AD-A082 461 UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFO--ETC F/G 9/2
MICROCODE VERIFICATION PROJECT.(U)
JAN 80 S D CROCKER, L MARCUS, D VAN-MIEROP F30602-78-C-0008
UNCLASSIFIED RADC-TR-79-353 NL

1 OF 1
AD
A082461

END
DATE FILMED
5-80
DTIC

RADC-TR-79-353
Interim Report
January 1980

MICROCODE VERIFICATION PROJECT

University of Southern California

Stephen D. Crocker
Leo Marcus
Dono van-Mierop

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-79-353 has been reviewed and is approved for publication.

APPROVED: Donald F. Roberts

DONALD F. ROBERTS
Project Engineer

APPROVED: Alan R. Barnum

ALAN R. BARNUM
Asst. Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIS), Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(19) REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER (18) RADC-TR-79-353 / 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) MICROCODE VERIFICATION PROJECT •	5. TYPE OF REPORT & PERIOD COVERED (9) Interim Report • 6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) (10) Stephen D. Crocker / Leo Marcus / Dono van-Mierop	8. CONTRACT OR GRANT NUMBER(s) (15) F30602-78-C-0008
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Southern California * 4676 Admiralty Way Marina del Ray CA 90291 (12) 63 (16)	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62702F 55812007 (17) 2.0
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIS) Griffiss AFB NY 13441 (11)	12. REPORT DATE Jan [redacted]80 13. NUMBER OF PAGES 581
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same	15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

*Information Sciences Institute

RADC Project Engineer: Donald F. Roberts (ISIS)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

ISPS
microcode
program verification
proof checker
simplifier
state deltas
symbolic simulation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The goal of the microcode verification project at ISI is the development of both the theory and tools for verification of microcode. Our strategy has been to push the development of a working system, letting the theoretical issues and the human engineering questions emerge during system development.

The ISPS language is used for encoding our machine descriptions. The State Delta formalism is used for reasoning about the course of the computation..

(Cont'd on reverse)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

~~We are currently building~~ a system to check proofs of microcode validity is currently built. The system is composed of a data base and simplifier that perform some automatic deductions to make checking of the proofsteps manageable, as well as a proof language and a user interface.

A particular computer (the FTSC) is used to establish a focus for the project and to provide a source of examples. This Document ~~Here we~~ report the results of verifying the microcode of the square root instruction of the FTSC, and the results of verifying the complete microcode of a much smaller fictitious example machine.

CONTENTS

1. OVERVIEW

The goal of the microcode verification project at ISI is to develop both the theory and the tools for verification of microcode. While some prior work has been done in this area, notably [Patterson 77, Birman & Joyner 76], the field was (and is) far from closed. Problems exist at every level, from fundamental questions of theory through questions of strategies of system design to problems of integration with other software engineering tools and education of users. Our strategy has been to concentrate on developing a working system, letting the theoretical issues emerge--sometimes painfully--amid system development. We have tried to delay overall consideration of the human engineering questions, but have been forced to consider some of these when it became too difficult to use our own system without improving the interface.

To establish a focus for the project and provide a source of examples, we selected a particular computer, the Fault-Tolerant Spaceborne Computer (FTSC), under development by Raytheon for the Space and Missile Systems Organization (SAMSO) of the Air Force. The FTSC has a number of unusual features related to its design goal for a five-year maintenance-free survival in space. These features appear primarily at the hardware level and in the operating system, however, not in the architecture seen or implemented by the microcode. At the machine language level, the programmer sees a 32-bit machine with 64K of memory, 8 general purpose registers and the usual types of instructions. At the microcode level, the machine is horizontally microprogrammed with 78-bit instructions decoded into 37 different fields. (As of this writing, the machine has been redesigned to have a shorter microinstruction. We have not taken these changes into account in the present work, but will focus on the new design in the next effort.) Documentation of the FTSC is given in [Raytheon Corp 79].

The key criterion for selecting the FTSC is that it is a real machine developed outside our control. We believe that it is possible to verify code for nearly arbitrary machines, irrespective of the techniques used to develop the code. This view differs somewhat from those of other verification researchers, notably [London 77]. To be fair, it is quite clear that much of the labor in the verification task can be reduced if verification and code development are carried out together and if the strategies, practices, and tools

used to develop the code are also geared toward verification. But we view this as a secondary concern and not fundamental to the verification task. In a moment, we will mention where the savings in labor would occur.

We view a microprogram verification system in the following terms. A user prepares formal descriptions of the ***host machine*** and the ***target instruction set***. He also obtains a copy of the microcode that runs on the host machine and allegedly implements the target instruction set. He then prepares a proof that the microcode does indeed behave as desired, and submits all four of these files--host description, microcode, target description, proof--to the verification system, which then examines the target description to determine all aspects of its behavior needing implementation. For each sequence of events that must be implemented, the system *symbolically executes* the microcode according to the rules of the host machine and demonstrates that the required sequence of events does take place.

No system can be quite smart enough to carry out all possible demonstrations completely automatically, so some help may be needed. Some systems operate on the principle that the system should try very hard to succeed on its own and then ask for help after it has tried all possible heuristics. While this approach seems attractive, it has a fundamental drawback. When the system asks the user for help, the user is generally unaware of what the system already has tried to do, what level of detail is needed, or even what problem the system is working on. The underlying difficulty is that the user must have some idea of how the system is constructed and understand how to drive the system. At the same time, we note that the system is really trying to formally document the rationale for each instruction in the microprogram. However, this is just what the programmer had to do himself when he wrote the program. Combining these two observations, we have taken the view that the verification system should be driven by the user, not the other way around. The user should have a complete understanding of what the verification system will and will not do, and the user should drive the verification system toward believing the correctness of the code. In this view, interaction between the system and the user takes the form of a prepared proof, and it becomes meaningful to ask what is the proper language for writing proofs. Wegbreit's

paper [Wegbreit 77] explores this area elegantly for well-structured algorithmic languages. For microcode generated with minimal assembly language tools, different engineering is required, but the basic idea is the same. At the present time, our "proof language" is nothing more than a set of commands to the proofchecker. However, as we gain experience with the system, it becomes clear how to structure these commands into phrases; thus the development of a proof language begins. At the same time, it is worthwhile to ask whether the production of both the microcode and the proof of its correctness can share any tools. The answer must be "yes," but we have not yet considered any specific implementation.

Although we wish our system to be as general and as useful as possible, our present design horizons embody the following limitations:

- The purpose of the microcode must be to implement the instruction set of a computer. This restriction is intended to limit the difficulty of specifying the intended behavior of the microcode. With this restriction, we rule out microcode that is just arbitrary lower level code to implement, say, operating systems, signal processing algorithms, device controllers, etc. This restriction is not really fundamental to our work and, as we shall see, does not quite guarantee that we shall always have a straightforward way to specify the intended behavior of the machine.

- Since we do not yet have sufficient tools to represent or reason about concurrency or time-dependent behavior, we demand that our microcode be written for a sequential machine and that it implement the instruction set of a sequential machine.

- We intend that the result of this research be a demonstrable system with the real possibility that someone other than ourselves should be able to formulate a task and carry it out. We do not intend, however, that the system be efficient, completely robust, smoothly human-engineered, or thoroughly documented. Users of the system should understand the state of development. Their success rate will be higher if they communicate with us before and during any experimentation.

In addition to the caveats above, the system we are building is not yet ready for release.

Carrying out a complete proof may be fairly tedious. Preparation of the formal descriptions often appears to be a straighforward task of encoding the information in the manuals that accompany the machine, but we have noticed that many important details are often omitted from such documents, and others are misdocumented. Programmers developing the microcode come to understand these details and use their knowledge to write or debug their code. If the person writing the formal description is not similarly steeped in the culture of the machine under consideration, a similar learning period will be required.

Writing the proof may be tedious, for three reasons. First, a complete understanding of the code is necessary. The programmer understands the code; the person responsible for verification may not. A period of study may be necessary before any of the proof can be written. Of course, if the programmer were also responsible for preparation of the proof, then the verification would proceed all the faster. Unfortunately, with verification still in the research phase, programmers who build "real" programs are far too busy to spend the extra time required for verification. Also, since verification requires some special knowledge, production programmers may not be skilled in the art of preparing formal descriptions and proofs.

The second difficulty is that the code may be relatively complicated to verify. At the beginning we insisted that it should be possible to verify code even if it were written without knowledge that it would be subjected to verification. (We're assuming, of course, that the code does indeed work!) However, it is equally clear that there are many strategies for writing code and that some of them may be equally good from the programmer's point of view but require very different levels of effort in verification.

The third difficulty is that proofs may be tediously long. We have said that the user must drive the verification system with a proof and that the verification system must proceed so as to give the user a clear idea of what the system is doing. However, a trivial way to build such a system is to make it extremely simple, with the result that proofs will be extremely long and require the user to spend a long time preparing them. In the extreme, this is not permissible; it is necessary to build the system with enough knowledge so the "straightforward" deductions are carried out automatically. There is no possibility that any system can know a "maximum" of knowledge, for there will always

be problems that can be proven with a system, but not proven automatically. At the same time, there is no limit to making a system smarter; we can always go beyond the previous limits and build a next system that understands more than the last. Clear measures of the smartness of one system compared to another do not yet exist, but it is a question that is likely to gain attention as various verification systems are used for larger and larger problems.

As we said earlier, we have restricted our interest to microcode that implements the instruction set of some computer. The intention of this limitation is to make it easy to specify the intended behavior. Unfortunately, this restriction does not quite work. In the description of the host architecture, we have no difficulty in formalizing all aspects of concern, excepting, of course, timing and concurrency. We view the host machine as operating on bitstrings of finite length. The operators for bitstrings are concatenation and selection, logical operations, e.g., AND, OR and NOT, and the simple integer arithmetic operations. At the target level, however, we have not been so fortunate. Bitstrings remain the dominant datatype, and all of the bitstring operators are still required, but new operations exist that are not simply characterized by short descriptions. Floating point arithmetic is the most obvious and extensive area, but some machines have other instructions whose behavior is quite difficult to characterize in terms of bitstrings. Edit and format instructions provide many examples, as do instructions that find the lowest-order or higher-order 1 bit.

The FTSC computer is blessed with the usual complement of floating point instructions; indeed, it even has a floating point square root instruction. On the grounds that avoiding these instructions would trivialize the effort and leave us an undetermined distance from realizing a system capable of verifying real microprograms for real machines, we decided to tackle the floating point arithmetic head on.

We divided the specification of the target machine into two levels. The first is written in the same terms as the host machine description. It is restricted to simple bitstring operators. At this level, the simple target machine instructions, e.g., load, store, integer add, jump, etc., are stated as succinctly as they will ever be stated and no further work is required. The floating point instructions, however, look like short but complicated

algorithms that provide an explicit view of how the words are divided into a mantissa and exponent, how normalization takes place, etc.

For these instructions, we provide a higher level of specification that shows that the result of that algorithmic specification has certain properties. This higher level of specification requires the introduction of the reals, and the properties are stated in terms of the interpretation of the floating point bitstrings as real numbers. For example, the desired property of the square root instruction is that it computes the largest floating point number whose square is not larger than the original number. (The notion of "largest floating point number" requires even a little more; the granularity of the floating point numbers is also an issue.)

In the work to date, we have written a complete specification of the FTSC at both the host and algorithmic target level, but we have not defined the properties required of the floating point instructions except for the square root instruction. We have focused on the square root instruction simply because it seemed to expose all of the issues likely to come up in any other instruction.

