1.2.

In the first step, we prove the following theorem that corresponds to the theorems **P-1** to **P-7**.

THEOREM: *isqrt-correctness*

$$
\begin{array}{l}
\textbf{let } sn \textbf{ be } \mathit{stepn}(s,\ \mathit{isqrt\text{-}t}(i)) \\
\textbf{in} \\
\mathit{isqrt\text{-}statep}(s,\ i) \\
\Rightarrow\ ((\mathit{mc\text{-}status}(sn) = \texttt{'running}) \\
\qquad \land\ (\mathit{mc\text{-}pc}(sn) = \mathit{rts\text{-}addr}(s)) \\
\qquad \land\ (\mathit{read\text{-}rn}(\texttt{32},\ \texttt{14},\ \mathit{mc\text{-}rfile}(sn)) = \mathit{read\text{-}an}(\texttt{32},\ \texttt{6},\ s)) \\
\qquad \land\ (\mathit{read\text{-}rn}(\texttt{32},\ \texttt{15},\ \mathit{mc\text{-}rfile}(sn)) \\
\qquad\qquad =\ \mathit{add}(\texttt{32},\ \mathit{read\text{-}an}(\texttt{32},\ \texttt{7},\ s),\ \texttt{4})) \\
\qquad \land\ (\mathit{d2\text{-}7a2\text{-}5p}(rn) \\
\qquad\qquad \Rightarrow\ (\mathit{read\text{-}rn}(\mathit{oplen},\ rn,\ \mathit{mc\text{-}rfile}(sn)) \\
\qquad\qquad\qquad =\ \mathit{read\text{-}rn}(\mathit{oplen},\ rn,\ \mathit{mc\text{-}rfile}(s)))) \\
\qquad \land\ (\mathit{disjoint}(x,\ k,\ \mathit{sub}(\texttt{32},\ \texttt{12},\ \mathit{read\text{-}sp}(s)),\ \texttt{20}) \\
\qquad\qquad \Rightarrow\ (\mathit{read\text{-}mem}(x,\ \mathit{mc\text{-}mem}(sn),\ k) \\
\qquad\qquad\qquad =\ \mathit{read\text{-}mem}(x,\ \mathit{mc\text{-}mem}(s),\ k))) \\
\qquad \land\ (\mathit{iread\text{-}dn}(\texttt{32},\ \texttt{0},\ sn) = \mathit{isqrt}(i)))\ \textbf{endlet}
\end{array}
$$

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

THEOREM: *isqrt-logic-correctness*

$$
(\texttt{1} < i) \Rightarrow ((i < \mathit{sq}(\texttt{1} + \mathit{isqrt}(i))) \land (i \not< \mathit{sq}(\mathit{isqrt}(i))))
$$

5.3.3 A Simple Timing Analysis

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

THEOREM: *isqrt-t-ubound*

$$
((i < \mathit{exp}(\texttt{2},\ \texttt{31})) \land (\texttt{1} < i)) \Rightarrow (\mathit{isqrt\text{-}t}(i) \leq \texttt{322})
$$

5.4 Binary Search

In our third example, we study binary search. The following C function `bsearch` taken from page 58 of Kernighan and Ritchie [31] with some minor modification searches for the occurrence of a given integer in a sorted integer array. In this section, we describe the correctness proof of the object code of this C function.

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

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

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

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

```
<bsearch+60>:    moveml a6@(-8),d2-d3
<bsearch+66>:    unlk a6
<bsearch+68>:    rts
```

5.4.1 The Formalization

As desribed in Section 5.1.1, we need to formalize in the Nqthm logic the preconditions, the time function, and the functional behavior of this BSEARCH machine-code program.

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

DEFINITION:
BSEARCH-CODE

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

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

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

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

DEFINITION:
bsearch1-t(x, lst, i, j)

```
= let k be (i + j) ÷ 2
  in
  if i < j
```

```
    then if $ilessp(x, get\text{-}nth(k, lst))$
            then $splus(10, bsearch1\text{-}t(x, lst, i, k))$
            elseif $ilessp(get\text{-}nth(k, lst), x)$
            then $splus(13, bsearch1\text{-}t(x, lst, 1 + k, j))$
            else 13 endif
    else 6 endif endlet
```

DEFINITION:
$bsearch\text{-}t(x, n, lst) = splus(6, bsearch1\text{-}t(x, lst, 0, n))$

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

DEFINITION:

```
$bsearch1(x, lst, i, j)$
= let $k$ be $(i + j) \div 2$
  in
  if $i < j$
  then if $ilessp(x, get\text{-}nth(k, lst))$ then $bsearch1(x, lst, i, k)$
          elseif $ilessp(get\text{-}nth(k, lst), x)$
          then $bsearch1(x, lst, 1 + k, j)$
          else $k$ endif
  else -1 endif endlet
```

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

5.4.2 The Proof

We strictly follow the two-step proof outlined in Section 5.1.2.

In the first step, we prove the following theorem that corresponds to the theorems **P-1** to **P-7**.

THEOREM: *bsearch-correctness*

```
let $sn$ be $stepn(s, bsearch\text{-}t(x, n, lst))$
in
$bsearch\text{-}statep(s, x, a, n, lst)$
$\Rightarrow$ $((mc\text{-}status(sn) = \text{'running})$
      $\wedge$ $(mc\text{-}pc(sn) = rts\text{-}addr(s))$
      $\wedge$ $(read\text{-}rn(32, 14, mc\text{-}rfile(sn))$
            $= read\text{-}rn(32, 14, mc\text{-}rfile(s)))$
      $\wedge$ $(read\text{-}rn(32, 15, mc\text{-}rfile(sn))$
            $= add(32, read\text{-}sp(s), 4))$
      $\wedge$ $((d2\text{-}7a2\text{-}5p(rn) \wedge (oplen \leq 32))$
            $\Rightarrow$ $(read\text{-}rn(oplen, rn, mc\text{-}rfile(sn))$
                  $= read\text{-}rn(oplen, rn, mc\text{-}rfile(s))))$
```

$$\begin{aligned}
\land\ \ & (\mathit{disjoint}\,(x,\ k,\ \mathit{sub}\,(\mathtt{32},\ \mathtt{12},\ \mathit{read\text{-}sp}\,(s)),\ \mathtt{28}) \\
& \quad \Rightarrow\ \ (\mathit{read\text{-}mem}\,(x,\ \mathit{mc\text{-}mem}\,(sn),\ k) \\
& \qquad\quad =\ \ \mathit{read\text{-}mem}\,(x,\ \mathit{mc\text{-}mem}\,(s),\ k))) \\
\land\ \ & (\mathit{iread\text{-}dn}\,(\mathtt{32},\ \mathtt{0},\ sn) = \mathit{bsearch}\,(x,\ n,\ \mathit{lst}))) \ \textbf{endlet}
\end{aligned}$$

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

1. If $\mathit{bsearch}\,(x,\ n,\ \mathit{lst})$ returns other than -1, then it returns an (non-negative) integer k such that the kth element of *lst* is equal to the integer x.

2. If $\mathit{bsearch}\,(x,\ n,\ \mathit{lst})$ returns -1 and *lst* is ordered, then the integer x is not in *lst*.

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

THEOREM: *bsearch-found*

$$\begin{aligned}
& ((\mathit{bsearch}\,(x,\ n,\ \mathit{lst}) \neq \mathtt{-1}) \land \mathit{lst\text{-}integerp}\,(\mathit{lst}) \land \mathit{integerp}\,(x)) \\
& \Rightarrow\ \ (\mathit{get\text{-}nth}\,(\mathit{bsearch}\,(x,\ n,\ \mathit{lst}),\ \mathit{lst}) = x)
\end{aligned}$$

THEOREM: *bsearch-not-found*

$$\begin{aligned}
& ((\mathit{bsearch}\,(x,\ \mathit{len}\,(\mathit{lst}),\ \mathit{lst}) = \mathtt{-1}) \\
& \ \land\ \ \mathit{orderedp}\,(\mathit{lst}) \\
& \ \land\ \ \mathit{lst\text{-}integerp}\,(\mathit{lst}) \\
& \ \land\ \ \mathit{integerp}\,(x)) \\
& \Rightarrow\ \ (x \notin \mathit{lst})
\end{aligned}$$

5.4.3 A Simple Timing Analysis

In a way similar to the preceding two examples, we have proved that $\mathit{bsearch\text{-}t}\,(x,\ n,\ \mathit{lst})$ is at most 435, which gives us an upper bound of the number of instructions executed by the machine-code program BSEARCH. We assume that n is less than 2^{31}.

THEOREM: *bsearch-t-ubound*

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

5.5 Quicksort

Quicksort was our first example dealing with recursion. The following C program `qsort` taken from page 87 of Kernighan and Ritchie [31] with some minor modification sorts an integer array into ascending order. The correctness proof of the object code of this program was rather complicated. It took us a couple of weeks to come up with a proof. It seemed that our life would have been much easier had we first studied some simpler example, something like Fibonacci numbers.

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

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

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

```
0x22b8 <qsort>:         linkw fp,#0
0x22bc <qsort+4>:       moveml d2-d4/a2-a3,sp@-
0x22c0 <qsort+8>:       moveal fp@(8),a3
0x22c4 <qsort+12>:      movel fp@(12),d3
0x22c8 <qsort+16>:      movel fp@(16),d4
0x22cc <qsort+20>:      cmpl d3,d4
0x22ce <qsort+22>:      ble 0x2338 <qsort+128>
```

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

5.5.1 The Formalization

According to our approach, we need to formalize in the Nqthm logic the preconditions, the time function, and the functional behavior of this QSORT machine-code program.

The function QSORT-CODE represents the machine code of `qsort` as a list of unsigned integers. The function *qstack*(l, r, *lst*) specifies the stack space needed for the program. The function *qsort-statep* (s, a, l, r, n, lst) characterizes the preconditions of the initial state s.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.5.2 The Proof

We follow strictly the two-step proof outlined in Section 5.1.2.

In the first step, we prove the following theorem that corresponds to the theorems **P-1** to **P-7**.

