A. Birman

On Proving Correctness of Microprograms

Abstract: This paper describes the results of an investigation in proving the correctness of microprograms. The vehicle used is the S-machine, which is a very simple "paper" computer. The approach to the proof of correctness is based on formally defining the machine-instruction level and the microprogramming level of the given machine, and then showing that these "interfaces" are equivalent through the use of a concept called algebraic simulation.

Introduction

This paper presents the results of an investigation [1] into proving the correctness of microprograms. The vehicle chosen for this investigation is the S-machine, a variation on a computer described by Gear [2] for which a simulation package is available [3]. This very simple computer is being used for teaching microprogramming in the Education Department at IBM Poughkeepsie.

The question of whether a microprogram is correct leads to another question: "What is the microprogram supposed to do?" In trying to answer the latter question we realize that the functions of a microprogram are related to the operation of other parts of the computer: control store, registers, etc. Therefore the microprogram together with other parts of the computer constitute an "interface" or a "level" of the system. The concept of interface is an old and natural one. When designing a computer we start by specifying the machine at the highest interface. As more decisions are made on how to implement this level, a new and more detailed interface is drawn up. For example, the documentation for the S-machine consists of

1. a description of the machine-instruction level, the so-called "principles of operation" manual, and
2. a description of the microinstruction level, or the "microprogramming manual."

The language used in descriptions of this kind is informal; the English language description is complemented by graphic illustrations. When considering the problem of correctness, however, we have to formalize our definition in a way that will enable us to prove whatever we claim to be correct. If we want to describe an interface of a given system together with all the data structures and processes related to this interface, the formal counterpart of the usual English-language description can be an *abstract machine*.

The concept of an abstract machine was first formulated in the field of programming language semantics [4-9]. An abstract machine which interprets a program in a given language is considered as one way of specifying the semantics of that language. Our use of the concept of abstract machine is similar to that of Lee [10]. Work in formalization of system definitions can also be found in Falkoff [11], Bell and Newell [12], and others.

A language that has been used in defining abstract machines is the Vienna Definition Language (VDL) [7-9, 13-15]. In order to have a useful definition language for machine interfaces we had to supplement the VDL with basic operators and predicates. Since we are dealing with objects like registers, flip-flop circuits, memories, etc., which are represented by binary vectors and arrays, we selected as basic operators a small number of APL operators. Using this VDL/APL language we define two abstract machines: *abstract machine* S, which relates to the machine instruction interface, and *abstract machine* μS, for the microinstruction level (The abstract machine S is not to be confused with the "S-machine" which is the name of our computer.). The microprogram itself is part of μS.

The abstract machine μS is in some sense equivalent to S. This kind of equivalence is found in the concept called "algebraic simulation of one program by another," described by Milner [16]; abstract machines are themselves programs or, more precisely, "abstract programs."

The essence of the idea of simulation can be informally described as follows. A simulation of P by P' implies that anything computed by P can be computed by P'. The simulation should have certain properties: 1) For any program P there should exist a simulation of P by P, and 2) if there exists a simulation of P by P_1 and also a simulation of P_1 by P_2, then there should exist a simulation of P by P_2. We display such a concept and prove that it has these properties.

In this paper we develop an approach to proving the correctness of microprograms and we apply it to the S-machine. The proof of correctness for the S-machine implementation consists of the following steps:

1. Definition of abstract machine S corresponding to the machine-instruction level.
2. Definition of abstract machine μS for the microinstruction level.
3. Determining the desired simulation relation R.
4. Proving that μS simulates S with respect to R.

The paper is organized as follows. The second section contains a brief description of the S-machine. For further details the reader is referred to [2, 3]. In the third section we introduce the VDL/APL language used in defining abstract machines, and we present the definition of the abstract machine S which is discussed through examples; the complete definitions of S and μS are found in Appendixes A and B, respectively. References [7-9, 13, 15] provide further details on VDL and [17] on APL. In the fourth section the concept of simulation is defined. An example is given to illustrate how one proves simulation. In the fifth section the simulation relation for (S, μS) is postulated and the proof of simulation which involves the method of inductive assertions [18] is given. Appendix C contains the microcode. Summary, discussions and directions for further research are the subject of the last section.

• *Notation*

Given two sets D and D', R is a relation if $R \subseteq D \times D'$. An element of R is denoted by $(d, d') \in R$ or $d \overset{R}{\rightarrow} d'$. The inverse of R is R^{-1}: $R^{-1} = \{(d', d) | (d, d') \in R\}$. The relation R is a (partial) function if for each $d \in D$ there is at most one $d' \in D'$ such that $(d, d') \in R$; in this case we also write $R : D \rightarrow D'$, and $R(d) = d'$ or $Rd = d'$ for $(d, d') \in R$. If R is a function and for $d \in D$ there exists d' such that $Rd = d'$ we say that R is defined at d and write $\langle Rd \rangle$. Let $F : D \rightarrow D$ be a function from D to D and let $d \in D$; $F^i d$ is defined by

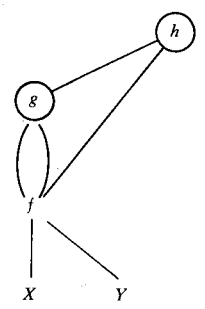
a. $F^0 d = d$;
b. $F^i d = F(F^{i-1} d)$ if $\langle F^{i-1} d \rangle$;
= undefined otherwise.

A straight line program $\pi = (J, I, O)$ over a set of variables Σ and a set of operators $\psi = \{f, g, \cdots\}$ consists of a sequence J of assignment statements, a set of input variables I, and a set of output variables O. Aho and Ullman [19] use a directed acyclic graph (*dag*) to represent straight line programs. We use an equivalent notation in the form of a sequence of assignments. Without giving a formal definition for our notation we illustrate it by an example.

Let $\pi = (J, I, O)$ be a straight line program over $\psi = \{f, g, h\}$ and $\Sigma = \{X, Y, A\}$ where $I = O = \{X, Y\}$ and J is the following sequence of statements:

$A \leftarrow f(X, Y)$
$Y \leftarrow g(A, A)$
$X \leftarrow h(A, Y)$

The *dag* [19] corresponding to π is



The circled nodes represent the most recent definitions of the output variables. Let $\Sigma_1 = \{s_1, s_2, \cdots\}$ be a sequence of new variables such that $\Sigma \cap \Sigma_1 = \phi$. We represent the same program as

$s_1 : f(X, Y)$
$Y \leftarrow s_2 : g(s_1, s_1)$
$X \leftarrow s_3 : h(s_1, s_2)$

Here the most recent definition of the output variables X, Y are indicated by the arrow. Details on the subject of equivalence of straight line programs are contained in [19].