The basic plan for verifying the correctness of the microcode thus has two parts. One part is to verify that the microcode running on the host machine implements the algorithmic target level. The second part is to verify that the algorithmic target level has the additional properities desired.

At the present time, we have completed the proof that the algorithmic target description of the square root instruction has the desired property. We have not yet proven similar properties for other instructions, nor have we proven the correspondence between the host machine and the target instruction set, *for the FTSC*. We have, however, created a simple, fictitious machine and carried out a complete proof of the correctness of its microcode. This small machine is called the TOY machine. Both of these proofs are documented in chapter four.

Completion of proofs is one measure of progress, but there is much that precedes the ability to carry out proofs. A sound theoretical basis must exist or be developed and a

functioning proof system must be developed. These activities have consumed the majority of our time and resources.

In chapter two, we discuss the theoretical basis for our proof system and introduce the language we use for expressing the behavior of machines and the properties of programs. In chapter three, we outline the structure of the proof system and give details for selected components.

This work is still in progress. The details of language, structure and capabilities are all evolving.

2. LANGUAGE AND THEORY

In this chapter we discuss the formal basis of and the language we have chosen for both encoding our descriptions of machines and reasoning about the course of computations. Internally, our notation is chosen for its precision and ease of processing, qualities that contrast with the desire for compactness and richness in the languages read and written by humans. Both levels exist, and there must be translation between them. As often happens, subtle and important issues emerge in the translation. At IBM, the difficulties of using two levels of language have been avoided by designing a special-purpose language that is both computationally tractable and not too unwieldy for humans. That language is documented in [Joyner et al. 78].

ISPS

To represent the host and target machines, we have chosen to use the ISPS language. ISPS, a derivative of Bell and Newell's ISP language [Bell and Newell 71], is now in modest use by a number of organizations. Documentation of the current version is given in [Barbacci et al. 77]; the examples in chapter four are written in ISPS.

Descriptions of machines have been written in ISPS for a number of different purposes, including simulation, architecture evaluation, documentation, computer-aided design, and (in variants of ISPS) automatic generation of code generators and assemblers. This variety of activity associated with the language is useful in two ways. On the one hand, the use by large numbers of people improves the possibility that a standard will emerge, that documentation of computers will be more accurate and more complete, and that the task of preparing formal descriptions of the host and target levels of a microprogrammed machine will be carried out by the machine designers instead of by the verification group.

On the other hand, the wide variety of applications using ISPS, each with its own software to process ISPS descriptions, has tended to expose the lack of a precise semantics for the language. As an experiment to gain some leverage on the semantics of ISPS, Pete Alfvin developed a denotational semantic definition of AMDL, an abstract syntax version of ISPS in use at ISI [Alfvin 79].

As we mentioned in the overview, while it may look simple to encode the details of a machine's instruction set in ISPS, it may be tedious in actuality. In the case of the FTSC, a machine under development and redesign, a number of small but important details were either undocumented or misdocumented. We developed simulation tools to execute the descriptions we wrote and used the simulations to execute the diagnostics for the machine at both the host and target levels. In essence, this amounted to a "verification by testing" approach; since the microcode itself was used in some of these tests, it is reasonable to ask if we perturbed the description of the machine in order to make the code work. Stated another way, how do we know that the description of the host machine is an accurate representation of how the hardware really works, and how do we know that the description of the target machine is an accurate representation of how the target machine is supposed to work? There can be no completely satisfactory answers to these questions. The descriptions at both levels must be accepted; they cannot be checked in any rigorous sense within the confines of the microcode verification paradigm. If there exists another description at a higher or lower level, then the corresponding descriptions may be checked against it. However, this merely pushes the problem off one level, and there is no ultimate exemption from a requirement to accept the bottom level description as the way the machine actually works and the top level description as the way the system is supposed to work.

Complete assurance having been denied us, we can ask what lesser assurance is available. By using a language understood by a number of people (in particular by the designers of the machine, the microprogrammers of the machine, and the programmers at the assembly language level) we can have some hope that they all share the same understanding of the machine if they were to depend upon the same descriptions as their reference. This is not yet the case for any machine with any description system, but we see no reason why it could not be.

To complete our discussion of ISPS, we again mention that ISPS does not provide primitives for representing floating point operations; we have had to code them in ISPS as small algorithms. Since the lack of standard notions and designs of floating point arithmetic is a common problem, the choice of another language would not have improved matters.

STATE DELTAS

In order to build a proof system, a formal basis for reasoning about machines is required. Ordinary first-order predicate calculus is often used as a foundation, but it provides no machinery for reasoning about time or situations that change with time.

There are many possible solutions. Ours has been the development of an extension to the first-order predicate calculus by the addition of sentences called ***state deltas***. State deltas were first introduced in [Crocker 77]. For a more formal treatment see [Marcus 79]. To motivate the development of state deltas, we give the observations and decisions that support our formulation.

- It is simple to think in theoretical terms that a computer can be characterized by a transition function that maps state vectors into state vectors. Given an initial state vector and a statement of the transition function, ordinary mathematical tools will provide the machinery for reasoning about successive states of the machine. However, direct use of this approach becomes unwieldy for even the simplest example.

- One of the first difficulties is the description of the state vector. It is quite inconvenient to think of the state vector as a single domain. For all real machines, the state vector is a messy patchwork of various domains. Each of the storage locations in the machine is a piece of the state vector. The primary memory is perhaps the most regular component, but there are many other components. Also, it may be desirable to subdivide the memory into smaller pieces. To deal with this, we use the usual programming practice of assigning names to different ***places***. A place is essentially a component of the state vector. Given the list of places that comprise the state vector, we will not actually need to symbolize the state vector as a single object. We will not even need to know exactly how the components comprise the state vector, e.g., it is not necessary to know if the state vector is represented as a tuple or whether the program counter is, say, the first or second element of that tuple.

- The precise granularity of time is not really of interest. We do not care whether a particular computation takes one or two time steps. Instead, we care that certain states follow one another eventually. Accordingly, we avoid describing individual transitions and describe the effect of multiple transitions instead. The result is quite similar to Manna and Waldinger's ***intermittent assertion*** idea [Manna & Waldinger 78], which is derived from

Burstall's paper [Burstall 74]. We make use of a ***precondition*** and a ***postcondition***, and our state delta encodes the idea that

> if the precondition holds at some point in time,
>
> then there will be a later time at which the postcondition holds.

- While it might be possible to state the behavior of a machine in a single sentence, it would be quite unwieldy. We make use of a collection of state deltas to specify the behavior of a machine. Each state delta defines the behavior of the machine in only particular circumstances. Of course, it is not necessary to cover all possible circumstances; it is perfectly reasonable to leave the behavior of the machine undefined in some cases.

- Most of the components of the state vector are unchanged at each step. Any straightforward description of the transition function would be dominated by simple statements of equality between large sections of the old and new states. To reduce this burden, our formalism encodes the assumption that all of the state remains unchanged except for a list of places in the state vector explicitly named. Accordingly, a state delta has a ***modification list***. The semantics of a state delta includes

> if the precondition holds at some point in time,
>
> then there will come a time at which the new state is the same as the present state except possibly for the values in the places listed in the modification list, and
>
> at that time the postcondition will also hold.

- Even with the implicit assumption that most of the state remains unchanged from one state to another, it may be necessary to include many details in the precondition. Quite often the precondition includes the requirement that much of the present state is identical to a particular prior state. This introduces a third time into the formalism. We have encoded this condition with another list of places, called the ***environment list***. The semantics of state delta are now stated as

> if the contents of the places listed in the environment list are the same at some time t_1 as they were at an earlier time t_0, and
>
> if the precondition is true at time t_1,
>
> then there will be a later time t_2 in which the new state is the

same as the state at time t_1 everywhere except possibly at the places listed in the modification list, and

the postcondition will also hold.

- To simplify our bookkeeping about times and states, we organize all of our thoughts in terms of a current time. In the formulation above, we anchor t_0 to the current time. We can restate the formulation as

  If at some future time t_1 all of the values in the places listed in the environment list are the same as they are now, and

  If the precondition holds at that time,

  then there will come a time t_2 whose values are the same as at time t_1 everywhere except possibly in the places list in the modification list, and

  the postcondition will hold.

- While this formulation is quite close to what we need to support efficient reasoning about places and states, the requirements imposed by the modification and environment lists are more difficult than they look. As stated, it is permitted that the values inside the environment list and outside the modification may change in the interim, as long as they are restored at the end of the interval. We have found it more useful to tighten this requirement so that the values that must be the same at the ends of the time intervals are in fact never changed during the intervals. It turns out that tightening the restriction of the environment and modification lists does not remove any essential power. On the contrary, this new version allows the restricted use of the modal operator "during" to form sentences which are not expressible using only pre- and postconditions. Our formulation is now

  If the values listed in the environment list remain unchanged from now until some future time, and

  If the precondition also holds at that time,

  then at the end of some succeeding time interval during which at most only the values listed in the modification list will have changed, and

  the postcondition will hold.

Note that there is no requirement that values that are unchanged from now until the precondition becomes true remain unchanged when the postcondition becomes true. In other words, it is possible that the same place may be listed in both the environment and modification lists. Later, we will see the use and effect of such an intersection.

The syntactical form of a state delta is

```
(SD     (pre: P)
        (mod: M)
        (env: E)
        (post: Q))
```

where P and Q are usually first order sentences in some language, but may in fact be state deltas themselves, and M is a list of places, as is E. See Chapter 4 for additional examples of state deltas.

Note that the logical implication P implies Q (in a given state) is equivalent to the state delta

```
(SD     (pre: P)
        (mod: )
        (env: OMEGA)
        (post: Q))
```

being true in that state, where OMEGA is a list of all places, or equivalently a single state "containing" all others.

Also note that one state delta may be derived from two others by a kind of case analysis.

If

```
(SD     (pre: P AND P')
        (mod: M)
        (env: E)
        (post: Q))
```

and

```
(SD     (pre: P AND (NOT P'))
        (mod: M)
        (env: E)
        (post: Q))
```

hold in a certain state, then

```
(SD     (pre: P)
        (mod: M)
        (env: E)
        (post: Q))
```

holds in that state.

An important tool is the "dot" operator .R, which when applied to a place R (for "Register") represents the value or contents of that place. Thus a state change entails a redefinition of dot, not a reinterpretation of the place itself.

When dot is used in a state delta it always refers to the contents at the time of the precondition. In order to reference the contents of a place at the time of the postcondition, the symbol # is used. For example,

```
(SD     (pre: .R GTR 0)
        (mod: R)
        (env: )
        (post: #R=.R-1))
```

means that if the value of R is greater than 0, then at some later time the new value will be one less (and nothing changed along the way except for R).

Here is an example of deriving one state delta from another by a form of induction: Assume the contents of places are nonnegative integers. If

```
(SD     (pre: P(.R) AND .R GTR 0)
        (mod: M)
        (env: E)
        (post: P(#R) AND .R GTR #R))
```

holds in a certain state, and in addition if M and E represent disjoint sets of places, then

```
(SD     (pre: P(.R) AND .R GTR 0)
        (mod: M)
        (env: E)
        (post: P(0))
```

holds in that state.

It is obvious how an input-output specification can be stated using state deltas. In the next sections we shall explain how a simulation relation between two programs can be proved using state deltas.

For now let us point out how a set of state deltas can be viewed as a program. Assume that we are given a set of state deltas, ordered in some way, and an "initial" state. The first state delta (according to the above ordering) whose precondition is true in the current state may be "applied", thus transforming the state into that specified by the postcondition (and the modification list). Actually the term "state" should perhaps be replaced by "set of states" since we do not demand that the postcondition completely determine the state; for example, the actual values of some places may not be determined, but rather some properties of these values are known. The components (sentences) of the old state which were dependent on, or "supported by", places in the modification list are removed from the state, and the list of sentences in the postcondition are added to the remaining sentences.