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

$$
\begin{array}{ll}
& \quad\quad = \ \textit{read-rn}(\textit{oplen},\ \textit{rn},\ \textit{mc-rfile}(s)))) \\
\wedge & ((\textit{disjoint}(\textit{sp},\ \textit{qstack}(l,\ r,\ \textit{lst}),\ x,\ k) \\
& \quad \wedge\ \ \textit{disjoint}(a,\ 4 * n,\ x,\ k)) \\
& \quad \Rightarrow\ \ (\textit{read-mem}(x,\ \textit{mc-mem}(\textit{sn}),\ k) \\
& \quad\quad =\ \ \textit{read-mem}(x,\ \textit{mc-mem}(s),\ k))) \\
\wedge & \textit{mem-ilst}(4,\ a,\ \textit{mc-mem}(\textit{sn}),\ n,\ \textit{qsort}(l,\ r,\ \textit{lst}))) \ \textbf{endlet}
\end{array}
$$

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

Definition:
$\textit{orderedp1}(l,\ r,\ \textit{lst})$

$$
\begin{array}{ll}
= & \textbf{if}\ r \le l\ \textbf{then t} \\
& \textbf{else}\ \textit{ileq}(\textit{get-nth}(l,\ \textit{lst}),\ \textit{get-nth}(1 + l,\ \textit{lst})) \\
& \quad\quad \wedge\ \ \textit{orderedp1}(1 + l,\ r,\ \textit{lst})\ \textbf{endif}
\end{array}
$$

Theorem: *qsort-orderedp1*
$\textit{orderedp1}(\textit{left},\ \textit{right},\ \textit{qsort}(\textit{left},\ \textit{right},\ \textit{lst}))$

Definition:
$\textit{count-lst}(x,\ l,\ r,\ \textit{lst})$

$$
\begin{array}{ll}
= & \textbf{if}\ r < l\ \textbf{then}\ 0 \\
& \textbf{elseif}\ x = \textit{get-nth}(l,\ \textit{lst})\ \ \textbf{then}\ 1 + \textit{count-lst}(x,\ 1 + l,\ r,\ \textit{lst}) \\
& \textbf{else}\ \textit{count-lst}(x,\ 1 + l,\ r,\ \textit{lst})\ \textbf{endif}
\end{array}
$$

Theorem: *count-lst-qsort*
$\textit{count-lst}(x,\ l,\ r,\ \textit{qsort}(l,\ r,\ \textit{lst})) = \textit{count-lst}(x,\ l,\ r,\ \textit{lst})$

Roughly speaking, the theorem *qsort-orderedp1* asserts that the list $\textit{qsort}(\textit{left},\ \textit{right},\ \textit{lst})$ is in ascending order; the theorem *count-lst-qsort* asserts that $\textit{qsort}(l,\ r,\ \textit{lst})$ is a permutation of *lst*. The proof of these two theorems required many supporting lemmas. We refer the interested readers to [53].

5.5.3 A Simple Stack Space Analysis

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

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

Theorem: *qstack-ubound*
$qstack(l,\ r,\ lst) \le (68 + (52 * (r - l)))$

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

Theorem: *qstack-ubound-la-1*
$(l < r)$
$\Rightarrow\ ((52 * (r - l))$
$\quad\not<\ (52 + (52 * ((qlast(l,\ r,\ lst) - 1) - l))))$

Theorem: *qstack-ubound-la-2*
$(l < r)$
$\Rightarrow\ ((52 * (r - l))$
$\quad\not<\ (52 + (52 * (r - (1 + qlast(l,\ r,\ lst))))))$

5.6 The Boyer-Moore Majority Voting Algorithm

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

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

struct winner {
  int x;
  int y;
};

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

  k = 0;
  for (i = 0; i < n; i++)
    if (k == 0) {
      cand = a[i];
      k = 1;
    }
  else {
    if (cand == a[i])
      k++;
    else
      k--;
  };
  temp.x = cand;
  if (k == 0) {
    temp.y = NO;
    return temp;
  };
  if (k > n/2) {
    temp.y = YES;
    return temp;
  };
  k = 0;
  for (i = 0; i < n; i++)
    if (a[i] == cand)
      k++;
  if (k > n/2)
    temp.y = YES;
  else temp.y = NO;
  return temp;
}
```

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

```
0x2310 <mjrty>:         linkw a6,#0
0x2314 <mjrty+4>:       moveml d2-d5,sp@-
0x2318 <mjrty+8>:       moveal a6@(8),a0
0x231c <mjrty+12>:      movel a6@(12),d2
0x2320 <mjrty+16>:      clrl d1
0x2322 <mjrty+18>:      clrl d0
0x2324 <mjrty+20>:      cmpl d0,d2
0x2326 <mjrty+22>:      ble 0x2346 <mjrty+54>
0x2328 <mjrty+24>:      tstl d1
0x232a <mjrty+26>:      bne 0x2334 <mjrty+36>
0x232c <mjrty+28>:      movel 0(a0)[d0.l*4],d3
```

```
0x2330 <mjrty+32>:      movel #1,d1
0x2332 <mjrty+34>:      bra 0x2340 <mjrty+48>
0x2334 <mjrty+36>:      cmpl 0(a0)[d0.l*4],d3
0x2338 <mjrty+40>:      bne 0x233e <mjrty+46>
0x233a <mjrty+42>:      addql #1,d1
0x233c <mjrty+44>:      bra 0x2340 <mjrty+48>
0x233e <mjrty+46>:      subl #1,d1
0x2340 <mjrty+48>:      addql #1,d0
0x2342 <mjrty+50>:      cmpl d0,d2
0x2344 <mjrty+52>:      bgt 0x2328 <mjrty+24>
0x2346 <mjrty+54>:      movel d3,d4
0x2348 <mjrty+56>:      tstl d1
0x234a <mjrty+58>:      beq 0x2382 <mjrty+114>
0x234c <mjrty+60>:      movel d2,d0
0x234e <mjrty+62>:      bge 0x2352 <mjrty+66>
0x2350 <mjrty+64>:      addql #1,d0
0x2352 <mjrty+66>:      asrl #1,d0
0x2354 <mjrty+68>:      cmpl d1,d0
0x2356 <mjrty+70>:      bge 0x235c <mjrty+76>
0x2358 <mjrty+72>:      movel #1,d5
0x235a <mjrty+74>:      bra 0x2384 <mjrty+116>
0x235c <mjrty+76>:      clrl d1
0x235e <mjrty+78>:      clrl d0
0x2360 <mjrty+80>:      cmpl d0,d2
0x2362 <mjrty+82>:      ble 0x2372 <mjrty+98>
0x2364 <mjrty+84>:      cmpl 0(a0)[d0.l*4],d3
0x2368 <mjrty+88>:      bne 0x236c <mjrty+92>
0x236a <mjrty+90>:      addql #1,d1
0x236c <mjrty+92>:      addql #1,d0
0x236e <mjrty+94>:      cmpl d0,d2
0x2370 <mjrty+96>:      bgt 0x2364 <mjrty+84>
0x2372 <mjrty+98>:      movel d2,d0
0x2374 <mjrty+100>:     bge 0x2378 <mjrty+104>
0x2376 <mjrty+102>:     addql #1,d0
0x2378 <mjrty+104>:     asrl #1,d0
0x237a <mjrty+106>:     cmpl d1,d0
0x237c <mjrty+108>:     bge 0x2382 <mjrty+114>
0x237e <mjrty+110>:     movel #1,d5
0x2380 <mjrty+112>:     bra 0x2384 <mjrty+116>
0x2382 <mjrty+114>:     clrl d5
0x2384 <mjrty+116>:     movel d4,d0
0x2386 <mjrty+118>:     movel d5,d1
0x2388 <mjrty+120>:     moveml a6@(-16),d2-d5
0x238e <mjrty+126>:     unlk a6
0x2390 <mjrty+128>:     rts
```

5.6.1 The Formalization

According to our approach, we need to formalize in the Nqthm logic the preconditions, the time function, and the functional behavior of this MJRTY machine-code program.

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

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

```
DEFINITION:
$mjrty\text{-}statep\,(s,\ a,\ n,\ lst)$
=  $((mc\text{-}status\,(s) = \texttt{'running})$
   $\land$  $evenp\,(mc\text{-}pc\,(s))$
   $\land$  $rom\text{-}addrp\,(mc\text{-}pc\,(s),\ mc\text{-}mem\,(s),\ 130)$
   $\land$  $mcode\text{-}addrp\,(mc\text{-}pc\,(s),\ mc\text{-}mem\,(s),\ \text{MJRTY-CODE})$
   $\land$  $ram\text{-}addrp\,(sub\,(32,\ 20,\ read\text{-}sp\,(s)),\ mc\text{-}mem\,(s),\ 32)$
   $\land$  $ram\text{-}addrp\,(a,\ mc\text{-}mem\,(s),\ 4 * n)$
   $\land$  $mem\text{-}ilst\,(4,\ a,\ mc\text{-}mem\,(s),\ n,\ lst)$
   $\land$  $disjoint\,(a,\ 4 * n,\ sub\,(32,\ 20,\ read\text{-}sp\,(s)),\ 32)$
   $\land$  $(a = read\text{-}mem\,(add\,(32,\ read\text{-}sp\,(s),\ 4),\ mc\text{-}mem\,(s),\ 4))$
   $\land$  $(n = iread\text{-}mem\,(add\,(32,\ read\text{-}sp\,(s),\ 8),\ mc\text{-}mem\,(s),\ 4))$
   $\land$  $(n \not\simeq 0))$
```

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

```
DEFINITION:
$mjrty\text{-}cand\text{-}t\,(a,\ n,\ lst,\ cand,\ i,\ k)$
=  if $i < n$
   then if $k \simeq 0$
        then let $cand1$  be  $get\text{-}nth\,(i,\ lst)$
             in
             $splus\,(8,\ mjrty\text{-}cand\text{-}t\,(a,\ n,\ lst,\ cand1,\ 1 + i,\ 1))$ endlet
        elseif $cand = get\text{-}nth\,(i,\ lst)$
```

```
            then splus(9, mjrty-cand-t(a, n, lst, cand, $1 + i$, $1 + k$))
            else splus(8, mjrty-cand-t(a, n, lst, cand, $1 + i$, $k - 1$)) endif
    elseif cand = get-nth(0, lst) then 18
    else 17 endif
```

DEFINITION:

```
mjrty-sn-t(a, n, lst, cand, i, k)
=   if $i < n$
    then if $k \simeq 0$
            then let cand1 be get-nth(i, lst)
                  in
                  splus(8, mjrty-sn-t(a, n, lst, cand1, $1 + i$, 1)) endlet
            elseif cand = get-nth(i, lst)
            then splus(9, mjrty-sn-t(a, n, lst, cand, $1 + i$, $1 + k$))
            else splus(8, mjrty-sn-t(a, n, lst, cand, $1 + i$, $k - 1$)) endif
    elseif $k \simeq 0$ then 11
    else 17 endif
```

DEFINITION:

```
cand-cnt-t(a, n, lst, cand, i, k)
=   if $i < n$
    then if cand = get-nth(i, lst)
            then splus(6, cand-cnt-t(a, n, lst, cand, $1 + i$, $1 + k$))
            else splus(5, cand-cnt-t(a, n, lst, cand, $1 + i$, k)) endif
    elseif $(n \div 2) < k$ then 14
    else 13 endif
```

DEFINITION:

```
mjrty-t(a, n, lst)
=   let cand be get-nth(0, lst)
    in
    splus(14,
            if (mjrty-k(n, lst, cand, 1, 1) $\simeq$ 0)
               $\vee$ (($n \div 2$) < mjrty-k(n, lst, cand, 1, 1))
            then mjrty-sn-t(a, n, lst, cand, 1, 1)
            else splus(mjrty-cand-t(a, n, lst, cand, 1, 1),
                         if cand = mjrty-cand(n,
                                                  lst,
                                                  cand,
                                                  1,
                                                  1)
                         then cand-cnt-t(a,
                                           n,
                                           lst,
                                           mjrty-cand(n,
                                                        lst,
                                                        cand,
                                                        1,
                              1),
                         1,
                         1)
                 else cand-cnt-t(a,
                                 n,
                                 lst,
                                 mjrty-cand(n,
                                            lst,
                                            cand,
                                            1,
                                            1),
                                 1,
                                 0) endif) endif) endlet
```

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

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

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

```
DEFINITION:
cand-cnt(n, lst, cand, i, k)
=  if i < n
   then if cand = get-nth(i, lst)
        then cand-cnt(n, lst, cand, 1 + i, 1 + k)
        else cand-cnt(n, lst, cand, 1 + i, k) endif
   else k endif
```

```
DEFINITION:
mjrty-p(n, lst, cand, i, k)
=  if mjrty-k(n, lst, cand, i, k) ≃ 0 then f
```