Brief description of the S-machine

The S-machine is a stack oriented machine. The stack stores a variable number of words; at any time, only the word stored at the top level of the stack is accessible. A new word can be stored in the stack by "pushing" the stack, that is, by creating a new level on top of the stack. If the stack is "popped," the word at the top of the stack is discarded and the word on the next level becomes the top of the stack. Part of the S-machine instructions manipulate the stack in various ways, as can be seen from Table 1. For example, the LOAD instruction will push the stack one level and place on the top of the stack a word

Table 1 The S-machine instruction set.

One-address instructions

op-code	*instruction*	*stk change*	*action*
0	LOAD	−1	Load stack from memory location
1	LDI	−1	Load immediate (Loads top of stack with address)
2	STORE	+1	Stores top of stack in memory
3	TRA	0	Transfers control to specified location
4	TPL	0	Transfers control if top of stack is non-negative
5	TMI	0	Transfers control if top of stack is negative
6	TZE	0	Transfers control if top of stack is zero
7	TNZ	0	Transfers control if top of stack is nonzero
8	ENTER	−1	Enter a subroutine by placing CC on top of the stack and transferring control
9	LDX	0	Load X with contents of memory location.
10	LDXI	0	Load X immediate, i.e., the address of this instruction
11	LOOP	0	Increment X by 1 and transfer control if it is nonzero.
12	LS	0	Left-shift N places, where N is the address
13	RS	0	Right-shift N places

Zero-address instructions

op-code	*instruction*	*stk change*	*action*
32	ADD	+1	Add top two levels of stack
33	SUB	+1	Subtract top two levels of stack
34	AND	+1	AND top two levels of stack
35	OR	+1	OR top two levels of stack
36	EOR	+1	EXCLUSIVE OR top two levels of stack
37	NOT	0	Complement top level of stack
38	XTS	−1	Index to stack (Puts contents of X on top of stack)
39	STX	+1	Stack to index (Removes top of stack and places it in X)
40	ADX	+1	Adds top of stack to X
41	SBX	+1	Subtracts top of stack from X
42	RET	+1	Transfers to address contained in top of stack
43	POP	+1	Discards top level of stack
44	STOP	0	Stops the machine by setting SW to zero

taken from a specified memory location. The STORE instruction will place the top of the stack in a main memory location and pop the stack.

The basic data element in the S-machine is the 32-bit word. (This is a variation on the machine described by Gear [2], which is byte oriented.) The main memory contains 2^{24} words with addresses from 0 to $2^{24} - 1$. All S-machine instructions take exactly one word; the instruction format is given in Fig. 1. As can be seen from the figure, both indexing (with respect to the index register X) and indirect addressing are possible in calculation of addresses.

Table 1 lists the S-machine instructions together with the values of their op-code fields, their effects on the stack, and their functions. There are two types of instructions: those which only manipulate the stack without using their address fields (or the indexing and indirect facilities) and are grouped under "zero-address instructions"; and "one-address instructions." The effect of an instruction on the stack is represented by the change it causes on the pointer to the top of the stack. The stack consists of a stack pointer, stored in the reserved register STK and a (variable) storage area in the higher-address region of the main memory. Initially the stack pointer has the value 2^{24}, i.e., it points to the bottom of the stack. The stack is pushed by decrementing the stack register by one, and popped by incrementing it by one.

The S-machine data flow is shown in Fig. 2. The machine operation can be described in the following manner.

The value of the IN1 bus is gated into the left side of the Arithmetic and Logic Unit ALU, while the value of IN2 is gated into the right side of the ALU. Both of these buses are 32 bits wide. One of eight different functions (addition, subtraction, etc.) can be performed in the ALU on the two input operands. The result is gated via the OUT bus into one of the eight registers shown. In addition, an optional memory reference, *read* or *write*, can occur. A *read* causes the 32-bit word addressed by the current content of the Memory Address Register (MAR) to be placed into the Memory Data Register (MDR). A *write* will cause the content of the MDR to

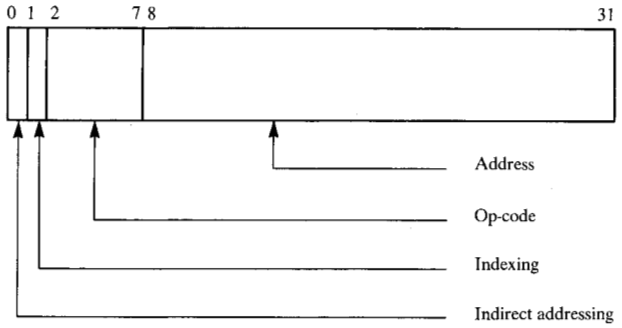


Figure 1 The S-machine instruction format.

be stored in the location addressed by the MAR. This whole sequence of events is known as the machine cycle. As mentioned before, STK contains the stack pointer; *X* is the index register; IR is used to store the op-code during instruction execution. A and B are general working registers; and CC is used to store the pointer to the current instruction in main memory.

The sequencing of data through the data flow is determined by the control which is microprogrammed (Fig. 3). The microprogram resides in the Control Store (CS). The operation of the control can be described as follows.

A word in CS, called a microinstruction, is placed in the Control Store Data Register (CSDR). Decoding of the various fields of the microinstruction takes place, the output of the decoders being connected to gates in the data flow. The boxes labeled "Test logic" and "+1 Adder" determine the address of the next microinstruction; the resulting address is placed in the Control Store Address Register (CSAR).

As illustrated in Fig. 4, the 16-bit microinstruction can have one of two possible formats. 1) When the value of bit zero is zero, indicating that the microinstruction controls the data flow, the fields of the microinstruction deal with various parts of the data flow. The memory field (bits 1, 2), for example, determines whether a *read*, a *write* or no memory reference occurs. Similarly, the function field (bits 9, 10, 11) determines the function to be performed by the ALU. 2) When the value of bit zero is one, indicating that the microinstruction controls the sequencing of the microprogram, the test condition field (bits 1, 2, 3) determines the test condition to be performed for branching. In the microprogram shown in Appendix C the microinstructions have their fields specified by a mnemonic code, instead of the binary code. For example, microinstruction

```
10 ADDM ZERO, B, MAR, R
```

specifies that IN1 bus takes value zero, IN2 bus takes the value of register B, the ALU performs the operation