Now the process is repeated in the new state. This process is called symbolic execution.

It is also possible to view a somewhat arbitrary program as a set of state deltas, or to translate a program into state deltas, as is discussed in Section 2.4.

SIMULATION

As stated in the overview, the process of microcode verification can be divided into two parts: the first showing that the Host Machine implements the Target Machine, the second showing that the Target Machine satisfies the Top Level Specification. We shall now discuss the first of these parts.

Let us think on the level of abstraction where both the host and microcode and the target may be considered as programs A_1, A_2. Intuitively, A_1 simulates A_2 if A_1 can "do" anything A_2 can; that is, the state changes due to A_2 are reflected in the state changes that A_1 causes. The state changes for A_1 and A_2 separately are computed using the symbolic execution of the previous section. To prove that A_1 (symbolically) simulates A_2 we need to establish a correspondence between the states of A_1 and those of A_2 such that given two corresponding states, S_2 (for A_2) and S_1 (for A_1), if S_2' is the next state after S_2 arrived at by executing A_2, then the (a) state S_1' corresponding to S_2' can be arrived at by executing A_1 from S_1 (though S_1' need not be the very next state after S_1).

In the system implementation, a state is specified (as in the precondition or postcondition of a state delta) by a list of first order sentences and SDs, and the correspondence between states is specified by a function called MAPPING. Again, recall that "state" as used here is not necessarily a complete description. Thus MAPPING is actually a correspondence between sets of complete states.

TRANSLATION OF ISPS INTO SDS

ISPS is a relatively well known language suitable for machine descriptions. We will see that SD notation is suitable for representing intermediate proof steps, performing symbolic execution, and utilizing the efficiency of the modification list. In order to retain the advantage of ISPS as an input language and SDs as an internal notation, we need to translate ISPS descriptions into SDs.

If we invent a place to represent the internal control state of a machine and we assign a symbolic value to the control place for each statement in an ISPS program, then the program could be represented with a set of SDs, where each SD represents a possible state change. References to control states could be made by including predicates of the form .PC=label in the precondition and postcondition (PC represents the internal control state "program counter"; "label" represents the control value). Representing all the state changes with SDs has two drawbacks: the thread of control that is implicit in the ISPS representation is lost and is encoded explicitly into the precondition and postcondition; the SD notation is different from the familiar ISPS (and somewhat more complicated).

Nested State Deltas

The scheme we are using is motivated by the need to model the control mechanism inside a machine. In an earlier formulation, we modelled the control mechanism as a single variable that took on explicit values. Each precondition and postcondition mentioned the value, e.g., .MicroPC=A312, and this control place was also mentioned in the modification list of every SD. It did not, of course, occur in the environment list. Since the names of the control state values were completely artificial and the explicit appearance in the pre- and postconditions of these equations was very cumbersome, we revised the formulation to an entirely equivalent scheme that simply made implicit use of

the value of control place. The only property of the control place we cared about is that it made some precondition true. By embedding the next SD in the postcondition of the current SD, the next SD is automatically made valid when the current SD is applied ("executed"). Of course, its validity disappears when the control place is changed, so it is necessary that the name of the control place appear in the environment list of the new SD. This is what gives rise to the appearance of the same control place in both the environment and modification lists. Of course, there are some SDs that will not have the control place in the environment list. The tops of loops need to be around forever, and we must resort to using names for the values of the control place at those points. SDs that exit from blocks will not generally have SDs in their postconditions; instead they will set relevant values of the control place.

Instead of describing a program by a *set of SDs* (one for each possible state change) we could describe it with *one SD* that represents the first state change and has a nested SD that represents the rest of the program in its postcondition. During symbolic execution, the process of applying an SD is repeated. The following happens for each SD application: the appropriate state change is made; the nested SD that represents *the rest of the program is added to the current state*; and the SD just applied is removed from the current state if it is supported by the (modified) control place.

The TR Notation

The use of the TR notation is a further compression of the translation from ISPS to SDs. We noticed that it was unnecessary to translate an ISPS description entirely into SDs and then work with the SDs. Instead, we embedded the translation process in the operation of the proof system and carried out just one step of the translation at a time. In essence, we now encode the value of the control place as a formula that tells what to do next. That formula is basically ISPS code, with embellishments to tell us where we are in the code and to keep track of the environment established by ISPS scope rules.

To improve the cumbersome notation of nested SDs to represent the tail of a program, we defined a function called TR that maps an ISPS description into an SD or a set of SDs. We distinguish between ISPS descriptions whose first statement is an assignment statement and those who start with a control change (conditional or unconditional). In

case of an assignment, the TR maps an ISPS program into an SD whose precondition is empty; the modlist includes a control place (MicroPC) and the name of the register that is being assigned to; the envlist includes only MicroPC; the postcondition includes the effect of the assignment and a TR whose parameter is the tail of the ISPS program. In case of a control change, the TR maps an ISPS program into a set of SDs. For each SD, the precondition includes the condition that leads to the control change, the modlist and envlist include MicroPC, and the postcondition includes a TR with the corresponding rest of the ISPS program. The symbolic execution using TRs is very similar to nested SDs, except that the rest of the program is represented as a TR applied to an ISPS description.

Marking ISPS Programs

The set of SDs that represents an ISPS program is not unique. We saw that it ranges from an SD for each ISPS statement to a single SD for the whole program. It depends on the "granularity" that the ISPS description is intended to be broken into. This granularity is specified by special markings of the ISPS description: Every SD that is part of the description of a marked ISPS program must cover a path of execution between two markings.

A control state of an ISPS description is a label or a procedure-entry (that specifies the "rest of the program"). A marking is a special kind of control state. The minimum set of markings needed to specify simulation are the entries and exits of all the procedures. Markings could be added in order to allow more SDs (i.e., a finer granularity). They should be added to break all the loops, for simplicity. Marking should also be added in order to avoid covering the same execution path by more than one SD, for efficiency.

The Translation Process

A marking M_i is a "successor" of M_j if M_i belongs to the set of markings that can be reached by symbolic execution from M_j without visiting any other marking. The translation algorithm generates one SD for each path of execution between two succeeding markings that are reachable from the initial one. The number of SDs generated is determined by the granularity (i.e., the number of markings). When showing

simulation, we will usually use a very fine granularity for the lower level machine (the Host) and a coarser one for the Target. The TR function is used for performing the symbolic execution.

For simplicity we will refer in this paragraph to the translation of the target machine. The control place for the target machine is MacroPC.

The following information is accumulated during the symbolic execution for generating each SD: all the "path conditions" that have to be true in order to reach a successor; the list of places that are modified during execution; the new symbolic state. The new SD covers the path of execution between a marking and its successor, and includes the following: in the precondition the accumulated path condition and .MacroPC="initial label"; in the modlist the accumulated modified places and MacroPC; the envlist is empty; in the postcondition the accumulated symbolic state and .MacroPC=label. A concrete example of translation of an ISPS program is shown in a subsequent chapter.

3. THE SYSTEM

We are building a system to check proofs of microcode validity at the two levels of host to target and target to top-level. Thus essentially the theorems to be proven are of the form: one set of state deltas implies another. The first component is a language in which to write these proofs. Then we need a component to perform some automatic deductions and simplifications to make the checking of the proofsteps a manageable process. Finally we have the interface between the user and the system through which the proof itself is input and the proofchecking can be directed.

PROOF LANGUAGE

Recall that the theorems to be proven are of the form: prove that a given state delta is true in a given state. Typically, for a theorem of the form one state delta implies another, the given state above will be empty except for the assumed state delta. In the case of intermediate lemmas, the state may contain special information about the place values.

The main tools for writing proofs are:

Open(S)	S is a state delta to be proven. Open(S) starts a subordinate proof. The current state is set to the environment and precondition. The intention is that now you will try to symbolically execute with state deltas in the state until the postcondition of S becomes true.
Close()	Finds the previously Opened state delta and checks to see if its postcondition is now true. If so, then this state delta is thereby proved correct and added to the state.
CombineCases(S)	Proves a state delta S from two others by case analysis. An example is given in the introductory section on state deltas.
PerformInduction(S)	Proves a state delta from existing state deltas by a form of induction. We shall not go into the details of the specific induction principle in use at present.

THE SIMPLIFIER

In the following we describe the principles behind some simplifications for expressions in the state delta language. This is not intended to be a complete survey of all possible

simplifications, but rather a representative list of those simplifications found useful in the actual practice of verification, especially the square root algorithm of the FTSC. Thus there is a close correspondence between these simplifications and those actually implemented in the system. Here, though, we describe only the "interesting" ones, and some of these may be stated in different form without mentioning all the cases and specifying the implementation details.

BSC (bitstring constructor) terms

The primitive operations for constructing bitstrings are concatenation a@b, substring selector a<i:j>, and shifts. The definitions of concatenation and shifts are standard. Our conventions for substring selector are that bitstrings are numbered from the right-most bit a<0> to the left-most a<lh(a)-1> where lh(a) is the length of a. Note that we shall allow bitstrings to have variable length. These are called generalized bitstrings. For integer i, j a<i:j> represents the string consisting of bits i down to j of a, that is, a<i>@a<i-1>...@a<j>. If j is greater than i, then this string is nonexistent, and is called EMPTY. If i<0 or i≥lh(a) then a<i> is EMPTY. In the following f(i) and g(i) will be functions attaining integer values at integer values of the argument i. We will occasionally omit mention of i and write just f, g.

A (generalized) substring is a term of the form a<f:g> where a is atomic.

A simplified substring is the EMPTY string or is a substring of the form a<f:g> where

$\forall i\ f(i) < lh(a)$, $\forall i\ g(i) \geq 0$, $\neg\forall i\ f(i) < g(i)$.

Note that when f and g are constants, these conditions become f<lh(a), g≥0, f≥g. Note also that we cannot demand ∀i f(i)≥g(i), since for example a<0:-i> is either EMPTY or a<0> depending on i. From our definition of the semantics of substring, it follows that any substring is equivalent to a simplified substring: a<f:g>= a<min{f, lh(a)-1}, max{g,0}> or EMPTY. If a canonical simplified substring is desired, some standard values of f and g will have to be taken in the case that f(i)<g(i), for example f(i)=0 and g(i)=1.

Length is defined for a (generalized) substring as the following function of i: (Let a, f, and g be functions of i)

```
lh(a<f:g>)(i) = if f(i)≥ lh(a(i)) then lh(a<lh(a)-1:g>(i))
                elseif g(i) < 0 then lh(a<f:0>(i))
                elseif f(i)<g(i) then 0
                else f(i)-g(i)+1.
```

An equivalent closed form is

$$lh(a\langle f:g\rangle) = \min\{lh(a), \max\{\min\{f, lh(a)-1\} - \max\{g, 0\} + 1, 0\}\}$$

This allows the following rewriting: Let 0(f) denote a string of f zeroes.

If a is of the form 0(f)<g:h>, then a ⇒ 0(lh(a)). (1)

A BSC (bitstring constructor) term is any term formed from atomic bitstrings, concatenation, substring, and shifts.

A simplified BSC term is of the form $b_1@b_2@...@b_n$ where $n\geq 1$ and each b_i is a simplified substring.