```
elseif (n ÷ 2) < mjrty-k(n, lst, cand, i, k)  then t
else (n ÷ 2)
      <  cand-cnt(n, lst, mjrty-cand(n, lst, cand, i, k), i, k) endif
```

5.6.2 The Proof

We follow strictly the two-step proof outlined in Section 5.1.2.

In the first step, we prove the following theorem that corresponds to the theorems **P-1** to **P-7**.

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

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

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

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

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

which is given formally as the following two theorems.

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

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

5.6.3 A Simple Timing Analysis

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

THEOREM: *mjrty-t-ubound*
$mjrty\text{-}t\,(a,\ n,\ lst) \le (\mathtt{46} + (\mathtt{15} * (n - \mathtt{1})))$

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

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

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

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

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

Chapter 6

Issues in Machine-Code Program Proving

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

There are four sections in this chapter, each of which addresses one program-proving problem that we feel is important and interesting. In these sections, we discuss the verification of some simple programs that illustrate how we handle those program-proving problems at the machine-code level. The examples used in this chapter are toy programs designed just for the purpose of exposition. The full proof script of these examples is given in [53].

6.1 Subroutine Calling

Composing proofs in program proving is as essential as composing programs in programming. Handling subroutine calling in machine-code program proving has been one of the main considerations in our formulation of correctness for machine-code programs. The correctness theorem of a subroutine should characterize the behavior of the subroutine well enough so that the same theorem can be used repeatedly in the proof of a large class of programs that call the subroutine, just as the same subroutine can be used repeatedly in many programs. In this section, we use an extremely simple example to illustrate how to handle subroutine calling in our formalization. To some extent, we have encountered this problem in the `qsort` example of Chapter 5. But we avoided discussing it there.

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

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

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

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

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

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

CONSERVATIVE AXIOM: *gcd3-load*
gcd3-loadp (s)
= (*evenp* (GCD3-ADDR)
  $\wedge$ (GCD3-ADDR $\in \mathbf{N}$)
  $\wedge$ *nat-rangep* (GCD3-ADDR, 32)
  $\wedge$ *rom-addrp* (GCD3-ADDR, *mc-mem* (s), 40)
  $\wedge$ *mcode-addrp* (GCD3-ADDR, *mc-mem* (s), GCD3-CODE)
  $\wedge$ *gcd-loadp* (s)
  $\wedge$ (*pc-read-mem* (*add* (32, GCD3-ADDR, 20), *mc-mem* (s), 4)
    = GCD-ADDR))

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

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

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

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

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

DEFINITION: $\mathit{gcd3\text{-}t2}(a,\ b,\ c) = \mathtt{3}$

DEFINITION: $\mathit{gcd3\text{-}t3}(a,\ b,\ c) = \mathit{gcd\text{-}t}(\mathit{gcd}(a,\ b),\ c)$

DEFINITION: $\mathit{gcd3\text{-}t4}(a,\ b,\ c) = \mathtt{3}$

DEFINITION:

$$
\begin{array}{l}
\mathit{gcd3\text{-}t}(a,\ b,\ c) \\
= \ \ \mathit{splus}(\mathit{gcd3\text{-}t0}(a,\ b,\ c), \\
\qquad\qquad \mathit{splus}(\mathit{gcd3\text{-}t1}(a,\ b,\ c), \\
\qquad\qquad\qquad\quad \mathit{splus}(\mathit{gcd3\text{-}t2}(a,\ b,\ c), \\
\qquad\qquad\qquad\qquad\qquad \mathit{splus}(\mathit{gcd3\text{-}t3}(a,\ b,\ c),\ \mathit{gcd3\text{-}t4}(a,\ b,\ c)))))
\end{array}
$$

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

DEFINITION: $\mathit{gcd3}(a,\ b,\ c) = \mathit{gcd}(\mathit{gcd}(a,\ b),\ c)$

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

THEOREM: *gcd3-correctness*

$$
\begin{array}{l}
\textbf{let}\ \mathit{sn}\ \ \textbf{be}\ \ \mathit{stepn}(s,\ \mathit{gcd3\text{-}t}(a,\ b,\ c)) \\
\textbf{in} \\
\mathit{gcd3\text{-}statep}(s,\ a,\ b,\ c) \\
\Rightarrow\ \ ((\mathit{mc\text{-}status}(\mathit{sn}) = \mathtt{'running}) \\
\qquad \land\ \ (\mathit{mc\text{-}pc}(\mathit{sn}) = \mathit{rts\text{-}addr}(s)) \\
\qquad \land\ \ (\mathit{read\text{-}rn}(\mathtt{32},\ \mathtt{14},\ \mathit{mc\text{-}rfile}(\mathit{sn})) \\
\qquad\qquad = \ \ \mathit{read\text{-}rn}(\mathtt{32},\ \mathtt{14},\ \mathit{mc\text{-}rfile}(s))) \\
\qquad \land\ \ (\mathit{read\text{-}rn}(\mathtt{32},\ \mathtt{15},\ \mathit{mc\text{-}rfile}(\mathit{sn})) \\
\qquad\qquad = \ \ \mathit{add}(\mathtt{32},\ \mathit{read\text{-}rn}(\mathtt{32},\ \mathtt{15},\ \mathit{mc\text{-}rfile}(s)),\ \mathtt{4})) \\
\qquad \land\ \ (((\mathit{oplen} \le \mathtt{32}) \land \mathit{d2\text{-}7a2\text{-}5p}(\mathit{rn})) \\
\qquad\qquad \Rightarrow\ \ (\mathit{read\text{-}rn}(\mathit{oplen},\ \mathit{rn},\ \mathit{mc\text{-}rfile}(\mathit{sn})) \\
\qquad\qquad\qquad = \ \ \mathit{read\text{-}rn}(\mathit{oplen},\ \mathit{rn},\ \mathit{mc\text{-}rfile}(s)))) \\
\qquad \land\ \ (\mathit{disjoint}(x,\ k,\ \mathit{sub}(\mathtt{32},\ \mathtt{36},\ \mathit{read\text{-}sp}(s)),\ \mathtt{52}) \\
\qquad\qquad \Rightarrow\ \ (\mathit{read\text{-}mem}(x,\ \mathit{mc\text{-}mem}(\mathit{sn}),\ k) \\
\qquad\qquad\qquad = \ \ \mathit{read\text{-}mem}(x,\ \mathit{mc\text{-}mem}(s),\ k))) \\
\qquad \land\ \ (\mathit{iread\text{-}dn}(\mathtt{32},\ \mathtt{0},\ \mathit{sn}) = \mathit{gcd}(\mathit{gcd}(a,\ b),\ c)))\ \textbf{endlet}
\end{array}
$$

THEOREM: *gcd3-is-cd*

$$
\begin{array}{l}
((a\ \textbf{mod}\ \mathit{gcd3}(a,\ b,\ c)) = \mathtt{0}) \\
\land\ \ ((b\ \textbf{mod}\ \mathit{gcd3}(a,\ b,\ c)) = \mathtt{0}) \\
\land\ \ ((c\ \textbf{mod}\ \mathit{gcd3}(a,\ b,\ c)) = \mathtt{0})
\end{array}
$$

THEOREM: *gcd3-the-greatest*

$$
\begin{array}{l}
((a \not\simeq \mathtt{0}) \\
\ \land\ \ (b \not\simeq \mathtt{0}) \\
\ \land\ \ (c \not\simeq \mathtt{0}) \\
\ \land\ \ ((a\ \textbf{mod}\ x) = \mathtt{0}) \\
\ \land\ \ ((b\ \textbf{mod}\ x) = \mathtt{0}) \\
\ \land\ \ ((c\ \textbf{mod}\ x) = \mathtt{0})) \\
\Rightarrow\ \ (\mathit{gcd3}(a,\ b,\ c) \not< x)
\end{array}
$$

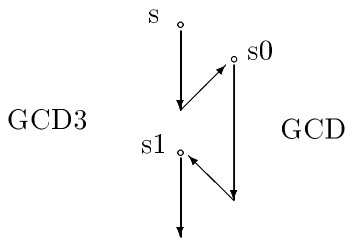

Figure 6.1: How to Use the Correctness of GCD in GCD3

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

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

DEFINITION:

gcd3-s0p (*s*, *a*, *b*, *c*)

= ((*mc-status* (*s*) = `'running`)

∧ *gcd3-loadp* (*s*)

∧ (*mc-pc* (*s*) = GCD-ADDR)

∧ (*read-an* (32, 2, *s*) = GCD-ADDR)

∧ (*rts-addr* (*s*) = *add* (32, GCD3-ADDR, 26))

∧ *ram-addrp* (*sub* (32, 12, *read-sp* (*s*)), *mc-mem* (*s*), 52)

∧ *equal** (*read-an* (32, 6, *s*), *add* (32, *read-sp* (*s*), 20))

∧ (*a* = *iread-mem* (*add* (32, *read-sp* (*s*), 4), *mc-mem* (*s*), 4))

∧ (*b* = *iread-mem* (*add* (32, *read-sp* (*s*), 8), *mc-mem* (*s*), 4))

∧ (*c* = *iread-mem* (*add* (32, *read-sp* (*s*), 12), *mc-mem* (*s*), 4))

∧ (0 < *a*)

∧ (0 < *b*)

∧ (0 < *c*))

DEFINITION:

$$
\begin{array}{l}
\mathit{gcd3\text{-}s1p}\,(s,\ a,\ b,\ c)\\
=\ \ ((\mathit{mc\text{-}status}\,(s) = \texttt{'running})\\
\qquad \wedge\ \ \mathit{gcd3\text{-}loadp}\,(s)\\
\qquad \wedge\ \ (\mathit{read\text{-}an}\,(\texttt{32},\ \texttt{2},\ s) = \textsc{gcd-addr})\\
\qquad \wedge\ \ (\mathit{mc\text{-}pc}\,(s) = \mathit{add}\,(\texttt{32},\ \textsc{gcd3-addr},\ \texttt{26}))\\
\qquad \wedge\ \ \mathit{ram\text{-}addrp}\,(\mathit{sub}\,(\texttt{32},\ \texttt{16},\ \mathit{read\text{-}sp}\,(s)),\ \mathit{mc\text{-}mem}\,(s),\ \texttt{52})\\
\qquad \wedge\ \ \mathit{equal*}\,(\mathit{read\text{-}an}\,(\texttt{32},\ \texttt{6},\ s),\ \mathit{add}\,(\texttt{32},\ \mathit{read\text{-}sp}\,(s),\ \texttt{16}))\\
\qquad \wedge\ \ (\mathit{iread\text{-}dn}\,(\texttt{32},\ \texttt{0},\ s) = \mathit{gcd}\,(a,\ b))\\
\qquad \wedge\ \ (c = \mathit{iread\text{-}mem}\,(\mathit{add}\,(\texttt{32},\ \mathit{read\text{-}sp}\,(s),\ \texttt{8}),\ \mathit{mc\text{-}mem}\,(s),\ \texttt{4}))\\
\qquad \wedge\ \ (\texttt{0} < a)\\
\qquad \wedge\ \ (\texttt{0} < b)\\
\qquad \wedge\ \ (\texttt{0} < c))
\end{array}
$$

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

6.2 Functional Parameters

Taking functions as arguments has long perplexed the programming language community, and the current theoretical solutions to its semantics are subtle. Many formal program verification systems have deliberately avoided considering this issue by simply working on a language subset with this functional parameter feature excluded [5, 15]. As far as we can tell, handling functional parameters in machine-code program proving could be at least as difficult as program proving at higher levels. In this section, we address this important issue in the context of machine-code program proving.

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

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

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

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

0x2320 <max>:           linkw fp,#0
0x2324 <max+4>:         moveml d2-d3,sp@-
0x2328 <max+8>:         movel fp@(8),d3
0x232c <max+12>:        movel fp@(12),d2
0x2330 <max+16>:        movel d2,sp@-
0x2332 <max+18>:        movel d3,sp@-
0x2334 <max+20>:        moveal fp@(16),a0
0x2338 <max+24>:        jsr a0@
0x233a <max+26>:        tstl d0
0x233c <max+28>:        bge 0x2342 <max+34>
0x233e <max+30>:        movel d2,d0
0x2340 <max+32>:        bra 0x2344 <max+36>
0x2342 <max+34>:        movel d3,d0
0x2344 <max+36>:        moveml fp@(-8),d2-d3
0x234a <max+42>:        unlk fp
0x234c <max+44>:        rts
```

First, the correctness of the functional parameter is formalized by the following constraint definition *comp-correctness*. There are three new "undefined" functions *comp-statep*$(s,\ a,\ b)$, *comp-t*$(a,\ b)$, and *comp*$(a,\ b)$ introduced into the logic by *comp-correctness*, each of which has its intended meaning as the precondition on the initial state, the time function, and the behavior function, respectively. The correctness statement is the standard one we have been using throughout this work.