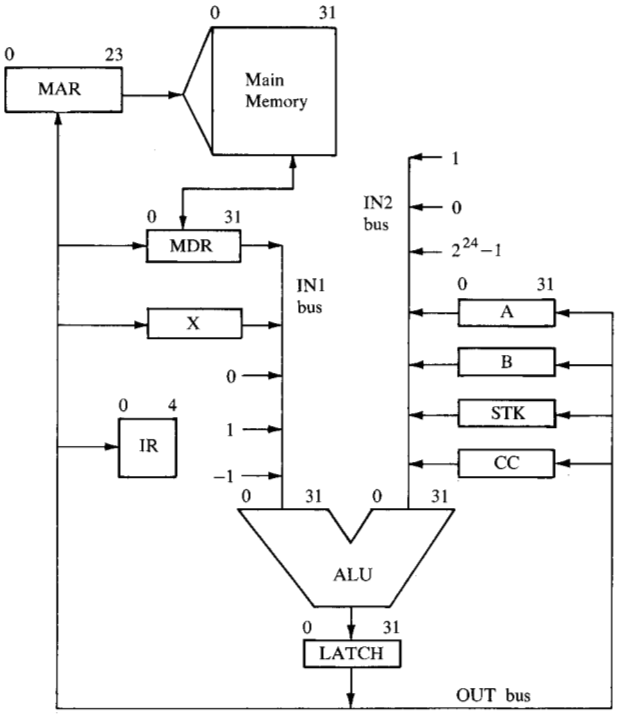


Figure 2 The S-machine data flow.

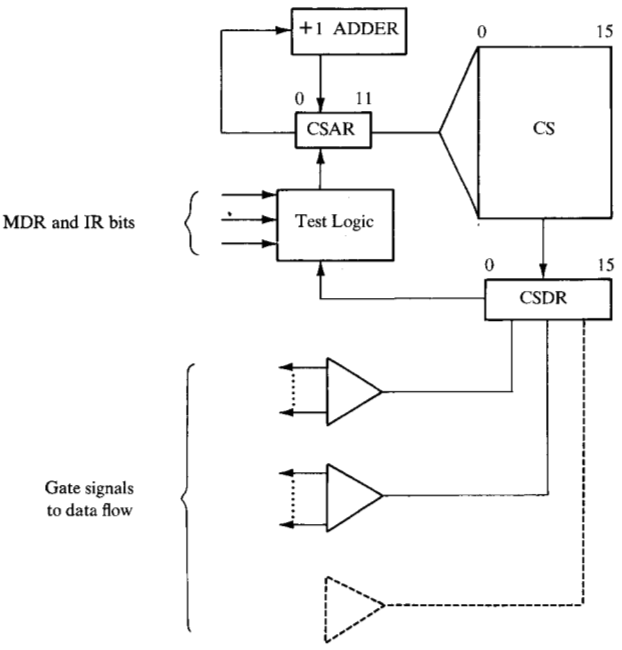


Figure 3 The microprogrammed control.

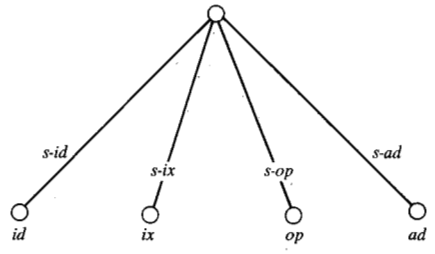
of addition, and the result goes to MAR. In addition, a *read* is started in main memory. The encoding of various fields and other details on the S-machine may be found in [2, 3].

The following is a list of properties of the S-machine which in various ways point to limitation of our results on correctness:

1. The microprogramming in the S-machine is "vertical."
2. There are no I/O instructions.
3. References to memory (both main store and control store) involve no delay; i.e., we have instantaneous read-out.
4. An arbitrary limitation of at most one level of indirect addressing was imposed on the original S-machine.
5. The instruction set is limited; for instance, there is no multiply or divide instruction.

Definition language

The language used to define the abstract machines S and μS is based on the VDL language and makes use of the VDL data structures. We have two classes of *data objects—elementary* and *composite*. The composite objects have a set of *components* that may be selected by unique *selectors*. The following *tree* representation of such an object i shows that i has four components *id*, *ix*, *op*, and *ad*, which can be addressed by using the respective selector. Names associated with the branches of the tree identify the unique selectors. The leaves (terminal nodes) of the tree are formed by the components.



For example, $s\text{-}id(i) = id$. The object i may be constructed by using the construction operator μ_0 as follows:

$$i : \mu_0(\langle s\text{-}id : id\rangle, \langle s\text{-}ix : ix\rangle, \langle s\text{-}op : op\rangle, \langle s\text{-}ad : ad\rangle)$$

The object i appears in the abstract machine S and represents an S-machine instruction. The components of i correspond to the instruction fields: indirect addressing bit, indexing bit, op-code field, address field.

For manipulation of VDL objects the operator μ is available in the language; however, the need for it does not arise in the definitions of S and μS. Often we have to test, given a VDL object, whether it belongs to a certain class of objects. This is done in the VDL through the use of *predicates*. For example a predicate *is-inst* defining the set of all S-machine instruction may be written as

$$\begin{aligned} is\text{-}inst = (&\langle s\text{-}id : is\text{-}bit\rangle,\\ &\langle s\text{-}ix : is\text{-}bit\rangle,\\ &\langle s\text{-}op : is\text{-}opfield\rangle,\\ &\langle s\text{-}ad : is\text{-}adfield\rangle) \end{aligned}$$

where *is-bit* is a basic predicate and the others have to be further defined.

In defining S and μS we made use of the VDL as a definitional system. We supplied the elementary objects for our particular application together with the basic functions and predicates which operate on those objects. The abstract machines we define are similar in structure, in that they are characterized by a *state*, also a VDL object, with the following components:

1. Control;
2. Macrolibrary;
3. Data.

The *control*, represented by a tree, is built from VDL instructions. Given a *control tree*, the leaves of this tree are candidates for processing in the next step; any of those instructions can be picked. The *instructions* are of two types:

Macroinstructions Processing consists of replacing the vertex they occupy in the control tree by a *subtree* of instructions, without modifying any other component.

Elementary instructions Execution usually produces assignments to various components and returns *values* up the control tree [13]. At termination the vertex is deleted from the control tree.

A macroinstruction has the following format:

$$\begin{aligned} &\underline{macro1}\ (param_1, \cdots, \text{param}_k) =\\ &\qquad cond_1 \rightarrow c\text{-}tree_1\\ &\qquad cond_2 \rightarrow c\text{-}tree_2\\ &\qquad \vdots\\ &\qquad cond_m \rightarrow c\text{-}tree_m \end{aligned}$$

One of the control trees $c\text{-}tree_i$ is selected during the macro-expansion to replace the vertex labeled *macro1* according to which predicate $cond_i$ returns the value *true*. The elementary instruction has the format

$$\begin{aligned} &\underline{elem1}\ (param_1, \cdots, param_k) =\\ &\qquad \underline{PASS}: e_0\\ &\qquad s\text{-}sc1: e_1\\ &\qquad \vdots\\ &\qquad s\text{-}scm: e_m, \end{aligned}$$

where the e_i are expressions which may make use of basic functions and predicates. (For convenience, the body of an instruction may appear instead of a control tree $c\text{-}tree_i$ in a macroinstruction; in this case the instruction may use all the parameters of the macroinstruction.) Both macroinstructions and elementary instructions have their definitions in the macrolibrary. The basic functions and predicates are a subset of APL. A list of those actually used is given in Table 2. Not shown in the table are predi-

cates that characterize the elementary objects *is-bit*, *is-binvector*, and *is-binmatrix*. Some changes to the VDL are adopted for convenience:

1. The *list* facility for constructing objects in the VDL and its related function *length* are not being used. Instead, the APL vector and matrix facilities are employed.
2. Predicates are treated as functions with domain {0, 1}. If a predicate appears on the left side of a macroexpansion, then it must have a definition in the library which is evaluated in the normal fashion.
3. If x is a selector operating on state S, we may write x instead of $x(S)$, if S is determined by the context.