It can be shown that every BSC term is equivalent to a simplified BSC term. The main simplification rules used in simplifying a BSC term are

$$(a@b)\langle f:g\rangle \Rightarrow a\langle f-lh(b): g-lh(b)\rangle @ b\langle f:g\rangle \quad (2)$$

$$a \text{ SLO } f \Rightarrow 0(\min\{lh(a),-f\})@a\langle lh(a)-f-1:\max\{-f,0\}\rangle@0(\min\{lh(a),f\}) \quad (3)$$

$$a\langle f_1:g_1\rangle\langle f_2:g_2\rangle \Rightarrow a\langle \min\{f_1,f_2+g_1\}:\max\{g_1,g_1+g_2\}\rangle \quad (4)$$

Example Assume lh(a)=4, lh(b)=5, lh(c)=6.

(a@(b@c) SLO 5)<13:3><6:1> ⇒

(0(-5)@(a@(b@c))<9:0>@0(5))<9:4> ⇒

(EMPTY@(a<-2:-11>@(b@c)<9:0>)@0(5))<9:4> ⇒

(b<3:0>@c<9:0>@0(5))<9:4> ⇒

(b<3:0>@c@0(5))<9:4> ⇒

c<4:0>@0(1)

BSA (bitstring arithmetic) terms

All the bitstring addition operators are translated into BITPLUS; BITPLUS is noncarry addition between two bitstrings of equal length. When the sign + appears between bitstrings it will always denote BITPLUS. We also use + for numerical addition, but it is clear from the context which is intended. USVAL(a) is the nonnegative integer represented in binary by the bitstring a.

If b and c are constant bitstrings and USVAL(b)+USVAL(c) < $2^{lh(b)}$, then

$$(a@b)+c \Rightarrow a@(b+c)\langle lh(b)-1:0\rangle \qquad (5)$$

A similar simplification rule holds for c+(a@b). Of course the two sides of 5 are equivalent even if b and c are not constants, but then the right side is not necessarily simpler.

BSR (bitstring relational) terms

There are two main classes of bitstring relations: unsigned value and two's complement. Every unsigned bitstring relation is equivalent to the the corresponding real relation on the USVAL's of its arguments For example, USEQL(a,b) is equivalent to USVAL(a)=USVAL(b). Similarly for two's complement. The simplification of this type of relation will be given in this section. The section on real relations will include (among others) "mixed relations", i.e., those containing both USVAL and TCVAL. TCVAL(a) is the (signed) integer which is the two's complement interpretation of the bitstring a.

Equality

We let $a =_{US} b$ denote USEQL(a,b)=T and similarly for TCEQL. We write = with no subscript if identity between bitstrings is intended.

If $\forall ij\ (f_1(i)<j\leq f_2(i) \vee f_2(i)<j\leq f_1(i) \rightarrow a\langle j\rangle=0)$, then

$$a\langle f_1:g\rangle =_{US} a\langle f_2:g\rangle \qquad (6)$$

If $a_1 =_{US} a_2$ and $b_1 =_{US} b_2$ and $lh(b_1)=lh(b_2)$,or if $b_1 =_{US} b_2\langle lh(b_1)-1:0\rangle$ and $a_1 =_{US} a_2@b_2\langle lh(b_2)-1:lh(b_1)\rangle$, then

$$a_1@b_1 =_{US} a_2@b_2 \tag{7}$$

If $a =_{US} 0$ and $b =_{US} 0$, then

$$a@b =_{US} 0 \tag{8}$$

Of course, there are the obvious generalizations when an arbitrary constant is in place of 0.

If $a_1 =_{US} a_2$ and $b_1 =_{US} b_2$ or $a_1 =_{US} b_2$ and $b_1 =_{US} a_2$, then

$$a_1+b_1 =_{US} a_2+b_2 \tag{9}$$

If $USVAL(a) \geq 2^{lh(a)}-2^f$ or $0 > TCVAL(a) > -2^f-1$, then

$$a\langle f\rangle = 1 \tag{10}$$

If $a\langle f_1:g_1\rangle =_{US} 0$ for some $f_1 \geq f$, $g_1 \leq g$, then

$$a\langle f:g\rangle =_{US} 0 \tag{11}$$

If $a =_{US} b$ and $a\langle lh(a)-1\rangle = b\langle lh(b)-1\rangle$ (or $lh(a)=lh(b)$), then

$$a =_{TC} b \tag{12}$$

If $a\langle f\rangle = a\langle f+1\rangle = \ldots = a\langle lh(a)-1\rangle$, then

$$a\langle f:0\rangle =_{TC} a \tag{13}$$

If $a\langle f+1\rangle = a\langle f\rangle = a\langle f-1\rangle$ and $b\langle f+1\rangle = b\langle f\rangle = b\langle f-1\rangle$, then

$$(a \pm b)\langle f\rangle = (a \pm b)\langle f+1\rangle \tag{14}$$

If $f_1-g_1 = f_2-g_2$, $a\langle f_1':g_1'\rangle =_{US} b\langle f_2':g_2'\rangle$, $f_1' \geq f_1$, $g_1' \leq g_1$, $f_1'-f_1 = f_2'-f_2$, $g_1-g_1' = g_2-g_2'$;

or If $a\langle lh(a)-1:g_1\rangle =_{US} b\langle lh(b)-1:g_2\rangle$, $a\langle f_1+1\rangle = \ldots = a\langle lh(a)-1\rangle = 0$, $b\langle f_2+1\rangle = \ldots = b\langle lh(b)-1\rangle = 0$, then

$$a\langle f_1:g_1\rangle =_{US} b\langle f_2:g_2\rangle \tag{15}$$

Ordering

$$0 \leq_{TC} a \tag{16}$$

If and only if $a\langle lh(a)-1\rangle = 0$.

BSV (bitstring value) terms

If $a\langle lh(a)-1\rangle=0$, then

$$TCVAL(a) \Rightarrow USVAL(a) \tag{17}$$

If $a\langle lh(a)-1\rangle=0$, then

$$USVAL(a) \Rightarrow USVAL(a\langle lh(a)-2:0\rangle) \tag{18}$$

$$TCVAL(a@b) \Rightarrow 2^{lh(b)}*TCVAL(a) + USVAL(b) \tag{19}$$

$$USVAL(a@b) \Rightarrow 2^{lh(b)}*USVAL(a) + USVAL(b) \tag{20}$$

If $lh(a)=lh(b)$, $a\langle f-1\rangle=b\langle f-1\rangle=0$, $a\langle f\rangle=a\langle f+1\rangle=\ldots=a\langle lh(a)-1\rangle$, $b\langle f\rangle=b\langle f+1\rangle=\ldots=b\langle lh(b)-1\rangle$, then

$$TCVAL((a+b)\langle f:0\rangle) \Rightarrow TCVAL(a+b) \tag{21}$$

If $lh(a)=lh(b)$ and $TCVAL(a) + TCVAL(b) \geq 2^{lh(a)-1}$, then

$$TCVAL(a+b) \Rightarrow TCVAL(a)+TCVAL(b)-2^{lh(a)} \tag{22}$$

If $lh(a)=lh(b)$ and $TCVAL(a) + TCVAL(b) < -2^{lh(a)-1}$, then

$$TCVAL(a+b) \Rightarrow TCVAL(a) + TCVAL(b) + 2^{lh(a)} \tag{23}$$

If $lh(a)=lh(b)$ and $-2^{lh(a)-1} \leq TCVAL(a) + TCVAL(b) < 2^{lh(a)-1}$, then

$$TCVAL(a+b) \Rightarrow TCVAL(a) + TCVAL(b). \tag{24}$$

RA (real arithmetic) terms

We list here only the rules concerning RA terms which contain BSV terms.

Let c_1 and c_2 be functions of I (as are the f's and g's). If $c_1, c_2>0$, $f_1 \geq f_2$, $g_1=g_2$, and $\forall I(c_1(I)\neq c_2(I) \Rightarrow g_2(I)>f_2(I))$, then

$$c_1*v(a\langle f_1:g_1\rangle) - c_2*v(a\langle f_2:g_2\rangle) \Rightarrow \tag{25}$$
$$c_1*2^{max(f_2-g_2+1,0)}*v(a\langle f_1:g_1+max(f_2-g_2+1,0)\rangle).$$

Note that we do not demand that $\forall I(f_2 \geq g_2)$.

If $a\langle lh(a)-1\rangle=1$, then

$$TCVAL(a) + 2^{lh(a)} \Rightarrow USVAL(a). \tag{26}$$

RR (real relational) terms

$$TCVAL(a\langle lh(a)-1:n\rangle) \leq 2^{-n} * TCVAL(a) \qquad (27)$$

THE USER INTERFACE

The SD proofchecker is controlled by the Kernel which executes a sequence of low level proofsteps that are submitted to it by the User Interface. The User Interface assists the user in entering the proofsteps in the right format (User Mode); in entering groups of proofsteps that were prepared earlier (Batch Mode); and in entering frequently used sequences of proofsteps (Propose Mode or Symbolic Simulate). The User Interface provides miscellaneous services (in Exec Mode) such as initialization, i.e., assigning a fixed symbolic value to contents of places, probing the status of the proof, redoing proofsteps, entering other modes, etc. Following are some more details about using the User Interface.

The Exec Mode is initiated from INTERLISP by evaluating "(StartExec)"; it is suspended by "QUIT"; and restarted by "(ContinueExec)". In addition to calling the other modes, Exec Mode does the following:

FixLast	allows editing and resubmitting the last proofstep that was submitted to the Kernel (useful in case of failure).
GetISPS	vsubmits an "Open" and sequence of "NewDecomposition"s that setup proofs that involve Symbolic Execution of ISPS programs.
ResetProof	resets the whole proof system.
SetSwitches	sets or resets one of the trace switches.
DisplayState	displays separately the following parts of the current state: State Deltas, Variable Values, General Facts, Coverings, Mappings, Place Map, and Other Predicates.

User Mode assists the user in constructing the correct format of the proofsteps and allows the user to correct the proofstep and okay it before submitting to the Kernel. User Mode uses the full power of the INTERLISP editor and ASKUSER.

A batch is a sequence of proofsteps. It can be edited off-line and generated by the "Transcript" trace. Batch Mode takes the following commands:

PerformNext	submits the next proofstep from the batch.
Doit	submits the next n proofsteps from the batch.
DisplayNext	shows the proofstep that PerformNext would perform.
SkipNext	skips the proofstep that PerformNext would submit.

The User Interface includes two heuristics, one for symbolic execution and one for symbolic simulation. The one for symbolic execution is referred to as ProposeMode, for historical reasons. ProposeMode generates and submits proofsteps until one of the following conditions is met: Control-Y is issued, a "Close" proofstep is submitted, the breakpoint is true in the current state, or it has nothing to propose. The only proofsteps it (currently) tries to propose are "Close" and "ApplySD".

The Symbolic Simulation heuristic generates and submits the following sequence of proofsteps for each SD in the current "GOAL": Open the SD; Apply a mapping; Propose till the state can be mapped again; Apply a mapping; Close. An additional "Close" is submitted to add the "GOAL" to the current state.

4. EXPERIENCE AND EXAMPLES

The bulk of our work has used examples taken from the FTSC. As we outlined in the overview, we have divided the FTSC target description into two levels. One level provides an algorithmic description for the instructions. For the simple instructions, e.g., load, store, and integer arithmetic instructions, this level of description is easy to read and requires no further refinement. However, for the floating point instructions, an algorithmic description of the effect of an instruction is nearly opaque and is useful only to a specialist who needs to track down the detailed results for particular cases. For these instructions, we need to prove that the results guaranteed by the algorithmic description may be understood in terms of some simply stated properties. The square root instruction is the most interesting example in this area, and we have focused most of our attention on proving just the simple property that the effect of the square root instruction *as described by the algorithmic description* does indeed compute the largest floating point number whose square is not greater than the original number. We felt this example would expose the hardest issues first and provide some chance that the rest of the proof would be comparatively easy. We have not yet determined whether this strategy will be successful.