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

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

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

Simultaneously, we introduce the new function symbols *comp-statep*, *comp-t*, and *comp*.

Assuming the correctness of its functional parameter, we can prove the correctness of the binary of `max`. As shown below, *max-comp* (*a*, *b*) is the behavior function; *max-t* (*a*, *b*) is the time function, *max-statep* (*s*, *a*, *b*) is the precondition on the initial state; finally, *max-correctness* gives the correctness of this program.

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

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

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

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

DEFINITION:

$$
\begin{array}{l}
\textit{max-sp}\,(s,\ a,\ b)\\
=\ \ ((\textit{mc-status}\,(s) = \texttt{'running})\\
\qquad \wedge\ \ \textit{evenp}\,(\textit{mc-pc}\,(s))\\
\qquad \wedge\ \ \textit{rom-addrp}\,(\textit{mc-pc}\,(s),\ \textit{mc-mem}\,(s),\ \texttt{46})\\
\qquad \wedge\ \ \textit{mcode-addrp}\,(\textit{mc-pc}\,(s),\ \textit{mc-mem}\,(s),\ \textsc{max-code})\\
\qquad \wedge\ \ (a = \textit{iread-mem}\,(\textit{add}\,(\texttt{32},\ \textit{read-sp}\,(s),\ \texttt{4}),\ \textit{mc-mem}\,(s),\ \texttt{4}))\\
\qquad \wedge\ \ (b = \textit{iread-mem}\,(\textit{add}\,(\texttt{32},\ \textit{read-sp}\,(s),\ \texttt{8}),\ \textit{mc-mem}\,(s),\ \texttt{4}))\\
\qquad \wedge\ \ \textit{ram-addrp}\,(\textit{sub}\,(\texttt{32},\ \texttt{24},\ \textit{read-sp}\,(s)),\ \textit{mc-mem}\,(s),\ \texttt{40}))
\end{array}
$$

CONSERVATIVE AXIOM: *max-state*

$$
\begin{array}{l}
(\textit{max-statep}\,(s,\ a,\ b) \Rightarrow \textit{comp-statep}\,(\textit{stepn}\,(s,\ \textit{max-t0}\,(a,\ b)),\ a,\ b))\\
\wedge\ \ (\textit{max-statep}\,(s,\ a,\ b)\\
\qquad \Rightarrow\ \ \textit{p-disjoint}\,(\textit{add}\,(\texttt{32},\\
\qquad\qquad\qquad\qquad\qquad\quad \textit{read-rn}\,(\texttt{32},\ \texttt{15},\ \textit{mc-rfile}\,(s)),\\
\qquad\qquad\qquad\qquad\qquad\quad \texttt{4294967272}),\\
\qquad\qquad\qquad\qquad\quad \texttt{40},\\
\qquad\qquad\qquad\qquad\quad \textit{stepn}\,(s,\ \textit{max-t0}\,(a,\ b))))\\
\wedge\ \ (\textit{max-statep}\,(s,\ a,\ b) = (\textit{max-sp}\,(s,\ a,\ b) \wedge \textit{comp-loadp}\,(s,\ a,\ b)))
\end{array}
$$

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

THEOREM: *max-correctness*

$$
\begin{array}{l}
\textbf{let}\ \textit{sn}\ \ \textbf{be}\ \ \textit{stepn}\,(s,\ \textit{max-t}\,(a,\ b))\\
\textbf{in}\\
\textit{max-statep}\,(s,\ a,\ b)\\
\Rightarrow\ \ ((\textit{mc-status}\,(\textit{sn}) = \texttt{'running})\\
\qquad \wedge\ \ (\textit{mc-pc}\,(\textit{sn}) = \textit{rts-addr}\,(s))\\
\qquad \wedge\ \ (\textit{read-rn}\,(\texttt{32},\ \texttt{14},\ \textit{mc-rfile}\,(\textit{sn}))\\
\qquad\qquad =\ \ \textit{read-rn}\,(\texttt{32},\ \texttt{14},\ \textit{mc-rfile}\,(s)))\\
\qquad \wedge\ \ (\textit{read-rn}\,(\texttt{32},\ \texttt{15},\ \textit{mc-rfile}\,(\textit{sn}))\\
\qquad\qquad =\ \ \textit{add}\,(\texttt{32},\ \textit{read-sp}\,(s),\ \texttt{4}))\\
\qquad \wedge\ \ (((\textit{oplen} \leq \texttt{32}) \wedge \textit{d2-7a2-5p}\,(\textit{rn}))\\
\qquad\qquad \Rightarrow\ \ (\textit{read-rn}\,(\textit{oplen},\ \textit{rn},\ \textit{mc-rfile}\,(\textit{sn}))\\
\qquad\qquad\qquad =\ \ \textit{read-rn}\,(\textit{oplen},\ \textit{rn},\ \textit{mc-rfile}\,(s))))\\
\qquad \wedge\ \ ((\textit{disjoint}\,(x,\ k,\ \textit{sub}\,(\texttt{32},\ \texttt{24},\ \textit{read-sp}\,(s)),\ \texttt{40})\\
\qquad\qquad \wedge\ \ \textit{max-disjoint}\,(x,\ k,\ s))\\
\qquad\qquad \Rightarrow\ \ (\textit{read-mem}\,(x,\ \textit{mc-mem}\,(\textit{sn}),\ k)\\
\qquad\qquad\qquad =\ \ \textit{read-mem}\,(x,\ \textit{mc-mem}\,(s),\ k)))\\
\qquad \wedge\ \ (\textit{iread-dn}\,(\texttt{32},\ \texttt{0},\ \textit{sn}) = \textit{max-comp}\,(a,\ b)))\ \textbf{endlet}
\end{array}
$$

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

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

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

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

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

{*gt-statep*/*comp-statep*, *gt-t*/*comp-t*, *gt*/*comp*}

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

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

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

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

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

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

We then, in the second step, prove the correctness of the binary of *max*(a, b, gt) by instantiating the theorem *max-correctness*. The functions *max-gt-statep* (*s*, *a*, *b*), *max-gt-t* (*a*, *b*), *max-gt* (*a*, *b*) formalize the precondition, the time function, and the functional behavior of this program, respectively. Finally, the theorem *max-gt-correctness* shows the correctness of the program.

```
DEFINITION:
max-gt-statep (s, a, b)
=  let comp be read-mem (add (32, read-sp (s), 12), mc-mem (s), 4)
   in
   (mc-status (s) = 'running)
   ∧  evenp (mc-pc (s))
   ∧  rom-addrp (mc-pc (s), mc-mem (s), 46)
```

```
     ∧  mcode-addrp (mc-pc (s), mc-mem (s), MAX-CODE)
     ∧  (a = iread-mem (add (32, read-sp (s), 4), mc-mem (s), 4))
     ∧  (b = iread-mem (add (32, read-sp (s), 8), mc-mem (s), 4))
     ∧  evenp (comp)
     ∧  rom-addrp (comp, mc-mem (s), len (GT-CODE))
     ∧  mcode-addrp (comp, mc-mem (s), GT-CODE)
     ∧  ram-addrp (sub (32, 28, read-sp (s)), mc-mem (s), 44) endlet
```

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

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

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

The theorem *max-gt-correctness* above is simply an intantiation of the theorem *max-correctness* by substituting *max-gt-statep* for *max-statep*, *max-gt-t* for *max-t*, *max-t* for *max-comp*, and etc. We recommend that the interested reader study the complete proof script in [53].

6.3 Switch Statement

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

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

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

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

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

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

```
0x23b2 <foo>:      linkw a6,#0
0x23b6 <foo+4>:    movel a6@(8),d0
0x23ba <foo+8>:    movel #4,d1
0x23bc <foo+10>:   cmpl d1,d0
0x23be <foo+12>:   bhi 0x23e4 <foo+50>
0x23c0 <foo+14>:   movew 0x23c8[d0.l*2],d1
0x23c4 <foo+18>:   jmp 0x23c8[d1.w]
0x23c8 <foo+22>:   orb #14,a2
0x23cc <foo+26>:   orb #22,a2@
0x23d0 <foo+30>:   orb #-128,a2@+
0x23d4 <foo+34>:   bra 0x23e4 <foo+50>
0x23d6 <foo+36>:   movel #1,d0
0x23d8 <foo+38>:   bra 0x23e4 <foo+50>
0x23da <foo+40>:   movel #4,d0
0x23dc <foo+42>:   bra 0x23e4 <foo+50>
0x23de <foo+44>:   movel #9,d0
0x23e0 <foo+46>:   bra 0x23e4 <foo+50>
0x23e2 <foo+48>:   movel #16,d0
0x23e4 <foo+50>:   unlk a6
0x23e6 <foo+52>:   rts
```

The correctness proof of this toy program is trivial, and completely automatic with the help of the special-purpose lemmas we have added into the lemma library. The formalization is no different from the other examples: FOO-CODE is the machine code of the program `foo`; *foo-statep* $(s,\ n)$ formalizes the preconditions on the initial state; *foo-t* (n) defines the exact number of instructions to complete this program; and *foo* (n) characterizes the functional behavior of this program. Finally, the theorem *foo-correctness* asserts the correctness of this program.

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

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

DEFINITION:

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

DEFINITION:

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

THEOREM: *foo-correctness*

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

6.4 Embedded Assembly Code

The semantics of high-level programming languages cannot make clear the meaning of embedded assembly code in programs, simply because assembly code is intrinsically machine dependent. By considering directly the binary code of high-level programs after compilation, we do not need to address this semantics issue. Programs and embedded assembly codes are all translated into the formalized world of machine instructions, and their correctness can be studied on the basis of a formal processor semantics. To make our discussion concrete, let us study a very simple example in this section. Our example also demonstrates how easily we can handle embedded assembly code. All we need to know is what the programmer should know when he writes the embedded assembly code.

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

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

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

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

As always, we formalize the preconditions of the initial state, the time function, and the functional behavior of the program, which are given below as the functions *foo-statep*, *foo-t*, and *foo*, respectively.