• *Abstract machine S*

The definition of S is given in Appendix A; it consists of three parts:

1. Predicate *is-S* (for abstract syntax of S),
2. Initial state;
3. Macrolibrary.

The predicate *is-S* identifies those components of the S-machine which are known at this interface—main store *mem*, stack register *stk*, index register x, instruction counter *cc* and the machine-on-off bit *sw*.

The initial state specifies the value of *stk* and the control tree *s-control* (S); all other components have arbitrary values.

The macrolibrary component of the state contains definitions of VDL instructions; the main macrodefinition is *exec-pgm*:

```
exec-pgm =
  is-run(S) → exec-pgm
                  exec-inst (i)
                      i: fetch-inst
      else → Ω
```

This macro calls itself recursively until the predicate *is-run* takes the value 0 (or *false*) which happens when bit *sw* is turned off, as can be seen for the definition of *is-run*. Each of these calls executes one machine instruction.

The cycle has two parts: *instruction fetch* by macro *fetch-inst* and *instruction execution* by *exec-inst*. As a further example, consider the execution of a LOAD instruction; the part of the macrodefinition *exec-inst* relevant for this case is

```
is-load → load-stk(a)
            a: fetch-word(b)
                b: calc-addr(i)
          adv-ctr
```

Here i is the parameter passed to *exec-inst* and is the object representing the instruction executed. The macro *adv-ctr* increments *cc* by one for the next instruction address. As the tree structure of a macrodefinition is represented by indentation [13], it is seen that after the macro-expansion of *exec-inst* the control tree has two leaves: *adv-ctr* and *calc-addr*(i). The choice of the next step is, however, irrelevant if the definition is consistent; that is, the machine S is determinate. We choose to execute *adv-ctr* first. The execution of LOAD is completed by the sequence of macros: *calc-addr* which computes the address of the word to be loaded on top of the stack; *fetch-word* which fetches that word; and finally *load-stack*, which loads the word on the stack.

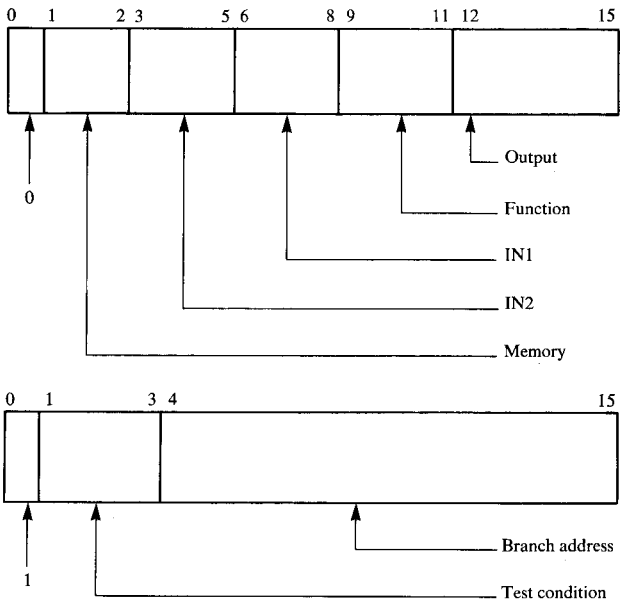


Figure 4 Microinstruction formats.

Table 2 Basic (APL) functions in the definition language.

Function	*Definition or example**
$+, -, \times, *$	Binary arithmetic functions
$<, \leq, =, \neq$	Binary relations on scalars
$\wedge, \vee, \sim$	Boolean functions
ρY	Size of Y; $\rho 2\ 3\ 1\ 5 = 4$
$X\rho Y$	Reshape Y to size X; $3\rho 0 = 0\ 0\ 0$
$X[Y]$	Index X by Y; $2\ 3\ 1\ 5[2\ 3] = 1\ 5$
ιA	First A integers; $\iota 3 = 0\ 1\ 2$
$X \top A$	Representation of A in system with radii X
$A \perp Y$	Value of representation Y in radix A $2\ 2\ 2 \top 5 = 1\ 0\ 1$ $2 \perp 1\ 1\ 0 = 6$
X,Y	Catenation; $1\ 3\ 5, 6\ 3 = 1\ 3\ 5\ 6\ 3$
$A\|B$	A-residue of B; $4\|7 = 3$

* A, B are integers; X, Y are vectors

The abstract machine μS is similarly defined; its definition is given in Appendix B.

Simulation of one program by another

In this section we present a concept of simulation slightly different from Milner's [16] and illustrate a proof of simulation in an example.

Definition 1 An *abstract program* is a triple $P = (D, D_0, F)$ where D is a set called the *domain* of P; $D_0 \subseteq D$ is a set of initial values; and $F: D \to D$ is a partial function. A *computation* is a sequence $(d_0, d_1, d_2, \cdots)$ in which $d_0 \in D_0, d_{i+1} = Fd_i, i = 1, 2, \cdots$.

Consider, for example, the class of flowchart programs. These can be regarded as abstract programs having domain $D \subseteq M \times E$, where M is the set of nodes in the program graph and E is the set of state-vector values. For recursive programs, M could be the infinite set of states of a pushdown stack.