At the same time, we have been concerned that the mechanics of carrying out a complete proof should be well understood. Accordingly, we have hedged our bets a bit and constructed a very small fictitious example of a microcoded machine, written the microcode to implement a simple instruction set for that machine, and prepared a complete proof. We call the machine the "TOY" machine.

This chapter details the proofs for both of these examples. To give the flavor of a complete proof, we present the TOY machine first.

THE TOY MACHINE

The TOY machine is a simple microprogrammed machine. We have provided a formal description of its target instruction set and of its host architecture. We have written the microcode for the host level that implements the target instruction set, and we have specified the states in the host and target levels that correspond to each other.

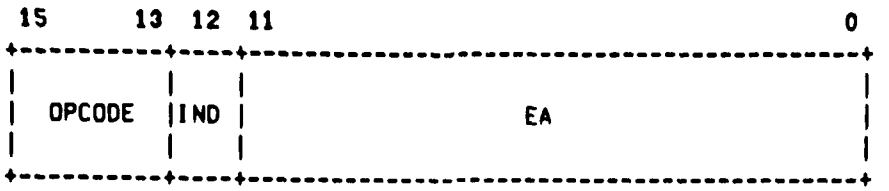
Finally, we have written a set of commands for the proofchecker to guide it toward proving that when the microcode runs on the host machine, it correctly implements the target instruction set. For a problem this simple, the commands to the proofchecker are entirely devoted to setting up the proof. The actual details are carried out completely automatically.

The TARGET Machine

In order to keep this experiment simple, but still deal with a realistic machine, we designed the TARGET machine according to the following requirements:

- 4K-word 16-bit memory
- a 12-bit program counter, a 16-bit accumulator, and a 16-bit instruct register
- infinite indirect addressing
- six possible operations: add, subtract, store, load, skip or negative, jump.

We decided on the following word format:



TOY starts operating by fetching the instruction from location 1 in memory. It proceeds by repeating the cycle of execution and fetching.

Fetching is performed as follows: the machine loads the instruction register from the memory location that the program counter points to; while the indirect bit is set, the 13 least significant bits of the instruction register are overwritten by the contents of the memory location that the effective address (EA) points to; then the program counter is incremented.

The execution performs one of the following operations according to the 3-bit opcode: add MEM[EA] to the accumulator; subtract MEM[EA] from the accumulator; load the accumulator with MEM[EA]; store the contents of the accumulator in MEM[EA]; skip the next operation if the most significant bit of the accumulator is one (negative accumulate); jump to EA.

The precise ISPS description of the TARGET machine was written according to the English description and is shown in Figure 1. The ISPS program is divided into the following declarations: the memory; the registers; the fetching algorithm; the execution algorithm; the main cycle.

The markings we selected in the TARGET machine are the labels MAIN, XFETCH, FLOOP, and EXEC. The paths that the algorithm found were one from MAIN to FETCH, one from FETCH to FLOOP, one from FLOOP to FLOOP, one from FLOOP to EXEC, nine from EXEC to FETCH.

MacroPC is a dummy place that holds the control state (the label) and TinvReg covers the internal registers. The complete set of *SDs that the ISPS to SD algorithm found* is shown in Figure 2. Let us look closer, for example, at the third SD: it describes the path from FLOOP to EXEC which is denoted by .MacroPC=FLOOP in the *pre:* and #MacroPC=EXEC in the *post:*. The *pre:* also includes .IR<12>=0, which is the precondition for taking this particular path The *post:* includes also the new value of PC, .PC+1.

The HOST Machine and the Microcode

The HOST machine is the actual hardware that implements the TOY machine. Because the goal of this experiment is microprogram verification, we chose a microprogrammed HOST. The HOST machine was somewhat tailored to the TARGET, for simplicity, but still much generality and extendability were maintained. The description of the HOST machine explicates all the details of registers, combination circuits, and data paths.

```
TARGET := BEGIN
        !! Memory !!
        MEM[0:4k]<15:0>

        !! Registers !!
        PC<11:0>,                ! program counter
        ACC<15:0>,               ! accumulator
        IR<15:0>,                ! instruction register
        OPCODE<2:0> := IR<15:13>, ! operation code
        EA<11:0> := IR<11:0>     ! effective address

        !! Instruction.Fetching !!
        XFETCH := BEGIN
                IR ← MEM[PC] NEXT
                FLOOP1 := REPEAT
                        FLOOP := DECODE IR<12> =>
                                        BEGIN
                                        0 := LEAVE FLOOP1,
                                        1 := IR<12:0> ← MEM[EA]
                                        END
                NEXT PC ← PC + 1
                END

        !! Instruction.Execution !!
        EXEC := BEGIN
                DECODE OPCODE =>
                        BEGIN
                        0\ADD := ACC ← ACC + MEM[EA],
                        1\SUB := ACC ← ACC - MEM[EA],
                        2\STR := MEM[EA] ← ACC,
                        3\LOAD := ACC ← MEM[EA],
                        4\SKPN := IF ACC<15> => PC ← PC + 1,
                        5\JMP := PC ← EA,
                        6 := NO.OP (),
                        7 := NO.OP ()
                        END
                END

        !! Execution.Cycle !!
        CYCLE[MAIN] := BEGIN
                PC←1 NEXT                ! program counter init
                REPEAT
                        BEGIN
                        XFETCH() NEXT    ! call fetch algorithm
                        EXEC()           ! call execution algorithm
                        END
                END
        END
```

Figure 1. ISPS description of the TARGET machine

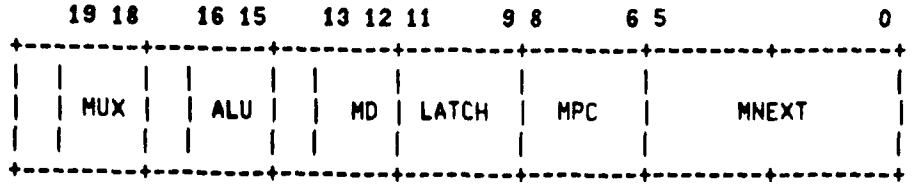
```
((SD (pre: (.MacroPC)=MAIN)
     (mod: TInvReg MacroPC PC)
     (env:)
     (post: #MacroPC=XFETCH #PC=1(12)))
 (SD (pre: (.MacroPC)=XFETCH)
     (mod: TInvReg MacroPC IR)
     (env:)
     (post: #MacroPC=FLOOP #IR=(DOT (WORDS MEM .PC .PC)
 (SD (pre: (.MacroPC)=FLOOP
           (NZEROP (USEQL (DOT (BITS IR 12))
                          0)))
     (mod: TInvReg MacroPC PC)
     (env:)
     (post: #MacroPC=EXEC #PC=(BITPLUS .PC 1(12))))
 (SD (pre: (.MacroPC)=EXEC
           (NZEROP (USEQL (DOT (BITS IR (PAIR 15 13)))
                          0)))
     (mod: TInvReg MacroPC ACC)
     (env:)
     (post: #MacroPC=XFETCH #ACC=(BITPLUS
              .ACC
              (DOT (WORDS MEM (USSUB .IR 11 0)
                              (USSUB .IR 11 0)
 (SD (pre: (.MacroPC)=EXEC
           (NZEROP (USEQL (DOT (BITS IR (PAIR 15 13)))
                          1)))
     (mod: TInvReg MacroPC ACC)
     (env:)
     (post: #MacroPC=XFETCH #ACC=(BITPLUS
              .ACC
              (BITMINUS (DOT (WORDS MEM
                                    (USSUB .IR 11 0)
                                    (USSUB .IR 11 0)
 (SD (pre: (.MacroPC)=EXEC
           (NZEROP (USEQL (DOT (BITS IR (PAIR 15 13)))
                          2)))
     (mod: TInvReg MacroPC
           (WORDS MEM (DOT (BITS IR (PAIR 11 0)
     (env:)
     (post: #MacroPC=XFETCH #(WORDS MEM
                                    (USSUB .IR 11 0)
                                (USSUB .IR 11 0))=(.ACC)))
 (SD (pre: (.MacroPC)=EXEC
           (NZEROP (USEQL (DOT (BITS IR (PAIR 15 13)))
                          3)))
     (mod: TInvReg MacroPC ACC)
     (env:)
     (post: #MacroPC=XFETCH #ACC=(DOT (WORDS MEM
                                             (USSUB .IR 11 0)
                                             (USSUB .IR 11 0)
```

Figure 2. The SD description of the TARGET

```
(SD [pre: (.MacroPC)=EXEC
          (NZEROP (USEQL (DOT (BITS IR (PAIR 15 13)))
                         4))
          (NZEROP (DOT (BITS ACC 15]
    (mod: TInvReg MacroPC PC)
    (env:)
    (post: #MacroPC=XFETCH #PC=(BITPLUS .PC 1(12))))
(SD [pre: (.MacroPC)=EXEC
          (NZEROP (USEQL (DOT (BITS IR (PAIR 15 13)))
                         4))
          ~(NZEROP (DOT (BITS ACC 15]
    (mod: TInvReg MacroPC)
    (env:)
    (post: #MacroPC=XFETCH))
(SD (pre: (.MacroPC)=EXEC
          (NZEROP (USEQL (DOT (BITS IR (PAIR 15 13)))
                         5)))
    (mod: TInvReg MacroPC PC)
    (env:)
    (post: #MacroPC=XFETCH #PC=(USSUB .IR 11 0)))
(SD (pre: (.MacroPC)=EXEC
          (NZEROP (USEQL (DOT (BITS IR (PAIR 15 13)))
                         6)))
    (mod: TInvReg MacroPC)
    (env:)
    (post: #MacroPC=XFETCH))
(SD (pre: (.MacroPC)=EXEC
          (NZEROP (USEQL (DOT (BITS IR (PAIR 15 13)))
                         7)))
    (mod: TInvReg MacroPC)
    (env:)
    (post: #MacroPC=XFETCH))
(SD (pre: (.MacroPC)=FLOOP
          (NZEROP (USEQL (DOT (BITS IR 12))
                         1)))
    (mod: TInvReg MacroPC IR)
    (env:)
    (post: #MacroPC=FLOOP #IR=(USCONC
             (USSUB .IR 15 13)
             (USSUB (DOT (WORDS MEM (USSUB .IR 11 0)
                                    (USSUB .IR 11 0)))
                    12 0]
```

Figure 2. (continued)

We decided to keep the microprogram in a 64-word 21-bit ROM. ROM words contain 21-bit microinstructions with the following format:



The HOST machine (see schematic in Figure 3) includes the following: two memories, STORE, and ROM; registers R1, R2, R3, MAD, MPC (microprogram counter) and MI (microinstruction register); combinational circuits ALU, MD, and MUX; data paths; the scanner. R1 holds the value from the ALU that receives its value either from STORE or from R1; R2 holds the value from R3 or increments its old value; R3 holds the value from MD that receives its value from STORE or R3; MAD holds the value from MUX that receives its value either from R2 or R3.

The HOST repeats the cycle of loading the microinstruction register from the location in ROM that the microprogram counter points to; incrementing the microprogram counter; and scanning the microinstruction and decoding a field at a time. The scanner sends signals that establish data paths and latch values into registers. It also receives values from registers.