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

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

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

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

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

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

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

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

Chapter 7

Proving Theorems about the Berkeley Unix C String Library

One of our main goals in defining a formal model for a widely used processor was to study the correctness of real programs executed on that particular processor. The results reported in the preceding chapters have demonstrated the potential to apply our verification methodology to some small programs that are in real use. We now investigate applying our verification system to some small, but real programs.

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

This chapter reports our work on proving mathematical theorems about the Berkeley Unix C String Library. We first give a very brief and informal introduction to the functions in the Berkeley Unix C String Library that we have considered and the mathematical theorems about these functions that we have proved. This should give the reader an overview of this small verification project. To formalize our discussion, we next look into the formal verification of the Berkeley C String Library. We present only the mechanical proofs of a couple of the most interesting and tricky functions in the library: `memmove` and `strstr`. Finally, we discuss the two programming errors we have discovered in studying this C string library. The complete proof script of all the string functions is given in [53].

7.1 The Berkeley Unix C String Library

The Berkeley Unix C string library is intended to be an implementation of the C string library of the ANSI/ISO standard, and is publicly released as part of the Berkeley Unix Operating System.[1] There are twenty-two string functions specified in the ANSI/ISO standard, and we have verified the binary of the Berkeley implementation of twenty-one of them. The binary was generated by the Gnu C Compiler for the MC68020. In this section, we give an informal description of this small verification project. For each of the string functions verified, we provide the formal syntax of the function, a paraphrase of the informal English specification of the ISO standard [27, 43], and an informal description of the theorems we proved about the function.

We adopt an informal, conventional notation to describe the theorems we have proved about these C string functions. We use $s, s1$, and $s2$ to denote strings, $s[i]$ to denote the ith character in the string s, and x' to denote the value of x in the post state. We also informally introduce an predicate $disjoint(s1, s2)$ to assert that the strings $s1$ and $s2$ do not overlap.

Our presentation below of the C string library is highly informal but follows closely the ISO standard [27], where the reader may find a more accurate and verbose English description of these functions. Still more formal is the treatment in [53].

7.1.1 The `memcpy` Function

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

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

Description. The memcpy function copies n characters from the object s2 into the object s1, and returns the value of s1. The behavior of the function is undefined if s1 and s2 overlap.
Theorem. We have: $i < n \Rightarrow s1'[i] = s2[i]$.[2]

7.1.2 The memmove Function

Synopsis. `void *memmove (void *s1, const void *s2, size_t n)`
Description. The memmove function copies n characters from the object s2 into the object s1, and returns the value of s1. The memmove function works correctly on any two objects.
Theorem. We have: $i < n \Rightarrow s1'[i] = s2[i]$.

7.1.3 The strcpy Function

Synopsis. `char *strcpy (char *s1, const char *s2)`
Description. The strcpy function copies the string s2 into the array s1, and returns the value of s1. The behavior of the function is undefined if the strings s1 and s2 overlap.
Theorem. Assuming $disjoint(s1, s2)$, we have:
$j \leq |s2| \Rightarrow s1'[j] = s2[j]$.

7.1.4 The strncpy Function

Synopsis. `char *strncpy (char *s1, const char *s2, size_t n)`
Description. The strncpy function copies at most n characters from the array s2 to the array s1, and returns the value of s1. The behavior of the function is undefined if the strings s1 and s2 overlap.
Theorem. Assuming $disjoint(s1, s2)$, we have:

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

7.1.5 The strcat Function

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

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

Theorem. Assume $disjoint(s1, s2)$, we have:
1. $j < |s1| \Rightarrow s1'[j] = s1[j]$.
2. $|s1| \leq j < |s1| + |s2| \Rightarrow (s1'[j] = s2[j - |s1|]$.

7.1.6 The `strncat` Function

Synopsis. `char *strncat (char *s1, const char *s2, size_t n)`
Description. The `strncat` function appends at most `n` characters from the array `s2` to the end of the string `s1`, and returns the value of `s1`. The behavior of the function is undefined if `s1` and `s2` overlap.
Theorem. Assuming $disjoint(s1, s2)$, we have:
1. $j < |s1| \Rightarrow s1'[j] = s1[j]$.
2. $|s1| \leq j < |s1| + min(|s2|, n) \Rightarrow s1'[j] = s2[j - |s1|]$.

7.1.7 The `memcmp` Function

Synopsis. `int memcmp (const void *s1, const void *s2, size_t n)`
Description. The `memcmp` function compares the first `n` characters of the objects `s1` and `s2`, and returns an integer greater than, equal to, or less than zero, according to the lexical order of the objects `s1` and `s2`.
Theorem. We have:
1. $memcmp(s1, s2, n) = 0 \Rightarrow \forall j < n(s1[j] = s2[j])$.
2. $memcmp(s1, s2, n) \neq 0 \Rightarrow \exists i < n(memcmp(s1, s2, n) = s1[i] - s2[i] \wedge \forall j < i(s1[j] = s2[j]))$
3. $memcmp(s2, s1, n) < 0 \leftrightarrow memcmp(s1, s2, n) > 0$

7.1.8 The `strcmp` Function

Synopsis. `int strcmp (const char *s1, const char *s2)`
Description. The `strcmp` function compares the string `s1` to the string `s2`, and returns an integer greater than, equal to, or less than zero, according to the lexical order of the strings `s1` and `s2`.
Theorem. We have:
1. $strcmp(s1, s2) = 0 \Rightarrow \forall j \leq |s1|(s1[j] = s2[j])$.
2. $strcmp(s1, s2) \neq 0 \Rightarrow \exists i < |s1|(strcmp(s1, s2) = s1[i] - s2[i] \wedge \forall j < i(s1[j] = s2[j]))$
3. $strcmp(s2, s1) < 0 \leftrightarrow strcmp(s1, s2) > 0$

7.1.9 The `strcoll` Function

Synopsis. `int strcoll (const char *s1, const char *s2)`
Description. Since `LC_COLLATE` is not implemented, the function `strcoll` is equivalent to `strcmp`.
Theorem. We have: $strcoll(s1, s2) = strcmp(s1, s2)$.

7.1.10 The `strncmp` Function

Synopsis. `int strncmp (const char *s1, const char *s2, size_t n)`
Description. The `strncmp` function compares at most `n` characters of the arrays `s1` and `s2`, and returns an integer greater than, equal to, or less than zero, according to the lexical order of the arrays `s1` and `s2`.
Theorem. We have:

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

7.1.11 The `strxfrm` Function

Synopsis. `size_t strxfrm (char *s1, const char *s2, size_t n)`
Description. Since `LC_COLLATE` is not implemented, the `strxfrm` function simply copies the string `s2` to the array `s1`, and returns the length of the string `s2`. At most `n` characters are copied to the array `s1`. If `n` is zero, `s1` is permitted to be a null pointer.
Theorem. Assuming $disjoint(s1, s2)$, we have:

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

7.1.12 The `memchr` Function

Synopsis. `void *memchr (const void *s, int c, size_t n)`
Description. The `memchr` function returns a pointer to the first occurrence of `c` in the initial `n` characters of the object `s`, or a null pointer if `c` is not found.
Theorem. We have:

1. $memchr(s, c, n) \neq 0 \Rightarrow s[memchr(s, c, n) - s] = c$.

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

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

7.1.13 The `strchr` Function

Synopsis. `char *strchr (const char *s, int c)`
Description. The `strchr` function returns a pointer to the first occurrence of `c` in the string `s`, or a null pointer if `c` is not found.
Theorem. We have:
1. $strchr(s,c) \neq 0 \Rightarrow s[strchr(s,c) - s] = c$.
2. $strchr(s,c) = 0 \Rightarrow \forall j < |s|, s[j] \neq c$.
3. $j < (strchr(s,c) - s) \Rightarrow s[j] \neq c$.

7.1.14 The `strcspn` Function

Synopsis. `size_t strcspn (const char *s1, const char *s2)`
Description. The `strcspn` function returns the length of the maximum initial segment of the string `s1` which consists entirely of characters not from the string `s2`.
Theorem. We have:
1. $strchr(s2, s1[strcspn(s1,s2)]) \neq 0$.
2. $j < strcspn(s1,s2) \Rightarrow strchr(s2, s1[j]) = 0$

7.1.15 The `strpbrk` Function

Synopsis. `char *strpbrk (const char *s1, const char *s2)`
Description. The `strpbrk` function returns a pointer to the first occurrence in the string `s1` of any character from the string `s2`, or a null pointer if no character from `s2` occurs in `s1`.
Theorem. We have:
1. $strpbrk(s1,s2) \neq 0 \Rightarrow strchr1(s2, s1[strpbrk(s1,s2) - s1]) \neq 0$.
2. $j < (strpbrk(s1,s2) - s1) \Rightarrow strchr1(s2, s1[j]) = 0$.

7.1.16 The `strrchr` Function

Synopsis. `char *strrchr (const char *s, int c)`
Description. The `strrchr` function returns a pointer to the last occurrence of `c` in the string `s`, or a null pointer if `c` is not found.
Theorem. We have:
1. $strrchr(s,c) \neq 0 \Rightarrow s[strrchr(s,c) - s] = c$.

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

7.1.17 The `strspn` Function

Synopsis. `size_t strspn (const char *s1, const char *s2)`
Description. The `strspn` function returns the length of the maximum initial segment of the string `s1` which consists entirely of characters from the string `s2`.
Theorem. We have:
1. $strspn(s1, s2) < |s1| \Rightarrow strchr1(s2, s1[strspn(s1, s2)]) = 0$.
2. $j < strspn(s1, s2) \Rightarrow strchr1(s2, s1[j]) \neq 0$.

7.1.18 The `strstr` Function

Synopsis. `char *strstr (const char *s1, const char *s2)`
Description. The `strstr` function returns a pointer to the first occurrence in the string `s1` of the string `s2`, or a null pointer if the string `s2` is not found.
Theorem. We have:
1. $strstr(s1, s2) \neq 0 \Rightarrow strncmp(strstr(s1, s2), s2, |s2|) = 0$.
2. $strstr(s1, s2) = 0 \Rightarrow \forall j < |s1|(strncmp(s1 + j, s2, |s2|) \neq 0)$.
3. $s1 \leq s < strstr(s1, s2) \Rightarrow strncmp(s, s2, |s2|) \neq 0$.

7.1.19 The `strtok` Function

Synopsis. `char *strtok (char *str1, const char *str2)`
Description. A sequence of calls to the `strtok` function breaks the string `s1` into a sequence of tokens, each of which is delimited by a character from the separator string `s2`. The `strtok` function returns a pointer to the first character of the current token, or a null pointer if there is no token found in the token string. Please see [52, 27] for more detailed descriptions.
Theorem. Let $i(s1)$ be $strspn(s1, s2)$, $j(s1)$ be $strcspn(s1 + i(s1), s2)$, and $last$ be the static variable, we have:
1. $((s1 \neq 0) \wedge (s1[i(s1)] = 0)) \Rightarrow (strtok(s1, s2) = 0) \wedge (last' = 0))$
2. $((s1 \neq 0) \wedge (s1[i(s1)] \neq 0) \wedge (s1[j(s1)] = 0)) \Rightarrow ((strtok(s1, s2) = s1 + i(s1)) \wedge (last' = 0))$
3. $((s1 \neq 0) \wedge (s1[i(s1)] \neq 0) \wedge (s1[j(s1)] \neq 0)) \Rightarrow ((strtok(s1, s2) = s1 + i(s1)) \wedge (last' = s1 + j(s1) + 1) \wedge (s1'[j(s1)] = 0))$
4. $(last = 0) \Rightarrow ((strtok(0, s2) = 0) \wedge (last' = 0))$

5. $((last \neq 0) \wedge (last[i(last)] \neq 0) \wedge (last[j(last)] = 0)) \Rightarrow ((strtok(0, s2) = last + i(last)) \wedge (last' = 0))$

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

7.1.20 The `memset` Function

Synopsis. `void *memset (void *s, const int c, size_t n)`
Description. The `memset` function copies the value of `c` into each of the first `n` characters of the object `s`.
Theorem. We have:
1. $i \leq j < n \Rightarrow s'[j] = ch$.
2. $n \leq j < |s| \Rightarrow s'[j] = s[j]$.