Assume we have a flowchart program and we define $D \subseteq M \times E$. With each flowchart we can associate a function *advance* which takes $s \in M \times E$ as argument and returns a value also in $M \times E$ after executing one instruction. Assume further that we have an edge in the flowchart from node n_i to node n_j and labeled by the assignment $x_k \leftarrow f(x_1, \cdots, x_n)$; let $(x_1, \cdots, x_n) \in E$. Then

$$advance\ (n_i, x_1, \cdots, x_n) = (n_j, x_1, \cdots, x_{k-1}, f(x_1, \cdots, x_n), x_{k+1}, \cdots, x_n).$$

Now define a function *advance'* as follows:

$$\text{advance}'(s) = s \text{ if } s \in D$$
$$= advance'\ (advance(s)) \text{ if } s \notin D.$$

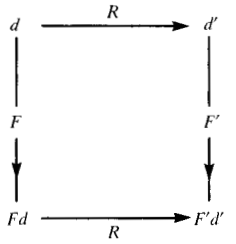
The function *advance'* returns the first value in D encountered by repetitive applications of advance. Finally, $F(d) = advance'\ (advance(d))$ if $d \in D$ and $\langle advance'\ (advance(d))\rangle$ *undefined otherwise.*

By specifying D we have essentially determined F; F is obtained by use of the functions *advance*, *advance'* or some similar device. Often $d \in D$ is given as an expression containing free variables. Calculating Fd amounts to "symbolic execution" of a path. Then we may want to use, instead of the expression Fd, the path that gives rise to Fd.

Definition 2 Let $P = (D, D_0, F)$ and $P' = (D', D_0', F')$ be two programs. A relation $R \subseteq D \times D'$ is a *strong simulation* (or simply, *simulation*) of P by P' if

C1. $\forall (d, d') \in R$ $\langle Fd\rangle$ iff $\langle F'd'\rangle$ and if both are defined then $(Fd, F'd') \in R$.
C2. $\forall d_0 \in D_0$ $\exists\, d_0' \in D_0'$ $(d_0, d_0') \in R$
C3. $\forall d_0' \in D_0'$ $\exists\, d_0 \in D_0$ $(d_0, d_0') \in R$
C4. $\forall d \in D$ $\forall d' \in D'$ $(d, d') \in R$ implies: $[\forall d_1 \in D$ if $(d_1, d') \in R$ then $d_1 = d]$

Condition C1, which is called *weak simulation*, means that the following diagram *commutes*:



This can be also stated as: R is a *weak homomorphism* between the algebraic structures (D, F), (D', F').

Condition C2 asserts that R is total on D_0; Condition C3 asserts that R^{-1} is total on D_0'; Condition C4 means that R^{-1} is single-valued. If R is a strong simulation of P by P' we write $sim(P, R, P')$.

If R is a strong simulation of P by P', P' can compute anything computed by P and this is shown in the following theorem.

Theorem 1 Let $P = (D, D_0, F)$ and $P' = (D', D_0', F')$ be two programs and let $R \subseteq D \times D'$ be such that $sim(P, R, P')$. Then

$\forall d_0 \in D_0 \forall d_0' \in D_0'$ $(d_0, d_0) \in R$ implies
$\langle F^i d_0\rangle$ iff $\langle F'^i d_0'\rangle$, $i = 1, 2, \cdots$,
and if both are defined, then $F^i d_0 = R^{-1}[F'^i d_0']$

Proof The proof is by induction on i. Assume $i = 1$. From C1 of Definition 2 it follows that: $\langle Fd_0\rangle$ iff $\langle F'd_0'\rangle$ and if both are defined $(Fd_0, F'd_0') \in R$. Since R^{-1} is single-valued $Fd_0 = R^{-1}[F'd_0']$. Assume the theorem is true for $i < n$. Then $\langle F^{i-1}d_0\rangle$ iff $\langle F'^{i-1}d_0'\rangle$; if both are defined then $\langle F^i d_0\rangle$ iff $\langle F'^i d_0'\rangle$ according to C1 of Definition 2. The rest of the theorem follows by an induction step.

Theorem 1 states that if $(d_0, d_0') \in R$ and we apply F and F', respectively, to the states of P and P' any number of times, then all pairs of states obtained in this manner are in R. Moreover, the state in P is retrievable from the state of P' due to the single-valued nature of R^{-1}. Therefore, for any computation in P from $d_0 \in D_0$ to $d \in D$, we can use R to obtain the same result as follows. Take d_0' such that $(d_0, d_0') \in R$; the existence of d_0' is guaranteed by C2 of Definition 2. Then use P' to compute $d' \in D'$ such that $(d, d') \in R$. Finally take $d = R^{-1}(d')$.

As mentioned in the introduction, a reasonable concept of simulation should have the following property: For any program P there should exist a simulation of P by P. If there exists a simulation of P by P_1 and also a simulation of P_1 by P_2, then there should exist a simulation of P by P_2. Definition 3 and Theorem 2 show these properties for our simulation concept as given by Definition 2.

Definition 3 Let P, P' be two programs. We define the relation $<$ between two programs by $P < P'$ if there exists R such that $sim\ (P, R, P')$.

Theorem 2 The relation $<$ over the class of programs is reflexive and transitive.

Proof First we show that for any program $P\ \exists\ R$ such that $sim(P, R, P)$. Let $P = (D, D_0, F)$, $R = \{(d, d) | d \in D\}$, i.e., R is the identity relation on D. Using Definition 2 it is easy to check that $sim(P, R, P)$. It remains to be shown that $<$ is transitive. Let $P = (D, D_0, F)$, $P' = (D', D_0', F')$ and $P'' = (D', D_0'', F'')$ be programs and let R, R' be such that $sim(P, R, P')$ and $sim(P', R', P'')$; then $P < P'$, $P' < P''$. Let $Q = RR' = \{(d, d'') | d \in D, d'' \in D'', \exists\, d' \in D'\, (d, d') \in R$ and $(d', d'') \in R'\}$. Using Definition 2 we check that $sim(P, Q, P'')$.

The following example indicates how one proves simulation of one program by another and illustrates in a simplified way some features of the proof for the S-machine.

• *Example*

Consider the programs P, P' (Fig. 5). In P, x is an integer and y is a 32-bit binary vector; a left shift is performed iteratively by executing a one-bit left shift each iteration. The set of integers is denoted by N. We define

$$E = \{(x, y) | x \in N, y \in \{0, 1\}^{32}\}$$
$$M = \{1, 2, 3\}.$$

Then P is represented in our formalism by

$$D = M \times E$$
$$D_0 = \{(1, e) | e \in E\}.$$

For P′ we define

$$E' = \{(x', y', z', A) | x', z' \in N, y' \in \{0, 1\}^{32}, A \in N^4\}$$
$$M' = \{1, 2, 3\}$$
$$D' = \{(m', e') | m' \in \{1, 3\}, e' \in E'\} \cup \{(2, e') | e' \in E', z' = 1\}$$
$$D_0' = \{(1, e') | e' \in E'\}.$$

We postulate the following simulation R:

$$R = R_{11} \cup R_{22} \cup R_{33}$$
$$R_{11} = \{(d, d') | m = m' = 1, x = x', y = y'\}$$
$$R_{22} = \{(d, d') | m = m' = 2, x = x', y = y', A = (0\ 1\ 2\ 3), z' = 1\}$$
$$R_{33} = \{(d, d') | m = m' = 3, x = x', y = y'\}.$$

Then we show that $sim(P, R, P')$.