The precise ISPS description of the HOST machine is shown in Figure 4, and the description of the ROM in Figure 5. The description of the HOST includes the following declarations: the memories; the registers; the combinational logic; and the execution cycle that fetches and scans the IR. The microprogram is specified as a set of assignments to ROM. The comment in each assignment shows the microinstruction in a mnemonic form: The nonzero fields of each microinstruction are separated by @. The mnemonics correspond to the ones in the DECODE statements in Figure 4. For example,

MUXR3@LMAD@ONIND@10 means that MUX = 3, ALU = 0, MD = 0, LATCH = 6, MPC = 2 and MNEXT = 10.

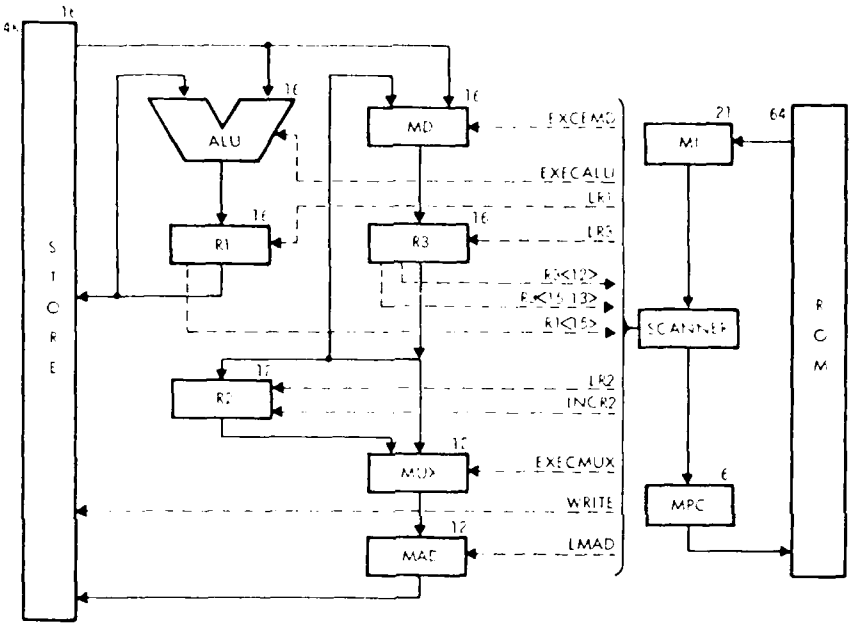


Figure 3. Schematic of the TOY Host

```
HOST := BEGIN

        !** Memory **!
        ROM[0:63]<20:0>,
        STORE[0:4k]<15:0>

        !** Registers **!
        MPC<5:0>,                       ! micro program counter
        MI<20:0>,                       ! micro instruction register
        MNEXT<5:0> := MI<5:0>,          ! next micro instruction
        R1<15:0>,                       ! Accumulator
        R2<11:0>,                       ! Program Counter
        R3<15:0>,                       ! Instruction Register
        MAD<11:0>                       ! memory address

        !** Combinational.Circuits **!
        ALU<15:0>,                      ! arithmetic, logic unit
        MUX<11:0>,                      ! memory address multiplexer
        MD<15:0>                        ! memory data multiplexer

        !** Execution.Cycle **!
        CYCLE[MAIN] := BEGIN
                REPEAT
                BEGIN
                MI ← ROM[MPC] NEXT
                MPC ← MPC + 1
                NEXT
                DECODE MI<19:18> =>
                        BEGIN
                        0 := NO.OP (),
                        1 := NO.OP (),
                        2\MUXR2 := MUX ← R2<11:0>,
                        3\MUXR3 := MUX ← R3<11:0>
                        END NEXT

                DECODE MI<16:15> =>
                        BEGIN
                        0 := NO.OP (),
                        1\ALUNOP := ALU ← STORE[MAD],
                        2\ALUADD := ALU ← R1 + STORE[MAD],
                        3\ALUSUB := ALU ← R1 - STORE[MAD]
                        END NEXT

                DECODE MI<13:12> =>
                        BEGIN
                        0 := NO.OP (),
                        1 := NO.OP (),
                        2\ALL := MD ← STORE[MAD],
                        3\ADD := MD ← R3<15:13>  STORE[MAD]<12:0>
                        END NEXT
```

Figure 4. ISPS description of the HOST

```
        DECODE MI<11:9> =>
                BEGIN
                0 := NO.OP (),
                1\LR1 := R1 ← ALU,
                2\LR2 := R2 ← R3<11:0>,
                3\LR3 := R3 ← MD,
                4\INCR2 := R2 ← R2 + 1,
                5\WRITE := STORE[MAD] ← R1,
                6\LMAD := MAD ← MUX,
                7\INIT := R2 ← 1
                END NEXT

        DECODE MI<8:6> =>
                BEGIN
                0 := NO.OP (),
                1\ONPOS := IF NOT R1<15> => MPC ← MNEXT,
                2\ONIND := IF R3<12> => MPC ← MNEXT,
                3 := NO.OP (),
                4\NXT := MPC ← MNEXT,
                5 := NO.OP (),
                6 := NO.OP (),
                7\ONOP := MPC ← R3<15:13>
                END
        END
        END

END
```

Figure 4. (continued)

```
ROM :=
        BEGIN

        ** Memory **

        ROM[0:63]<20:0>

        ** Execution.Cycle **

        CYCLE(MAIN) :=
                BEGIN
                ROM[0] ← #0201410 ;      ! ALUADD@LR1@NXT@8
                ROM[1] ← #0301410 ;      ! ALUSUB@LR1@NXT@8
                ROM[2] ← #0005410 ;      ! WRITE@NXT@8
                ROM[3] ← #0101410 ;      ! ALUNOP@LR1@NXT@8
                ROM[4] ← #0000416 ;      ! NXT@14
                ROM[5] ← #0002410 ;      ! LR2@NXT@8
                ROM[6] ← #0000410 ;      ! NXT@8
                ROM[7] ← #0000410 ;      ! NXT@8
                ROM[8] ← #2006000 ;      ! FETCH: MUXR2@LMAD
                ROM[9] ← #0023413 ;      ! ALL@LR3@NXT@11
                ROM[10] ← #0033000 ;     ! ADD@LR3
                ROM[11] ← #3006212 ;     ! FLOOP: MUXR3@LMAD@ONIND@10
                ROM[12] ← #0004000 ;     ! EXEC: INCR2
                ROM[13] ← #0000700 ;     ! ONOP@0
                ROM[14] ← #0000110 ;     ! ONPOS@8
                ROM[15] ← #0004410 ;     ! INCR2@NXT@8
                ROM[16] ← #0007410       ! INIT@NXT@8
        NEXT EXEC := NO.OP ()
                END
        END
```

Figure 5. The specification of the Microcode

The first phase of the proof converts the ISPS description of the HOST into a single SD whose *post:* field includes the complete representation of the HOST. This SD is used in the next section as the specification of the control state of the HOST in the mapping. The ISPS description of the microcode is converted to SD notation too.

The current implementation requires that the ISPS description of the HOST consist of a single cycle, for reasons of simplicity. The HOST will indeed usually be a single cycle because it represents hardware. Minor implementation changes will accommodate arbitrary ISPS descriptions of the HOST.

The next section introduces the mapping and the following section explains how the symbolic simulation of the TARGET by the microprogrammed HOST machine is set up and performed.

Relating the TARGET and the HOST

In order to show that one machine simulates another, a relation between the two must be established. The relation addresses control issues and data issues. The control part of the relation specifies all the pairs of control states (in the TARGET and HOST, respectively) that have the following properties: whenever a control state is reached in one machine then the corresponding one is reached in the other machine. Two obvious pairs are the pair of initial states and the pair of final states. A necessary condition for simulation (of terminating machines) is that corresponding initial states always lead to corresponding final states. The data part of the relation specifies the pairs of carriers that should have the same contents whenever a pair of control states is reached. This data relation is called a covering.

The control states in the TARGET machine to be mapped from or to were selected as the set of all the markings. For the particular TOY machine example the following markings were selected: the initial state is MAIN; the top of the main cycle is XFETCH; the infinite fetch loop is broken at FLOOP; the fetch algorithm is separated from the execution algorithm at all the control states in the TARGET map to or from a state described by the top of cycle of the HOST and an additional predicate (usually the value of the microprogram counter).

The top of Figure 6 shows a set of control relations; the first element of each is a marking (represented by an ISPS label) in the TARGET and the rest is a predicate that together with the code of the HOST makes up its control state. The bottom of Figure 6 shows the coverings that specify the relation between registers (or memories) in the TARGET to registers (or memories) in the HOST.

During the first phase of the proof, a set of internal MAPPING records is generated from the concise representation of Figure 6. Figure 7 shows two out of the eight mappings. A MAPPING record has three fields: *from:*, that specifies the control state of either the TARGET or the HOST; *to:*, that specifies the corresponding control state of the other machine; and *map:*, that specifies the covering. The notion of MAPPING records is built into the SD proofchecker and is used in the second phase.

We have described the TARGET, the HOST+microcode, and the relation between them in three forms: English, formal, and a form that can be processed by the SD proofchecker. The first phase of the proof generated the batch of SD commands from the formal descriptions.

Symbolic Simulation

The previous sections presented the TARGET machine, the HOST machine with its microprogram, and the mapping between the machines. This section shows how the proof of simulation of the TARGET by the HOST with respect to the mapping was performed using the SD command batch. The simulation is performed within the state delta symbolic execution framework, thus it is called symbolic simulation.

The SD proof system operates by maintaining a "current state" of the execution, which can be manipulated by opening or closing proofs, or by applying SDs or mappings. A SD is a notation for specifying a segment of execution, either as the "goal" or for changing the current state. A SD has 4 fields: *pre:*, *mod:*, *env:*, and *post:*. When a SD is used to *Open* a proof, then the *pre:* is added to the current state and the *post:* becomes the goal; when it is being "applied", then the *pre:* must be true in the current state, and the effect of the SD is removing from the current state everything that depends on *mod:* and adding *post:*. A MAPPING has three fields: *from:*, *to:*, and *map*. When a mapping is

```
((MAIN (.MPC)=16)
 (XFETCH (.MPC)=8)
 (FLOOP (.MPC)=11)
 (EXEC (.MPC)=13 (.MAD)=(USSUB .R3 11 0)))
```

```
((Covering MEM <<STORE 16 16>>)
 (Covering PC <<R2 12>>)
 (Covering ACC <<R1 16>>)
 (Covering IR <<R3 16>>)
 (Covering MacroPC <<MicroPC 2> <MPC 8>>)
 (Covering HInvReg <<MI 21> <MAD 12> <ALU 16> <MUX 12> <MD 16>>)
 (Covering TInvReg <<HInvReg 22>>))
```

Figure 6. Mapping between TARGET and HOST

```
(MAPPING [from: (.MPC)=11
                (SD (pre:)
                    (mod: MicroPC MI)
                    (env: MicroPC)
                    (post: #MI=(DOT (WORDS ROM .MPC))
                           (TR ((SEQ (USSET MPC $)
                                     (DECODE $ $ $ $ $)
                                     (DECODE $ $ $ $ $)
                                     (DECODE $ $ $ $ $)
                                     (DECODE $ $ $ $ $ $ $ $
                                             $)
                                     (DECODE $ $ $ $ $ $ $ $
                                             $))
                                (REPEAT $)
                                (ProcMark HOST]
         [to: (.MacroPC)=FLOOP)
         (map: (.MEM)=(.STORE)
               (.PC)=(.R2)
               (.ACC)=(.R1)
               (.IR)=(.R3)))
(MAPPING (from: (.MacroPC)=EXEC)
         [to: (.MPC)=13 (.MAD)=(USSUB .R3 11 0)
              (SD (pre:)
                  (mod: MicroPC MI)
                  (env: MicroPC)
                  (post: #MI=(DOT (WORDS ROM .MPC))
                         (TR ((SEQ (USSET MPC $)
                                   (DECODE $ $ $ $ $)
                                   (DECODE $ $ $ $ $)
                                   (DECODE $ $ $ $ $)
                                   (DECODE $ $ $ $ $ $ $ $ $)
                                   (DECODE $ $ $ $ $ $ $ $ $))
                              (REPEAT $)
                              (ProcMark HOST]
         (map: (.STORE)=(.MEM)
               (.R2)=(.PC)
               (.R1)=(.ACC)
               (.R3)=(.IR)))
```

Figure 7. Two of the MAPPING records

"applied", its *from:* must be true in the current state, and the effect of the mapping is adding *to:* and *map:* to the current state.