7.1.21 The `strlen` Function

Synopsis. `size_t strlen (const char *s)`
Description. The `strlen` function returns the length of the string `s`.
Theorem. We have:
1. $j < strlen(s) \Rightarrow s'[j] \neq 0$.
2. $s'[strlen(s)] = 0$.

7.2 Proving the String Functions Correct

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

7.2.1 Proving the `memmove` Function Correct

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

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

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

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

#define wsize   sizeof(word)
#define wmask   (wsize - 1)

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

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

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

        if ((unsigned long)dst < (unsigned long)src) {
                /*
                 * Copy forward.
                 */
                t = (int)src;   /* only need low bits */
                if ((t | (int)dst) & wmask) {
                        /*
                         * Try to align operands.  This cannot be done
                         * unless the low bits match.
                         */
                        if ((t ^ (int)dst) & wmask || length < wsize)
                                t = length;
                        else
                                t = wsize - (t & wmask);
                        length -= t;
                        TLOOP1(*dst++ = *src++);
                }
                /*
                 * Copy whole words, then mop up any trailing bytes.
                 */
                t = length / wsize;
                TLOOP(*(word *)dst = *(word *)src; src += wsize; dst += wsize);
                t = length & wmask;
                TLOOP(*dst++ = *src++);
        } else {
                /*
                 * Copy backwards.  Otherwise essentially the same.
                 * Alignment works as before, except that it takes
                 * (t&wmask) bytes to align, not wsize-(t&wmask).
                 */
                src += length;
                dst += length;
                t = (int)src;
                if ((t | (int)dst) & wmask) {
                        if ((t ^ (int)dst) & wmask || length <= wsize)
                                t = length;
                        else
                                t &= wmask;
                        length -= t;
                        TLOOP1(*--dst = *--src);
                }
                t = length / wsize;
                TLOOP(src -= wsize; dst -= wsize; *(word *)dst = *(word *)src);
                t = length & wmask;
                TLOOP(*--dst = *--src);
        }
done:
        return (dst0);
}
```

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

```
0x2550 <memmove>:       linkw fp,#0
0x2554 <memmove+4>:     moveml d2-d4,sp@-
0x2558 <memmove+8>:     movel fp@(8),d3
0x255c <memmove+12>:    movel fp@(16),d2
0x2560 <memmove+16>:    moveal d3,a1
0x2562 <memmove+18>:    moveal fp@(12),a0
0x2566 <memmove+22>:    beq 0x2604 <memmove+180>
0x256a <memmove+26>:    cmpal d3,a0
0x256c <memmove+28>:    beq 0x2604 <memmove+180>
0x2570 <memmove+32>:    bls 0x25bc <memmove+108>
```

```
0x2572 <memmove+34>:    movel a0,d1
0x2574 <memmove+36>:    movel d1,d0
0x2576 <memmove+38>:    orl d3,d0
0x2578 <memmove+40>:    movel #3,d4
0x257a <memmove+42>:    andl d4,d0
0x257c <memmove+44>:    beq 0x25a2 <memmove+82>
0x257e <memmove+46>:    movel d1,d0
0x2580 <memmove+48>:    eorl d3,d0
0x2582 <memmove+50>:    movel #3,d4
0x2584 <memmove+52>:    andl d4,d0
0x2586 <memmove+54>:    bne 0x258e <memmove+62>
0x2588 <memmove+56>:    movel #3,d4
0x258a <memmove+58>:    cmpl d2,d4
0x258c <memmove+60>:    bcs 0x2592 <memmove+66>
0x258e <memmove+62>:    movel d2,d1
0x2590 <memmove+64>:    bra 0x259a <memmove+74>
0x2592 <memmove+66>:    movel #3,d0
0x2594 <memmove+68>:    andl d1,d0
0x2596 <memmove+70>:    movel #4,d1
0x2598 <memmove+72>:    subl d0,d1
0x259a <memmove+74>:    subl d1,d2
0x259c <memmove+76>:    moveb a0@+,a1@+
0x259e <memmove+78>:    subl #1,d1
0x25a0 <memmove+80>:    bne 0x259c <memmove+76>
0x25a2 <memmove+82>:    movel d2,d1
0x25a4 <memmove+84>:    lsrl #2,d1
0x25a6 <memmove+86>:    beq 0x25ae <memmove+94>
0x25a8 <memmove+88>:    movel a0@+,a1@+
0x25aa <memmove+90>:    subl #1,d1
0x25ac <memmove+92>:    bne 0x25a8 <memmove+88>
0x25ae <memmove+94>:    movel #3,d1
0x25b0 <memmove+96>:    andl d2,d1
0x25b2 <memmove+98>:    beq 0x2604 <memmove+180>
0x25b4 <memmove+100>:   moveb a0@+,a1@+
0x25b6 <memmove+102>:   subl #1,d1
0x25b8 <memmove+104>:   bne 0x25b4 <memmove+100>
0x25ba <memmove+106>:   bra 0x2604 <memmove+180>
0x25bc <memmove+108>:   addal d2,a0
0x25be <memmove+110>:   addal d2,a1
0x25c0 <memmove+112>:   movel a0,d1
0x25c2 <memmove+114>:   movel a1,d0
0x25c4 <memmove+116>:   orl d1,d0
0x25c6 <memmove+118>:   movel #3,d4
0x25c8 <memmove+120>:   andl d4,d0
0x25ca <memmove+122>:   beq 0x25ec <memmove+156>
0x25cc <memmove+124>:   movel a1,d0
0x25ce <memmove+126>:   eorl d1,d0
```

```
0x25d0 <memmove+128>:   movel #3,d4
0x25d2 <memmove+130>:   andl d4,d0
0x25d4 <memmove+132>:   bne 0x25dc <memmove+140>
0x25d6 <memmove+134>:   movel #4,d4
0x25d8 <memmove+136>:   cmpl d2,d4
0x25da <memmove+138>:   bcs 0x25e0 <memmove+144>
0x25dc <memmove+140>:   movel d2,d1
0x25de <memmove+142>:   bra 0x25e4 <memmove+148>
0x25e0 <memmove+144>:   movel #3,d4
0x25e2 <memmove+146>:   andl d4,d1
0x25e4 <memmove+148>:   subl d1,d2
0x25e6 <memmove+150>:   moveb a0@-,a1@-
0x25e8 <memmove+152>:   subl #1,d1
0x25ea <memmove+154>:   bne 0x25e6 <memmove+150>
0x25ec <memmove+156>:   movel d2,d1
0x25ee <memmove+158>:   lsrl #2,d1
0x25f0 <memmove+160>:   beq 0x25f8 <memmove+168>
0x25f2 <memmove+162>:   movel a0@-,a1@-
0x25f4 <memmove+164>:   subl #1,d1
0x25f6 <memmove+166>:   bne 0x25f2 <memmove+162>
0x25f8 <memmove+168>:   movel #3,d1
0x25fa <memmove+170>:   andl d2,d1
0x25fc <memmove+172>:   beq 0x2604 <memmove+180>
0x25fe <memmove+174>:   moveb a0@-,a1@-
0x2600 <memmove+176>:   subl #1,d1
0x2602 <memmove+178>:   bne 0x25fe <memmove+174>
0x2604 <memmove+180>:   movel d3,d0
0x2606 <memmove+182>:   moveml fp@(-12),d2-d4
0x260c <memmove+188>:   unlk fp
0x260e <memmove+190>:   rts
```

We follow our formulation described in Chapter 5. The first step is therefore to formalize the precondition on the initial state *memmove-statep* (*s*, *str1*, *n*, *lst1*, *str2*, *lst2*), the time function *memmove-t* (*str1*, *str2*, *n*, *lst1*, *lst2*), and the behavior function *memmove* (*str1*, *str2*, *n*, *lst1*, *lst2*). Only *memmove-statep* is given here. The definition of the other two functions, though quite lengthy, is straightforward.

Definition:
memmove-statep (*s*, *str1*, *n*, *lst1*, *str2*, *lst2*)
= ((*mc-status* (*s*) = 'running)
∧ *evenp* (*mc-pc* (*s*))
∧ *rom-addrp* (*mc-pc* (*s*), *mc-mem* (*s*), 192)
∧ *mcode-addrp* (*mc-pc* (*s*), *mc-mem* (*s*), MEMMOVE-CODE)
∧ *ram-addrp* (*sub* (32, 16, *read-sp* (*s*)), *mc-mem* (*s*), 32)
∧ *ram-addrp* (*str1*, *mc-mem* (*s*), *n*)

$$
\begin{array}{ll}
\wedge & \textit{mem-lst}(\mathtt{1},\ \textit{str1},\ \textit{mc-mem}(s),\ n,\ \textit{lst1}) \\
\wedge & \textit{ram-addrp}(\textit{str2},\ \textit{mc-mem}(s),\ n) \\
\wedge & \textit{mem-lst}(\mathtt{1},\ \textit{str2},\ \textit{mc-mem}(s),\ n,\ \textit{lst2}) \\
\wedge & \textit{disjoint}(\textit{sub}(\mathtt{32},\ \mathtt{16},\ \textit{read-sp}(s)),\ \mathtt{32},\ \textit{str1},\ n) \\
\wedge & \textit{disjoint}(\textit{sub}(\mathtt{32},\ \mathtt{16},\ \textit{read-sp}(s)),\ \mathtt{32},\ \textit{str2},\ n) \\
\wedge & (\textit{str1} = \textit{read-mem}(\textit{add}(\mathtt{32},\ \textit{read-sp}(s),\ \mathtt{4}),\ \textit{mc-mem}(s),\ \mathtt{4})) \\
\wedge & (\textit{str2} = \textit{read-mem}(\textit{add}(\mathtt{32},\ \textit{read-sp}(s),\ \mathtt{8}),\ \textit{mc-mem}(s),\ \mathtt{4})) \\
\wedge & (n = \textit{uread-mem}(\textit{add}(\mathtt{32},\ \textit{read-sp}(s),\ \mathtt{12}),\ \textit{mc-mem}(s),\ \mathtt{4})) \\
\wedge & \textit{uint-rangep}(\textit{nat-to-uint}(\textit{str1}) + n,\ \mathtt{32}) \\
\wedge & \textit{uint-rangep}(\textit{nat-to-uint}(\textit{str2}) + n,\ \mathtt{32}))
\end{array}
$$

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

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

THEOREM: *memmove-correctness*

$$
\begin{array}{l}
\textbf{let}\ \textit{sn}\ \ \textbf{be}\ \ \textit{stepn}(s,\ \textit{memmove-t}(\textit{str1},\ \textit{str2},\ n,\ \textit{lst1},\ \textit{lst2})) \\
\textbf{in} \\
\textit{memmove-statep}(s,\ \textit{str1},\ n,\ \textit{lst1},\ \textit{str2},\ \textit{lst2}) \\
\Rightarrow\ \ ((\textit{mc-status}(\textit{sn}) = \mathtt{'running}) \\
\qquad \wedge\ \ (\textit{mc-pc}(\textit{sn}) = \textit{rts-addr}(s)) \\
\qquad \wedge\ \ (\textit{read-rn}(\mathtt{32},\ \mathtt{14},\ \textit{mc-rfile}(\textit{sn})) \\
\qquad\qquad =\ \ \textit{read-rn}(\mathtt{32},\ \mathtt{14},\ \textit{mc-rfile}(s))) \\
\qquad \wedge\ \ (\textit{read-rn}(\mathtt{32},\ \mathtt{15},\ \textit{mc-rfile}(\textit{sn})) \\
\qquad\qquad =\ \ \textit{add}(\mathtt{32},\ \textit{read-sp}(s),\ \mathtt{4})) \\
\qquad \wedge\ \ ((\textit{d2-7a2-5p}(\textit{rn}) \wedge (\textit{oplen} \le \mathtt{32})) \\
\qquad\qquad \Rightarrow\ \ (\textit{read-rn}(\textit{oplen},\ \textit{rn},\ \textit{mc-rfile}(\textit{sn})) \\
\qquad\qquad\qquad =\ \ \textit{read-rn}(\textit{oplen},\ \textit{rn},\ \textit{mc-rfile}(s)))) \\
\qquad \wedge\ \ ((\textit{disjoint}(x,\ k,\ \textit{sub}(\mathtt{32},\ \mathtt{16},\ \textit{read-sp}(s)),\ \mathtt{32}) \\
\qquad\qquad \wedge\ \ \textit{disjoint}(x,\ k,\ \textit{str1},\ n) \\
\qquad\qquad \wedge\ \ \textit{disjoint}(x,\ k,\ \textit{str2},\ n)) \\
\qquad\qquad \Rightarrow\ \ (\textit{read-mem}(x,\ \textit{mc-mem}(\textit{sn}),\ k) \\
\qquad\qquad\qquad =\ \ \textit{read-mem}(x,\ \textit{mc-mem}(s),\ k))) \\
\qquad \wedge\ \ (\textit{read-dn}(\mathtt{32},\ \mathtt{0},\ \textit{sn}) = \textit{str1}) \\
\qquad \wedge\ \ \textit{mem-lst}(\mathtt{1}, \\
\qquad\qquad\qquad\qquad \textit{str1}, \\
\qquad\qquad\qquad\qquad \textit{mc-mem}(\textit{sn}), \\
\qquad\qquad\qquad\qquad n, \\
\qquad\qquad\qquad\qquad \textit{memmove}(\textit{str1},\ \textit{str2},\ n,\ \textit{lst1},\ \textit{lst2}))) \ \textbf{endlet}
\end{array}
$$

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

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

7.2.2 Proving the strstr Function Correct

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

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

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

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

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

0x2718 <strstr>:        linkw fp,#0
0x271c <strstr+4>:      moveml d2-d3/a2-a3,sp@-
0x2720 <strstr+8>:      moveal fp@(8),a2
```