Proof We show that Conditions C1 through C4 in Definition 2 are satisfied for the R defined above.

Condition C1 is proved by cases. The cases are enumerated 1.1 through 1.4.

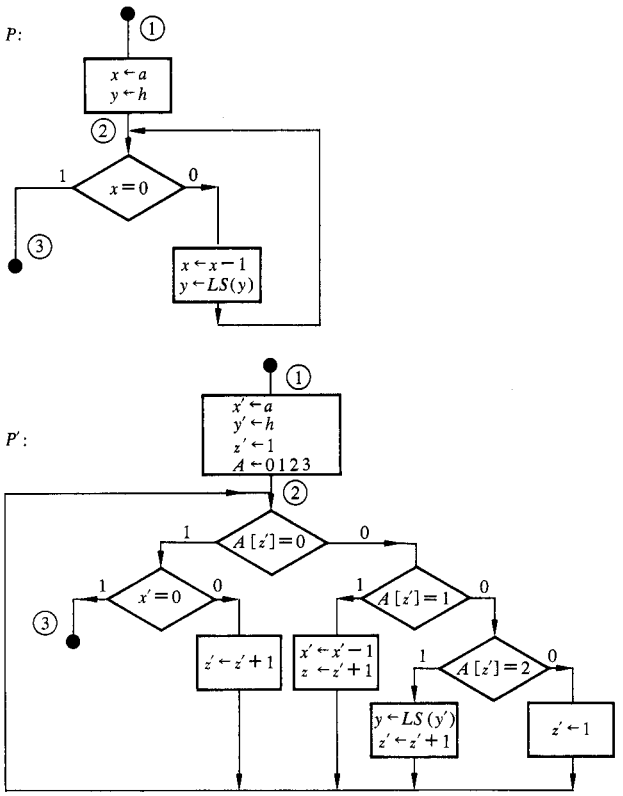


Figure 5 The programs P and P' (True ↔ 1, False ↔ 0).

1.1 Let $(d, d') \in R_{11}$; i.e., $m = m' = 1$. The symbolic execution of Fd, $F'd'$ corresponds to the following straight line programs (we use begin, end as delimiters for straight line programs):

P begin $(m = 1)$	P' begin $(m' = 1)$
$x \leftarrow p1$: a	$x' \leftarrow q1$: a
$y \leftarrow p2$: h	$y' \leftarrow q2$: h
P end $(m = 2)$	$z' \leftarrow q3$: 1
	$A \leftarrow q4$: (0 1 2 3)
	P' end $(m' = 2)$

To verify that $(Fd, F'd') \in R_{22}$, from the above it can be shown that at node 2 the following relations hold:

a. $z = 1$
b. $x = x'$
c. $y = y'$
d. $A = (0\ 1\ 2\ 3)$.

1.2 Let $(d, d') \in R_{22}$ such that $x = 0$.

P begin $(m = 2)$	P' begin $(m = 2)$
P end $(m = 3)$	P' end $(m = 3)$

Verify $(Fd, F'd') \in R_{33}$:

a. $x = x'$
b. $y = y'$.

1.3 Let $(d, d') \in R_{22}$ and $x \neq 0$.

P begin	P' begin
$x \leftarrow p1: x - 1$	$q1: z' + 1$
$y \leftarrow p2: LS(y)$	$x' \leftarrow q2: x' - 1$
	$q3: q1 + 1$
P end	$y' \leftarrow q4: LS(y')$
	$q5: q3 + 1$
	$z' \leftarrow q6: q5 + 1$
	P' end

Verify $(Fd, F'd') \in R_{22}$ by:

a. $x = x'$
b. $y = y'$
c. $A = (0\ 1\ 2\ 3)$
d. $z' = 1$.

1.4 Let $(d, d') \in R_{33}$. Both $Fd, F'd'$ are undefined.

The outlines of proof for C2, C3, and C4 are as follows
C2 Show $\forall d_0 \in D_0 \quad \exists\, d_0' \in D_0' \quad (d_0, d_0') \in R$. We have $d_0 = (1, e)$ for some $e \in E$; let $d_0' = (1, e')$, where $e' \in E'$ such that $x = x'$, $y = y'$. The above statement follows.
C3 Show $\forall d_0' \in D_0' \quad \exists\, d_0 \in D_0 \quad (d_0, d_0') \in R$. Let $d_0' = (i, e')$, for some $e' \in E'$; let $d_0 = (1, e)$ such that $x = x'$, and $y = y'$. By direct substitution one can verify that $(d_0, d_0') \in R$.
C4 Show R^{-1} is single-valued. Let $d' = (2, e')$, $e' \in E'$; suppose $\exists\, d, d_1 \in D$, $d = (2, e)$, and $d_1 = (2, e_1)$ such that $(d, d') \in R$, $(d_1, d') \in R$. Then $x = x'$, $y = y'$, $x_1 = x'$, $y_1 = y'$ and therefore $e_1 = (x_1, y_1) = (x, y) = e$. Similarly for $d' = (1, e')$ or $d' = (3, e')$.

Simulation of S by μS

Before we can define a relation R and prove that it is indeed a simulation of S by μS, we have to bring the abstract machines S and μS to the form of abstract programs. We define an abstract program $S = (D, D_0, F_S)$ where

$$D = \{\alpha | is\text{-}S(\alpha),\ s\text{-}control(\alpha) = \underline{exec\text{-}pgm} \vee s\text{-}control(\alpha) = \Omega\}$$

$$D_0 = \{\alpha | \alpha \in D,\ stk(\alpha) = (32\rho 2) \top 2{*}24,\ s\text{-}control(\alpha) = \underline{exec\text{-}pgm}\}.$$

Let us assume that we have a function *advance* which accepts one argument, a state of any VDL machine, and performs one elementary step, either a basic operation or a macro-expansion. Moreover, $advance(\alpha)$ is undefined if $s\text{-}control(\alpha) = \Omega$. Then let *ad-S* and F_S be defined by

$$ad\text{-}S(\alpha) = \alpha \text{ if } \alpha \in D,$$
$$= ad\text{-}S(advance(\alpha)) \text{ if } \alpha \notin D;$$
$$F_S(\alpha) = ad\text{-}S(advance(\alpha)) \text{ if } \alpha \in D \text{ and } \langle ad\text{-}S(advance(\alpha))\rangle,$$
$$= \text{undefined otherwise.}$$

The function *ad-S* takes as argument a state of S and by using *advance* repeatedly, produces the first state of S, which is also in D. The possibility exists that *ad-S* is undefined because the computation does not terminate. The function F_S accepts a state in D and returns the next state in D.