Figure 8 shows an outline of the batch of commands that drives the proof in the second phase. The first *Open* and *NewDecomposition* declare the memories and registers in the HOST machine. The *pre:* of the second *Open* includes the microcode and the mapping between the TARGET and the HOST. The *post:* of the same command includes the set of SDs that describes the TARGET machine. Executing this command adds the microcode and mapping to the current state and makes the TARGET the "goal". A sequence of seven *NewComposition* commands declares the memories and registers in the TARGET machine and their relation to the places in the HOST. The command *SymSimulate* performs the symbolic simulation according to a heuristic that we have developed.

The SymSimulate command executes a heuristic that drives the symbolic simulation. For each SD in the "goal" do the following: open the SD; apply a mapping from the TARGET to the HOST; symbolically execute (i.e., keep applying SDs) until the state can be mapped back to the TARGET; apply the mapping to the TARGET; close the SD. Finally close the whole "goal".

The combined effect of the two phases of the proof is the generation of a set of SDs from the TARGET using symbolic execution of the TARGET and proving these SDs by using symbolic execution of the HOST and microcode. The rest of the effort is setting up the right relations among the registers and memories and between the HOST and TARGET to assure integrity of the proof. Note that the only input needed is the ISPS description of the TARGET, HOST, and ROM and the concise representation of the mapping between the machines. The rest is done automatically.

THE FTSC

The FTSC was chosen as the real example on which to try out the microcode verification system because it is a general-purpose computer with enough features to thoroughly test the system; in addition, it is still in the development stage, so that successful verification or discovery of bugs would influence the final version.

```
((Open (vars: MicroPC EXP MD MUX ALU MAD R3 R2 R1 MI MPC STORE ROM UNDEFINED
            CLKLOC& LABLOC& ASSLOC& ARRLOC&)
       (SD (pre: (Covering OMEGA
                           <<MicroPC 1> <EXP 44Q> <MD 16> <MUX 12>
                             <ALU 16> <MAD 12> <R3 16> <R2 12> <R1 16>
                             <MI 21> <MPC 6> <STORE 16 10001Q>
                             <ROM 21 100Q> <UNDEFINED 44Q> <CLKLOC& 44Q>
                             <LABLOC& 44Q> <ASSLOC& 44Q> <ARRLOC& 44Q>>))
           (mod: OMEGA)
           (env:)
           (post:)))
 (NewDecomposition (Covering OMEGA
                             <<MicroPC 1> <EXP 44Q> <MD 16> <MUX 12>
                               <ALU 16> <MAD 12> <R3 16> <R2 12> <R1 16>
                               <MI 21> <MPC 6> <STORE 16 10001Q>
                               <ROM 21 100Q> <UNDEFINED 44Q> <CLKLOC& 44Q>
                               <LABLOC& 44Q> <ASSLOC& 44Q> <ARRLOC& 44Q>>>))
 (Open (vars: MicroPC EXP IR ACC PC MEM UNDEFINED CLKLOC& LABLOC& ASSLOC&
            ARRLOC&)
       (SD (pre: (DOT (WORDS ROM 0))=(OCONST 2014100 21)

        .... ((( Specification of microcode )))

                 (MAPPING (from: (.MacroPC)=MAIN)
                          (to: (.MPC)=16
                               (SD (pre:)
                                   (mod: MicroPC MI)
                                   (env: MicroPC)
                                   (post: #MI=(DOT (WORDS ROM .MPC))
                                          (TR ((SEQ (USSET MPC $)
                                                    (DECODE $ $ $ $ $)
                                                    (DECODE $ $ $ $ $)
                                                    (DECODE $ $ $ $ $)
                                                    (DECODE $ $ $ $ $ $ $ $ $ $)
                                                 (DECODE $ $ $ $ $ $ $ $ $))
                                               (REPEAT $)
                                               (ProcMark HOST)
                                   (map: (.STORE)=(.MEM)
                                         (.R2)=(.PC)
                                         (.R1)=(.ACC)
                                         (.R3)=(.IR)))

        .... ((( All mappings )))
```

Figure 8. Outline of the command batch

```
            (mod:)
            (env:)
            (post: (SD (pre: (.MacroPC)=MAIN)
                       (mod: TInvReg MacroPC PC)
                       (env:)
                       (post: #MacroPC•XFETCH #PC=1(12)))

        .... ((( State Delta representation of TARGET )))

(NewComposition (Covering MEM <<STORE 16 16>>))
(NewComposition (Covering PC <<R2 12>>))
(NewComposition (Covering ACC <<R1 16>>))
(NewComposition (Covering IR <<R3 16>>))
(NewComposition (Covering MacroPC <<MicroPC 2> <MPC 6>>))
(NewComposition (Covering HInvReg
                          <<MI 21> <MAD 12> <ALU 16> <MUX 12> <MD 16>>))
(NewComposition (Covering TInvReg <<HInvReg 22>>))
(SymSimulate))
```

Figure 8. (continued)

Some of the characteristics of the FTSC (as of May 1979) are:

- 112 instructions, including integer, floating point, and vector operations
- data formats: fixed point (32-bit, two's complement integer) and floating point (24-bit, two's complement mantissa; 8-bit, two's complement exponent)
- 9 address modes
- 8 general-purpose registers (that serve as accumulators, index registers, or address pointers) and 8 working registers
- 10 interrupt levels
- 61K of addressable program memory

The first step in the verification process is writing the formal host and target machine descriptions in ISPS. Ideally, the designer of the machine would write the formal description along with the informal description ("user's manual"). In lieu of this, the writer of the formal descriptions must submit them to the designer for "description verification" (that this is really the machine informally described in the manual) before proceeding with the proof. In addition, the writer of the formal descriptions may discover "bugs" (inconsistencies or incompleteness) in the user manual.

As explained earlier, we consider the total problem of microcode verification as consisting of two parts: the proof that the host machine with its microcode implements the target machine (as described in a language containing only those operations available to the host) and the proof that the target machine, instruction by instruction, satisfies some higher level specification. For example, the target machine description of the integer multiply and divide instructions, and all floating point instructions, would most likely consist of an algorithm using the host machines operations of shifting, testing, adding, XORing, etc. The higher level specification would be that these instructions do in fact find the product, quotient, etc. to a given precision. The instruction definitions given in the user manual, which are largely English, are most likely those instructions needing this second level of proof.

All of our work to date on the verification of the FTSC has been concerned with the step from the target to the higher specification. This seemed a wise choice, since we knew that at the start of our project the FTSC host machine design was not finalized, although the target machine would remain more or less the same. In addition, many aspects of the system had to be developed before a truly large example could be attacked.

The particular instruction chosen was square root. Square root was chosen because of the relative compactness of its algorithmic description in the target machine, and the wide difference between the algorithm and its higher specification. Although the second-level verification has nothing to do with the microcode or the host machine, one characteristic making it less than general program verification is that the data types used in the target and higher level descriptions are usually restricted to be bitstrings and integers in the target, and values of bitstrings and reals in the higher level. Thus we used the square root instruction as a testing ground for developing the automatic simplification of expressions in these data types.

The status of our work on the square root algorithm is that the simplifier is able to handle automatically all the derivations needed to complete the proof of correctness. Smoothing the user interface and gracefully setting up the induction needed for the loop remain to be done.

It is hoped that many of the special simplification rules adopted in proving the square root will also be useful in the other proofs of higher level correctness.

Square Root Proof

In this section we give the ISPS version of the algorithm that constitutes the FTSC target machine description of the floating point square root instruction (SRTF). See Figure 9. This description of the algorithm was written on the basis of the microcode flowchart, which is derived directly from the host description and the microcode. Then we show the derivations the simplifier is able to accomplish automatically in proving that SRTF finds the square root to within a certain accuracy.

```
SRTF:=

        BEGIN
        DECODE AMODE=>(W0←W1←GPXRA,W0←W1←MD) NEXT
        IF W0 LSS 0=>(OVFF←1 NEXT LEAVE SRTF) NEXT
        IF W0<31:8> EQL 0=>(GPXRB←"80 NEXT LEAVE SRTF) NEXT
        W0<31:8>←W0<31:8> SL0 1 NEXT
        W0<7:0>←0 NEXT
        DECODE W1<0>=>
                BEGIN
                0:=(GPXRB←W0<31:30> NEXT
                   W0←W0 SL0 2  NEXT
                   W1<31:8>←0   NEXT

                   W1<7:0>←W1<7>@W1<7:1> ),
                1:=(GPXRB←W0<31> NEXT
                   W0←W0 SL0 1 NEXT
                   W1<31:8>←0 NEXT
                   EXPOUT←W1<7>@W1<7:0> + 1 NEXT
                   W1<7:0>←EXPOUT<7:0> NEXT
                    W1<7:0>←W1<7>@W1<7:1> NEXT
                   IF EXPOUT<8> XOR EXPOUT<7>=>W1<7:0>←#100 )
                END

        NEXT
        SUM←GPXRB-1 NEXT
        GPXRB←SUM<29:0>@W0<31:30> NEXT
        COUNTER←0  NEXT
        SLOOP:=
                REPEAT
                BEGIN
                COUNTER←COUNTER+1 NEXT
                W0<31:8>←W0<31:8> SL0 2 NEXT
                DECODE SUM<31>=>
                BEGIN
                0:= (W1<31:8>←2*W1<31:8> + 1 NEXT
                        IF COUNTER EQL 23=>(LEAVE SLOOP) NEXT
                        W2←4*W1<31:8>  + 1 NEXT
                        SUM←GPXRB-W2 NEXT
                        GPXRB←SUM<29:0>@W0<31:30>),
                1:= (W1<31:8>←2*W1<31:8>  NEXT
                        IF COUNTER EQL 23=>(LEAVE SLOOP) NEXT
                        W2←4*W1<31:8> + 3 NEXT
                        SUM←GPXRB+W2 NEXT
                        GPXRB←SUM<29:0>@W0<31:30>)

                END
                END
        NEXT
        GPXRB←W1
        END
```

Figure 9. ISPS description of the square root algorithm

Let us "talk through" the algorithm now: The first line decides if the input is to be from register GPXRA or register MD. If the input is negative, the algorithm is terminated with overflow flag set. If the input is 0, the algorithm is terminated with output register GPXRB set to the floating representation of 0. From here on the algorithm splits into two parts: the calculation of the new exponent and the calculation of the new mantissa. The exponent calculation splits depending on whether it is even or odd. If the old value is even, the new exponent is half the old value. If the old value is odd, it is made even by adding 1 and shifting the mantissa accordingly (in the even case the mantissa is shifted two bits; in the odd case, only one bit). Now the new value is half the old value (with a check for exponent overflow thrown in). The mantissa is now calculated by a variation of the longhand high school square root algorithm. The mantissa is shifted two bits at a time through the loop 23 times. The loop has two branches according to the sign of the "remainder," the register SUM.