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

First, the precondition *strstr-statep* (*s*, *str1*, *n1*, *lst1*, *str2*, *n2*, *lst2*), the time function *strstr-t* (*n1*, *lst1*, *n2*, *lst2*), and the behavior function *strstr* (*n1*, *lst1*, *n2*, *lst2*) are defined. Like the preceding example, only *strstr-statep* is given as follows.

CONSERVATIVE AXIOM: *strstr-load*
strstr-loadp (*s*)
= (*evenp* (STRSTR-ADDR)
∧ (STRSTR-ADDR $\in$ **N**)
∧ *nat-rangep* (STRSTR-ADDR, 32)
∧ *rom-addrp* (STRSTR-ADDR, *mc-mem* (*s*), 78)
∧ *mcode-addrp* (STRSTR-ADDR, *mc-mem* (*s*), STRSTR-CODE)
∧ *strlen-loadp* (*s*)
∧ *strncmp-loadp* (*s*)
∧ (*pc-read-mem* (*add* (32, STRSTR-ADDR, 24), *mc-mem* (*s*), 4)
= STRLEN-ADDR)
∧ (*pc-read-mem* (*add* (32, STRSTR-ADDR, 52), *mc-mem* (*s*), 4)
= STRNCMP-ADDR))

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

Definition:

$$\begin{array}{l}
\textit{strstr-statep}\,(s,\ \textit{str1},\ \textit{n1},\ \textit{lst1},\ \textit{str2},\ \textit{n2},\ \textit{lst2}) \\
= \quad ((\textit{mc-status}\,(s) = \texttt{'running}) \\
\qquad \wedge \quad \textit{strstr-loadp}\,(s) \\
\qquad \wedge \quad (\textit{mc-pc}\,(s) = \textsc{strstr-addr}) \\
\qquad \wedge \quad \textit{ram-addrp}\,(\textit{sub}\,(\texttt{32},\ \texttt{48},\ \textit{read-sp}\,(s)),\ \textit{mc-mem}\,(s),\ \texttt{60}) \\
\qquad \wedge \quad \textit{ram-addrp}\,(\textit{str1},\ \textit{mc-mem}\,(s),\ \textit{n1}) \\
\qquad \wedge \quad \textit{mem-lst}\,(\texttt{1},\ \textit{str1},\ \textit{mc-mem}\,(s),\ \textit{n1},\ \textit{lst1}) \\
\qquad \wedge \quad \textit{ram-addrp}\,(\textit{str2},\ \textit{mc-mem}\,(s),\ \textit{n2}) \\
\qquad \wedge \quad \textit{mem-lst}\,(\texttt{1},\ \textit{str2},\ \textit{mc-mem}\,(s),\ \textit{n2},\ \textit{lst2}) \\
\qquad \wedge \quad \textit{disjoint}\,(\textit{str1},\ \textit{n1},\ \textit{sub}\,(\texttt{32},\ \texttt{48},\ \textit{read-sp}\,(s)),\ \texttt{60}) \\
\qquad \wedge \quad \textit{disjoint}\,(\textit{str2},\ \textit{n2},\ \textit{sub}\,(\texttt{32},\ \texttt{48},\ \textit{read-sp}\,(s)),\ \texttt{60}) \\
\qquad \wedge \quad (\textit{str1} = \textit{read-mem}\,(\textit{add}\,(\texttt{32},\ \textit{read-sp}\,(s),\ \texttt{4}),\ \textit{mc-mem}\,(s),\ \texttt{4})) \\
\qquad \wedge \quad (\textit{str2} = \textit{read-mem}\,(\textit{add}\,(\texttt{32},\ \textit{read-sp}\,(s),\ \texttt{8}),\ \textit{mc-mem}\,(s),\ \texttt{4})) \\
\qquad \wedge \quad (\textit{slen}\,(\texttt{0},\ \textit{n1},\ \textit{lst1}) < \textit{n1}) \\
\qquad \wedge \quad (\textit{slen}\,(\texttt{0},\ \textit{n2},\ \textit{lst2}) < \textit{n2}) \\
\qquad \wedge \quad (\textit{n1} \not\simeq \texttt{0}) \\
\qquad \wedge \quad \textit{uint-rangep}\,(\textit{n1},\ \texttt{32}) \\
\qquad \wedge \quad (\textit{n2} \not\simeq \texttt{0}) \\
\qquad \wedge \quad \textit{uint-rangep}\,(\textit{n2},\ \texttt{32}))
\end{array}$$

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

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

Theorem: *strstr-correctness*

$$\begin{array}{l}
\textbf{let}\ \ \textit{sn}\ \ \textbf{be}\ \ \textit{stepn}\,(s,\ \textit{strstr-t}\,(\textit{n1},\ \textit{lst1},\ \textit{n2},\ \textit{lst2})) \\
\textbf{in} \\
\textit{strstr-statep}\,(s,\ \textit{str1},\ \textit{n1},\ \textit{lst1},\ \textit{str2},\ \textit{n2},\ \textit{lst2}) \\
\Rightarrow \quad ((\textit{mc-status}\,(\textit{sn}) = \texttt{'running}) \\
\qquad \wedge \quad (\textit{mc-pc}\,(\textit{sn}) = \textit{rts-addr}\,(s)) \\
\qquad \wedge \quad (\textit{read-rn}\,(\texttt{32},\ \texttt{14},\ \textit{mc-rfile}\,(\textit{sn})) \\
\qquad \qquad = \quad \textit{read-rn}\,(\texttt{32},\ \texttt{14},\ \textit{mc-rfile}\,(s))) \\
\qquad \wedge \quad (\textit{read-rn}\,(\texttt{32},\ \texttt{15},\ \textit{mc-rfile}\,(\textit{sn})) \\
\qquad \qquad = \quad \textit{add}\,(\texttt{32},\ \textit{read-sp}\,(s),\ \texttt{4})) \\
\qquad \wedge \quad ((\textit{d2-7a2-5p}\,(\textit{rn}) \wedge (\textit{oplen} \le \texttt{32})) \\
\qquad \qquad \Rightarrow \quad (\textit{read-rn}\,(\textit{oplen},\ \textit{rn},\ \textit{mc-rfile}\,(\textit{sn})) \\
\qquad \qquad \qquad = \quad \textit{read-rn}\,(\textit{oplen},\ \textit{rn},\ \textit{mc-rfile}\,(s))))
\end{array}$$

$$\begin{array}{l}
\wedge\quad (\mathit{disjoint}(x,\ k,\ \mathit{sub}(\mathtt{32},\ \mathtt{48},\ \mathit{read\text{-}sp}(s)),\ \mathtt{60})\\
\qquad \Rightarrow\quad (\mathit{read\text{-}mem}(x,\ \mathit{mc\text{-}mem}(sn),\ k)\\
\qquad\qquad =\quad \mathit{read\text{-}mem}(x,\ \mathit{mc\text{-}mem}(s),\ k)))\\
\wedge\quad (\mathit{read\text{-}dn}(\mathtt{32},\ \mathtt{0},\ sn)\\
\qquad =\ \ \mathbf{if}\ \mathit{strstr}(\mathit{n1},\ \mathit{lst1},\ \mathit{n2},\ \mathit{lst2})\\
\qquad\qquad \mathbf{then}\ \mathit{add}(\mathtt{32},\ \mathit{str1},\ \mathit{strstr}^{*}(\mathit{n1},\ \mathit{lst1},\ \mathit{n2},\ \mathit{lst2}))\\
\qquad\qquad \mathbf{else}\ \mathtt{0}\ \mathbf{endif}))\ \mathbf{endlet}
\end{array}$$

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

Theorem: *strstr1-thm1*

$$\begin{array}{l}
\mathbf{let}\ j\ \ \mathbf{be}\ \ \mathit{strstr1}(i,\ \mathit{n1},\ \mathit{lst1},\ \mathit{n2},\ \mathit{lst2},\ \mathit{len})\\
\mathbf{in}\\
((j \in \mathbf{N}) \wedge (n = (1 + \mathit{len})))\\
\Rightarrow\quad (\mathit{strncmp2}(j,\ n,\ \mathit{lst1},\ \mathtt{0},\ \mathit{lst2}) = \mathtt{0})\ \mathbf{endlet}
\end{array}$$