Similarly, $\mu S = (D', D_0', F_{\mu S})$:

$$D' = \{\beta | is\text{-}\mu S(\beta),\ s\text{-}control(\beta) = \underline{exec\text{-}\mu pgm} \vee s\text{-}control(\beta) = \Omega,\ csar(\beta) = 12\rho 0\};$$

$$D_0' = \{\beta | \beta \in D',\ stk(\beta) = (32\rho 2) \top 2{*}24,\ cs(\beta) = MCODE1,\ s\text{-}control(\beta) = \underline{exec\text{-}\mu pgm}\};$$

$$ad\text{-}\mu S(\beta) = \beta \text{ if } \beta \in D',$$
$$= ad\text{-}\mu S(advance(\beta)) \text{ if } \beta \notin D';$$

$$F_{\mu S}(\beta) = ad\text{-}\mu S(advance(\beta)) \text{ if } \beta \in D' \text{ and } \langle ad\text{-}\mu S(advance(\beta))\rangle,$$
$$= \text{undefined otherwise.}$$

Let $R = R_1 \cup R_2$;

$$R_1 = \{(\alpha, \beta) | \alpha \in D, \beta \in D',\ mem(\alpha) = mem(\beta),\ stk(\alpha) = stk(\beta),\ cc(\alpha) = cc(\beta),\ x(\alpha) = x(\beta),\ sw(\alpha) = sw(\beta),\ s\text{-}control(\alpha) = \underline{exec\text{-}pgm},\ s\text{-}control(\beta) = \underline{exec\text{-}\mu pgm}\};$$

$$R_2 = \{(\alpha, \beta) | \alpha \in D, \beta \in D',\ mem(\alpha) = mem(\beta),\ stk(\alpha) = stk(\beta),\ cc(\alpha) = cc(\beta),\ x(\alpha) = x(\beta),\ sw(\alpha) = sw(\beta),\ s\text{-}control(\alpha) = \Omega,\ s\text{-}control(\beta) = \Omega\}.$$

The proof that R is indeed a strong simulation of S by μS follows the same pattern as the example in the previous section.

• *Proof of simulation*

Condition C1 of Definition 2 is proved by cases.

1.1 Let $(\alpha, \beta) \in R_1$ and $sw = 0$. By applying F_S to α and $F_{\mu S}$ to β we get the sequences:

S-begin $[\alpha]$;
S-end $[F_S(\alpha)]$ where $s\text{-}control\ (F_S(\alpha)) = \Omega$;
μS-begin $[\beta]$;
μS-end $[F_{\mu S}(\beta)]$ where $S\text{-}control\ (F_{\mu S}(\beta)) = \Omega$.

Using the values of $F_S(\alpha)$ and $F_{\mu S}(\beta)$ from above we verify that $(F_S(\alpha), F_{\mu S}(\beta)) \in R_2$.

1.2 Let $(\alpha, \beta) \in R_1$ and *P-LOAD*-00, where the predicate *P-LOAD*-00 is defined by

$$P\text{-}LOAD\text{-}00 = (sw = 1) \wedge (0 = 2\perp s\text{-}op(bi)) \wedge (0 = s\text{-}id(bi)) \wedge (0 = s\text{-}ix(bi)).$$

The VDL object *bi* is defined in Table 3 by a straight line program. The following sequences are then obtained for $F_S(\alpha), F_{\mu S}(\beta)$:

S-begin

$$\begin{aligned} cc \leftarrow\ & s1 : (32\rho 2)\top(2{*}32)|1 + 2\perp cc \\ & s2 : 2\perp sa\text{-}00[8 + \iota 24] \\ & s3 : mem[s2;] \\ stk \leftarrow\ & s4 : (32\rho 2)\top(2{*}32)|\bar{}1 + 2\perp stk \\ & s5 : 2\perp s4[8 + \iota 24] \\ mem[s5;] \leftarrow\ & s6 : s3 \end{aligned}$$

S-end

μS-begin

$$\begin{aligned} cc \leftarrow\ & m1 : (32\rho 2)\top(2{*}32)|1 + 2\perp cc \\ & m2 : 8\rho 0, mi[2 + \iota 24] \\ & m3 : ma\text{-}00[8 + \iota 24] \\ & m4 : mem[2\perp m3;] \\ stk \leftarrow\ & m5 : (32\rho 2)\top(2{*}32)|\bar{}1 + 2\perp stk \\ & m6 : m5[8 + \iota 24] \\ mem[2\perp m6;] \leftarrow\ & m7 : m4 \end{aligned}$$

μS-end

We verify $(F_S(\alpha), F_{\mu S}(\beta)) \in R_1$ by

a. $s1 = m1$;
b. $s4 = m5$;
c. $s5 = 2\perp m6$;
d. $s6 = m7$.

Other cases are handled similarly. Most of these symbolic executions induce to straight line paths and pose no special problems. In two cases, however, we obtain a loop in μS; those paths correspond to left-shift and right-shift instructions. The equivalence of the looping path in μS with the straight line path in S is shown through the method of inductive assertions [18].

Conditions C2, C3 and C4 are proved as follows:

C2 We show $\forall \alpha_0 \in D_0 \ \exists \beta_0 \in D_0' \ (\alpha_0, \beta_0) \in R$. Assume α_0 is given; let $\beta_0 \in D'$ be such that $s\text{-}control(\beta_0) = \underline{exec\text{-}\mu pgm}$, $mem(\alpha_0) = mem(\beta_0)$, $stk(\beta_0) = (32\rho 2)\top 2{*}24$, $cc(\alpha_0) = cc(\beta_0)$, $cs(\beta_0) = MCODE1$, $x(\beta_0) = x(\alpha_0)$, $sw(\alpha_0) = sw(\beta_0)$. From the above it follows $(\alpha_0, \beta_0) \in R_1$.

C3 We show $\forall \beta_0 \in D_0' \ \exists \alpha_0 \in D_0 \ (\alpha_0, \beta_0) \in R$. The proof is similar to the one above.

C4 To show that R^{-1} is single-valued, assume that $\alpha_1, \alpha_2 \in D$ such that $(\alpha_1, \beta) \in R$, $(\alpha_2, \beta) \in R$. It follows that $\alpha_1 = \alpha_2$.

Table 3 Some expressions used in the abstract machines.