The theorem which expresses the correctness of SRTF is

Theorem: If FL(INPUT)=x≥0, then SRTF terminates with $FL(OUTPUT)^2 \le x \le FL^+(OUTPUT)^2$.

If FL(INPUT)<0, then SRTF terminates with OVFF=1.

Explanation of notation: FL(R) is the value of the bitstring R as a floating point number in the FTSC format: 24 leftmost bits coding two's complement fractional mantissa and rightmost 8 bits coding two's complement exponent. INPUT is either the register GPXRA or MD, depending on AMODE. OUTPUT is the register GPXRB. $FL^+(R)$ is floating successor to FL(R), i.e.,

$$FL^+(R) = (TCVAL(R\langle 31:8\rangle)+1) * 2^{TCVAL(R\langle 7:0\rangle)-23}.$$

Letting $MAN(R) = TCVAL(R\langle 31:8\rangle) * 2^{-23}$ and $EXP(R) = TCVAL(R\langle 7:0\rangle)$, it is sufficient to prove

(I) If EXP(INPUT)=e is even and $MAN(INPUT)*2^{46}=ARG$, then SRTF terminates with $2*EXP(OUTPUT)=e$ and $(MAN(OUTPUT)*2^{23})^2 \le ARG \le (MAN(OUTPUT)*2^{23}+1)^2$, and

(II) If EXP(INPUT)=e is odd and $MAN(INPUT)*2^{45} = ARG$, then SRTF terminates with $2*EXP(OUTPUT)=e+1$ and $(MAN(OUTPUT)*2^{23})^2 \le ARG \le (MAN(OUTPUT)*2^{23}+1)^2$.

So the proof is carried out by

(1) symbolically executing through the end of the exponent calculation for even and odd input exponent, and proving the relevant parts of (I) and (II) at that point (note that OUTPUT is assigned the contents of working register W1 at the end of SRTF);

(2) at that point, for even input exponent,

$$MAN(INPUT)*2^{46} = USVAL(GPXRB\langle 1:0\rangle @W0\langle 31:10\rangle)*2^{22} = ARG,$$

and for odd exponent,

$$MAN(INPUT)*2^{45} = ARG.$$

Thus to complete both (I) and (II) it remains to show that

CLAIM: $TCVAL(OUTPUT\langle 31:8\rangle)^2 \leq ARG \leq TCVAL(OUTPUT\langle 31:8\rangle+1)^2$.

Here is where we use induction to prove loop invariants that lead to a proof of the CLAIM. Let R_i denote the contents of R after i times through the loop, that is, the last contents before COUNTER changes from i to i+1.

The CLAIM is proved from

SUBCLAIM: For $1 \leq i \leq 23$, $USVAL(W1_i\langle 30:8\rangle)^2 \leq int(ARG*2^{2i-46}) \leq (USVAL(W1_i\langle 30:8\rangle)+1)^2$.

(The actual calculation with the integer part function int is done by noting that if X=USVAL(R), then $int(X*2^{-k}) = USVAL(R\ SRO\ k)$.)

The CLAIM is proved from the SUBCLAIM by taking i=23. The SUBCLAIM is implied by the first three of the following loop invariants for $1 \leq i \leq 22$. ((H1) is shown here for the case of even exponent only).

(H1) $(2*USVAL(W1_i\langle 30:8\rangle)+1)^2 + TCVAL(SUM_i) = USVAL(a\langle 30:8\rangle @0(23)\ SRO\ 44-2i)$

(H2) $TCVAL(SUM_i) \leq 4*USVAL(W1_i\langle 30:8\rangle) + 2$

(H3) $-TCVAL(SUM_i) \leq 4*USVAL(W1_i\langle 30:8\rangle) + 1$

(H4) $W0_i =_{US} (a\langle 28:8\rangle @0(11)\ SLO\ 2i)$

(H5) $W1_I\langle 31:I+8\rangle =_{US} 0(24-I)$

(H6) $W2_I\langle 31:I+2\rangle =_{US} 0(30-I)$

(H7) $SUM_I\langle 29:0\rangle =_{US} GPXRB_I\langle 31:2\rangle$

(H8) $SUM_I =_{TC} GPXRB_I\langle 31:2\rangle$

(H9) $GPXRB_I\langle 1:0\rangle =_{US} W0_I\langle 31:30\rangle$

Thus we prove that if (H1)-(H9) are true for $1 \leq I \leq 21$, then they are true for I+1. Additional induction hypotheses ((H4)-(H9) were found to facilitate the proof of (H1)-(H3)). Then we prove that if the SUBCLAIM is true for $1 \leq I \leq 22$, then it is true for I+1. The simplifier automatically carries out these deductions.

5. CONCLUSIONS

PLANNED EXTENSIONS

The basic theoretical work for proofs of correctness of sequential microcode is reasonably complete, and a preliminary system for carrying out proofs has been built and exercised. Within the scope of the present work, the following extensions are planned.

Proof Language

The system is divided into a user interface and a rigorous proofchecker. In the present implementation, the user interface knows too little about the direction of the proof. In a proof by cases, for example, the separate cases are presented to the proofchecker, then combined. It is possible to declare the intended result in a superior proof, but no use is made of this information in either the user interface or the kernel.

We now see that the user interface can interpret a simple goal-oriented language. For a proof by cases, the user would specify what lemma is to be proven and would specify that the form of the proof is to be by cases with a given predicate. Room for specifying the details of each subproof would also exist, but the packaging of the separate proofs would be carried out by the proofchecker. In the present system, a proof by cases now looks like the following:

```
(Open P)
(Open P and C)
  <details of the proof of the first case>
(Close P and C)
(Open P and not C)
  <details of the proof of the second case>
(Close P and not C)
(CombineCases)
(Close P)
```

In many instances, the proof of each case may be carried out automatically. In the present system, a ProposeMode statement is required. We can eliminate the "obvious" proofs if we use null lists where proof details are permitted. Combined with the automatic setup and packaging of compound proofs, the proof above might become the following:

(Prove P (Cases C <room for details of positive subcase>
<room for details of negative subcase>)

Similar savings would result in proofs by induction. Some of the savings are not apparent from proof sketches like the ones above. The lemmas are often quite lengthy. Even with the lemma suppressed from the Close command, the current system requires three copies of the main lemma, one for the statement of the lemma in the main proof, and two more for the subcase proofs. The compressed form requires only one appearance of the lemma. In addition, the compressed form is much more readable and, we hope, more writable.

Editing

The present system permits only limited editing of the proof. Using the structured proofs illustrated above, it should be possible to edit a proof quite freely and have the proof restarted from the last point it was changed.

Efficiency

The present system is fairly slow. With a little experimentation, it has become clear that a lot of time is wasted in the simplifier. The simplifier has evolved through an accretion process, and is due for a complete redesign. We have also studied Derek Oppen's work (see, for example, [Nelson and Oppen 78]), and it appears reasonable to use his simplifier for parts of the system. His simplifier is carefully crafted and should be much faster.

FUTURE CONSIDERATIONS

A number of ideas for logical next steps have emerged, though these are beyond the scope of the present effort.

Floating Point Arithmetic Specification

Floating point arithmetic needs to be characterized precisely. Notation to describe the intended precision of the results and relationship between floating point operations and the corresponding abstract operations on the reals would materially reduce the size of

the target machine description and remove the need for proving a separate set of constraints.

Some of the initial work has been done by Brown and others [Brown 77, Brown 78, Wijngaarden 64, Kahan 77a, Kahan 77b].

Timing

Performance characteristics play a large part in the design of host machines and in the design of the microcode. However, to date no work has been done to characterize the running time of microcode. Proofs of running time limits should be reasonably straightforward, but work is needed on the specifications.

Concurrency

Essentially no work has been done on correctness proofs of truly concurrent microcode. The present work requires a sequentialized model of the host and target machines. Extensions to the basic theory will be required to model concurrency.

REFERENCES

[Alfvin 79] Peter W. Alfvin, A Formal Definition of AMDL, Master's thesis, University of California, Los Angeles, 1979.

[Barbacci et al. 77] Mario R. Barbacci, Gary E. Barnes, Roderic G. Cattell, and Daniel P. Siewiorek, The ISPS Computer Description Language, 1977. (Unpublished paper from Carnegie-Mellon University.)

[Bell and Newell 71] Gordon C. Bell and Allen Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, New York, 1971.

[Birman & Joyner 76] A. Birman and William H. Joyner, "A Problem-Reduction Approach to Proving Simulation Between Programs," *IEEE Transactions on Software Engineering* SE-2, (2), June 1976, 87-96.

[Brown 77] W. S. Brown, *A Realistic Model of Floating Point Arithmetic*, Bell Laboratories, Technical Report 58, 1977.

[Brown 78] W. Stanley Brown and Stuart I. Feldman, *Environment Parameters and Basic Functions for Floating-Point Computation*, Bell Laboratories, Technical Report 72, 1978.

[Burstall 74] R. M. Burstall, "Program Proving as Hand Simulation with a Little Induction," In *Information Processing 74*, pp. 308-312, North-Holland, Amsterdam, 1974.

[Crocker 77] Stephen D. Crocker, *State Deltas: A Formalism for Representing Segments of Computation*, Ph.D. thesis, University of California, Los Angeles, 1977.

[Gordon 79] Michael J. C. Gordon, *The Denotational Description of Programming Languages: An Introduction*, Springer-Verlag, New York, 1979.

[Joyner et al. 78] William H. Joyner Jr., William C. Carter, and Daniel Brand, "Using Machine Descriptions in Program Verification," In *Information Technology: Proceedings of the 3rd Jerusalem Conference on Information Technology (JCIT3)*, pp. 515-522, North-Holland, Amsterdam, 1978.

[Kahan 77a] W. Kahan and B. N. Parlett, *Can You Count on Your Calculator?*, University of California, Berkeley, Memorandum No. UCB/ERL M77/21, 1977.

[Kahan 77b] W. Kahan, *And Now for Something Completely Different: The Texas Instruments SR-52*, University of California, Berkeley, Memorandum No. UCB/ERL M77/23, 1977.

[London 77] Ralph L. London, "Perspectives on Program Verification," In Raymond T. Yeh (ed.), *Current Trends in Programming Methodology*, pp. 151-172, Prentice-Hall, 1977.

[Manna & Waldinger 78] Zohar Manna and Richard Waldinger, "Is 'Sometime' Sometimes Better than 'Always'?," *Communications of the ACM* 21, (2), February 1978, 159-172.

[Marcus 79] Leo Marcus, State Deltas that Remember: a System of Describing State Changes, 1979. (Submitted for publication.)

[Nelson and Oppen 78] C. G. Nelson and D. C. Oppen, *Simplification by Cooperating Decision Procedures*, Stanford University, CS Report No. STAN-CS-78-652, 1978. (AI Memo AIM311.)

[Patterson 77] David Patterson, *Verification of Microprograms*, Ph.D. thesis, University of California, Los Angeles, 1977.

[Raytheon Corp 79] Raytheon Corp., *Brassboard Fault Tolerant Spaceborne Computer (BFTSC)*, Raytheon Corp., Technical Report ER79-4135, May 1979.

[Teitelman 78] Warren Teitelman, *Interlisp Reference Manual*, Xerox Palo Alto Research Center, 1978.

[Wegbreit 77] Ben Wegbreit, "Constructive Methods in Program Verification," *IEEE Transactions on Software Engineering* SE-3, (3), May 1977, 193-209.

[Wijngaarden 64] A. van Wijngaarten, "Numerical Analysis as an Independent Science," *BIT* 6, 1964, 66-81.