Theorem: *strstr-thm2*

$$\begin{array}{l}
(\mathit{lst\text{-}of\text{-}chrp}(\mathit{lst1})\\
\ \wedge\quad \mathit{lst\text{-}of\text{-}chrp}(\mathit{lst2})\\
\ \wedge\quad (j < \mathit{strstr}(\mathit{n1},\ \mathit{lst1},\ \mathit{n2},\ \mathit{lst2}))\\
\ \wedge\quad (\mathit{n2} \not\simeq \mathtt{0}))\\
\Rightarrow\quad (\mathit{strncmp}(\mathit{strlen}(\mathtt{0},\ \mathit{n2},\ \mathit{lst2}),\ \mathit{mcdr}(j,\ \mathit{lst1}),\ \mathit{lst2}) \neq \mathtt{0})
\end{array}$$

Theorem: *strstr-thm3*

$$\begin{array}{l}
(\mathit{lst\text{-}of\text{-}chrp}(\mathit{lst1})\\
\ \wedge\quad \mathit{lst\text{-}of\text{-}chrp}(\mathit{lst2})\\
\ \wedge\quad (\neg\ \mathit{strstr}(\mathit{n1},\ \mathit{lst1},\ \mathit{n2},\ \mathit{lst2}))\\
\ \wedge\quad (j < \mathit{strlen}(\mathtt{0},\ \mathit{n1},\ \mathit{lst1}))\\
\ \wedge\quad (\mathit{n2} \not\simeq \mathtt{0}))\\
\Rightarrow\quad (\mathit{strncmp}(\mathit{strlen}(\mathtt{0},\ \mathit{n2},\ \mathit{lst2}),\ \mathit{mcdr}(j,\ \mathit{lst1}),\ \mathit{lst2}) \neq \mathtt{0})
\end{array}$$

7.3 Programming Errors

Generally, people believe that detecting errors in machine-code programs is hopelessly hard. But our experience with machine-code program proving indicates that finding bugs seems to be no harder than finding proofs in our framework. Discovering programming errors comes naturally as a by-product in the proof process. We add this short section to explain the three programming errors we found in the process of verifying the Berkeley Unix C string library, and to report our experience in finding them.

7.3.1 The Bug in the Berkeley strxfrm Function

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

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

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

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

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

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

7.3.2 The Bug in the Berkeley memmove Function

The second programming error we found was in the Berkeley C string function memmove, which had been detected prior to our work. The error has been corrected in the latest version of BSD4.3.

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

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

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

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

7.3.3 The Bug in Plauger's `strtok` Function

The third programming error we found was not in the Berkeley C string library, but in the `strtok` function of [43].[4] Plauger had detected this error by the time we had reported it to him.

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

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

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

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

Chapter 8

Conclusions

The main goal of the work reported here was to build a powerful proof system on top of Nqthm that could be used to mechanically verify MC68020 machine-code programs. Our experiments with realistic, though very small, machine-code programs demonstrate that we have achieved this goal. Furthermore, the methodologies used and developed in this work provide a general framework for program proving. Our approach can be characterized simply as symbolic execution and theorem proving with an interpreter semantics in a computational logic. We are optimistic that the verification techniques developed in this work can be applied to programs on many different microprocessors and for many different higher-level language compilers.

8.1 The State of the Work

The work described here has three major components:

1. We have formally described a substantial subset of the user mode of the MC68020 microprocessor at the instruction-set level. The formal specification is given as an interpreter in the formal logic of Nqthm.

2. We have developed a mathematical theory for machine-code reasoning, which we have mechanized as a lemma library in the automated reasoning system Nqthm. Each of the lemmas in the library is mechanically checked by Nqthm.

3. We have mechanically verified several dozen MC68020 machine-code programs. Most of the machine-code programs are the object code

produced by the Gnu C compilers from their C counterparts. Primarily to provide concrete evidence that this work is easily applicable to many 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 AKCL Common Lisp compiler for a fixnum GCD program. The programs verified include some of the C functions in Kernighan and Ritchie's book [31], in particular binary search and Quick Sort, a majority voting program, and the Berkeley implementation of the ANSI/ISO standard C string library.

8.2 Future Work

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

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

- The correctness of some moderate-sized piece of software that is in critical use. One good example is the verification of microcontroller programs, an important issue that has been largely ignored by the formal verification community due to the lack of formal methods to handle lower-level code.

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

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

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

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

As a next step, we plan to recast what we have learned and reapply it to another computer architecture. Some issues will be challenging; for example, dealing with the nondeterminism introduced via instructions such as "delayed branch" which may leave the program counter in an indeterminate state during some instructions. We have been investigating the idea of doing similar work on the SPARC [48] and Alpha [47] architectures.

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

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

Appendix A

Syntax Summary

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

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

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

3. Other constants. **t**, **f**, and **nil** are printed in bold. Quoted constants are printed in the ordinary fashion of the Nqthm logic, for example, `'(a b c)` is still printed just that way. `#b001` is printed as $\texttt{001}_2$, `#o765` is printed as $\texttt{765}_8$, and `#xA9` is printed as $\texttt{A9}_{16}$.

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

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

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

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

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

let *var1* **be** *val1*, *var2* **be** *val2* **in** *form* **endlet**.

7. The remaining function symbols that are printed specially are described in the following table.

Nqthm Syntax	Conventional Syntax
`(or x y)`	$x \vee y$
`(and x y)`	$x \wedge y$
`(times x y)`	$x * y$
`(plus x y)`	$x + y$
`(remainder x y)`	$x\ \mathbf{mod}\ y$
`(quotient x y)`	$x \div y$
`(difference x y)`	$x - y$
`(implies x y)`	$x \rightarrow y$
`(member x y)`	$x \in y$
`(geq x y)`	$x \geq y$
`(greaterp x y)`	$x > y$
`(leq x y)`	$x \leq y$
`(lessp x y)`	$x < y$
`(equal x y)`	$x = y$
`(not (member x y))`	$x \notin y$
`(not (geq x y))`	$x \not\geq y$
`(not (greaterp x y))`	$x \not> y$
`(not (leq x y))`	$x \not\leq y$
`(not (lessp x y))`	$x \not< y$
`(not (equal x y))`	$x \neq y$
`(minus x)`	$- x$
`(add1 x)`	$1 + x$
`(nlistp x)`	$x \simeq \mathbf{nil}$
`(zerop x)`	$x \simeq 0$
`(numberp x)`	$x \in \mathbf{N}$
`(sub1 x)`	$x - 1$
`(not (nlistp x))`	$x \not\simeq \mathbf{nil}$
`(not (zerop x))`	$x \not\simeq 0$
`(not (numberp x))`	$x \notin \mathbf{N}$

Acknowledgements

The work reported here is a concise version of my PhD dissertation. Many people contributed to making the work better than I could have done on my own.

First, I want to thank my thesis advisor Bob Boyer. He was an immense source of knowledge, ideas, and inspiration.

I would like to thank my committee members Woody Bledsoe, Don Good, Warren Hunt, Matt Kaufmann, and Bill Schelter.

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

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

Fay Goytowski read our MC68020 formal specification meticulously, and discovered a dozen or so errors.

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

The revision of my dissertation was supported by Digital Equipment Corporation. Cynthia Hibbard edited this version, and made many good suggestions.

Bibliography

[1] Gordon Bell and Allen Newell. The PMS and ISP descriptive systems for computer structures. In *Proceedings of the Spring Joint Computer Conference*. AFIPS Press, 1970.

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

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

[4] Jonathan Bowen. The formal specification of a microprocessor instruction set, technical monograph PRG-60. Technical report, Oxford University, January 1986.

[5] R. S. Boyer and J S. Moore. A verification condition generator for FORTRAN. In R. S. Boyer and J S. Moore, editors, *The Correctness Problem in Computer Science*. Academic Press, London, 1981.

[6] Robert S. Boyer and J Strother Moore. Proving theorems about LISP functions. *Journal of the ACM*, 22(1), 1975.

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

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

[9] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[10] Robert S. Boyer and J Strother Moore. MJRTY - a fast majority vote algorithm. In Robert S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 105–117. Kluwer Academic, 1991.

[11] Robert S. Boyer and Yuan Yu. 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.

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

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

[14] 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.

[15] D. Good, et al. Report on the language GYPSY version 2.0. Technical Report ICSCA-CMP-10, Institute for Computing Science and Computer Applications, University of Texas at Austin, 1978.

[16] A. Falkoff, K. Iverson, and E. Sussenguth. A formal description of system/360. *IBM Systems Journal*, 3(3):198–262, 1964.

[17] James R. Farr. A formal specification of the Transputer instruction set. Master's thesis, Oxford, September 1987.

[18] 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.

[19] 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.

[20] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*. Springer-Verlag, New York, 1979.

[21] Mike Gordon. LCF-LSM, a system for specifying and verifying hardware. Technical Report TR 41, Computer Laboratory, University of Cambridge, September 1983.

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

[23] Warren A. Hunt and B. Brock. A formal HDL and its use in the FM9001 verification. In *Proceedings of the Royal Society*, 1992.

[24] S. Igarashi, R.L. London, and D.C. Luckham. Automatic program verification I: A logical basis and its implementation. Technical Report ISI/RR-73-11, Information Science Institute, USC, 1973.

[25] 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.

[26] Kenneth E. Iverson. *A Programming Language*. John Wiley and Sons, New York, 1962.

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

[28] Matt Kaufmann. A user's manual for an interactive enhancement to the Boyer-Moore theorem prover. Technical Report CLI-19, Computational Logic, Inc., May 1988.

[29] Matt Kaufmann. DEFN-SK: An extension of the Boyer-Moore theorem prover to handle first-order quantifiers. Technical Report CLI-43, Computational Logic, Inc., 1989.

[30] Matt Kaufmann. An integer library for Nqthm. Technical Report CLI Internal 182, Computational Logic, Inc., March 1990.

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

[32] J. C. King. *A Program Verifier.* PhD thesis, Carnegie-Mellon University, 1969.

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

[34] Tim Leonard. Specification of computer architectures: A survey and annotated bibliography. Technical Report 188, University of Cambridge, January 1990.

[35] 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.

[36] 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.

[37] John McCarthy. Computer programs for checking mathematical proofs. In *Recursive Function Theory, Proceedings of a Symposium in Pure Mathematics*, volume V, pages 219–227, Providence, Rhode Island, 1962. American Mathematical Society.

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

[39] John McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Mathematical Aspects of Computer Science, Proc. Symp. Appl. Math., American Mathematical Society*, volume XIX, Providence, Rhode Island, 1967.

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

[41] Ministry of Defence (Britain). Interim defence standard 00-55, requirements for the procurement of safety critical software in defence equipment. Technical report, Directorate of Standardization, Ministry of Defence, Kentigern House 65, Brown Street, Glasgow G2 8EX, Great Britain, 1989.

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

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

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

[45] Phillip Rose. A partial specification of the M68000 microprocessor. Master's thesis, Oxford, September 1987.

[46] D.P. Siewiorek, Gordon Bell, and Allen Newell. *Computer Structures: Principles and examples.* McGraw-Hill, 1982.

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

[48] SPARC International, Inc. *The SPARC Architecture Manual, Version 8.* SPARC International, Inc., Menlo Park, California, 1991.

[49] Chris Torek. Private communication.

[50] 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.

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

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

[53] Yuan Yu. *Automated Proofs of Object Code For a Widely Used Microprocessor.* PhD thesis, University of Texas at Austin, 1992.