Machine S

```
   a1: 2⊥cc [8 + ι 24]
    si: mem[a1;]
   a2: si[8 + ι 24]
   a3: si[2 + ι 6]
   a4: si[1]
   a5: si[0]
    bi: μ0(⟨s-id:a5⟩,
           ⟨s-ix:a4⟩,
           ⟨s-op:a3⟩,
           ⟨s-ad:a2⟩)
   ac: (32ρ2)⊤(2*32)| 1 + 2⊥cc
sa-00: 8ρ0, s-ad(bi)
sa-01: (32ρ2)⊤(2*32)|(2⊥x) + 2⊥s-ad(bi)
   a6: mem[2⊥s-ad(bi);]
sa-10: 8ρ0, a6[8 + ι24]
   a7: mem[2⊥sa-01[8 + ι 24];]
sa-11: 8ρ0, a7 [8 + ι 24]
```

Machine μS

```
   b1: cc[8 + ι 24]
   mi: mem[2⊥b1;]
   b2: mi[ι8], 24ρ0
   ii: b2 [4 + ι5]
ma-00: 8ρ0, mi[8 + ι24]
ma-01: (32ρ2) ⊤ (2*32)|(2 ⊥ x) + 2 ⊥ma-00
   b3: ma-00 [8 + ι 24]
ma-10: 8ρ0, mem[2⊥b3; 8 + ι24]
   b4: ma-01 [8 + ι24]
ma-11: 8ρ0, mem [2⊥b4; 8 + ι24]
```

Summary and discussion

At the start of our investigation a complete, independently written microprogram for the S-machine was available. In the course of the simulation proof three errors were discovered in the microprogram (the corrected microprogram is found in Appendix C.):

1. One error involved the left-shift (LS) and right-shift (RS) instructions. The instructions were executed correctly only if the address field (holding the number of bits to be shifted) was not zero. If the address field was zero the microprogram entered a loop which counted down $2^{24} - 1$ one-bit shifts.
2. Another error concerned the instructions stack-to-index (STX), add-stack-to-index (ADX), and subtract-stack-from-index (SBX). For any of these instructions STX was executed.
3. A third error was found in the instruction-fetch part of the microcode. If the address field of the instruction interpreted was less then 2^{23} there was no ill effect; if, however, the address was larger than or equal to 2^{23}, no instruction was executed correctly, not even the STOP instruction. Moreover, unlike the two previous errors, which could have been found using the simulator package [3], this error could not be found

because the main memory in the simulator was, for convenience, only 100 words in size.

In this paper we have introduced an approach to proving correctness of microprograms and we illustrated it by showing the correctness of the S-machine. At first, we defined abstract machines S and μS corresponding to two interfaces—the machine-instruction level and the microinstruction level.

Defining S and μS is an iterative process, involving several passes, for the following reasons:

1. The original definition of S, although otherwise acceptable, might pose unnecessary restrictions on the implementation; we remove those restrictions by changing S.
2. A change in S might make μS easier to understand.
3. We want to change S because we found a "mistake". This is to be expected since defining μS involves, in a sense, redefining S and sometimes discovering that things are not as were intended in the first place.

Defining an interface in a precise language seems to be of value even if we are not concerned with proving correctness. The precise definition removes the ambiguities of the natural language and provides the means for good documentation against which any inconsistencies that might arise in implementation can be checked. It is true that for an uninitiated person it is harder to read a document written in the definition language than a manual written in English, but the effort required to learn the definition language will pay off. Another advantage of a precise definition: An interpreter for the definition language can be written which will give the designer the possibility to "run" the abstract machine and thus gain a better understanding of the design. And finally, when the problem of proving correctness of implementation comes up, the abstract machine is an essential starting point.

One of the things we learned from this work concerns the mechanization of the correctness proof. Because our proof was carried out by hand, it became clear that for practical applications the proof has to be mechanized to a large extent. Most of the difficulty in carrying out the proof by hand was not in the complexity of the proof, but mainly in the large amount of detail one has to keep track of in such a proof. Specifically, one has to cope with the large number of paths to be compared, the symbolic execution of those paths, and proofs of equivalence of expressions involving APL operators. It is encouraging that most of the correctness proof for the S-machine seems amenable to automation. The approach to correctness developed in the course of our experiment could be used in practical applications, especially for hardware/firmware design. The work presented here could be extended in the following directions:

1. Applying our approach to a more realistic computer. I/O facilities, "Horizontal" microprogramming, more realistic timing constraints could be included.
2. Investigating simulation of parallel programs. In our experiment we reduced whatever parallelism appeared in S and μS to a sequential case. In a more general case one has to cope with the problem of determinacy even before attacking the problem of correctness.
3. Designing an experimental interactive system which will mechanize the proof procedure.

Appendix A: Definition of S

• *Abstract syntax of S*

```
is-S = (⟨mem : is-mem⟩,
        ⟨stk : is-reg⟩,
        ⟨x: is-reg⟩,
        ⟨cc: is-reg⟩,
        ⟨sw: is-bit⟩,
        ⟨s-control: is-control⟩,
        ⟨s-lib: is-lib⟩)
```

• *Initial state*

```
stk: (32ρ2)⊤2*24
s-control(S) : exec-pgm
```

• *Macro library*

```
1. exec-pgm =
     is-run(S) → exec-pgm
                   exec-inst (i)
                     i: fetch-inst
           else → Ω
2. fetch-inst =
     PASS: build-inst (a)
             a: fetch-word (cc)
3. fetch-word (t) =
     PASS: mem [m ;]
             m: 2 ⊥ t [8 + ι 24]
4. build-inst (t) =
     PASS: μ0(⟨s-id:id⟩, ⟨s-ix:ix⟩,
     ⟨s-op:op⟩, ⟨s-ad:ad⟩)
       id: t[0]
       ix: t[1]
       op: t[2 + ι6]
       ad: t[8 + ι24]
5. exec-inst (i) =
     is-load (i)   → load-stk (a)
                       a: fetch-word (b)
                         b: calc-addr (i)
                     adv-ctr
     is-ldi(i)     → load-stk (8ρ0, s-ad (i))
                     adv-ctr
```

```
is-store(i)   → store-word (a, b)
                  a: pop-stk
                    b: calc-addr (i)
                adv-ctr
is-branch (i) → cond (i) → cc: calc-addr (i)
                else       → adv-ctr
is-enter (i)  → cc: calc-addr (i)
                    load-stk (cc)
                      adv-ctr
is-ldx (i)    → x : fetch-word (a)
                      a: calc-addr (i)
                adv-ctr
is-ldxi (i)   → x : 8ρ0, s-ad (i)
                adv-ctr
is-loop (i)   → test-loop(i)
                  incr-x
is-ls(i)      → store-word (a, stk)
                    a: ls (b, 2 ⊥ s-ad (i))
                      b: fetch-word (stk)
                adv-ctr
is-rs(i)      → store-word (a, stk)
                    a: rs (b, 2 ⊥ s-ad (i))
                      b: fetch-word (stk)
                adv-ctr
is-bin (i)    → store-word (a, stk)
                    a: bin (b, c, i)
                      c: fetch-word (stk)
                        b: pop-stk
                adv-ctr
is-not (i)    → store-word (a, stk)
                    a: ~ fetch-word (a, stk)
                adv-ctr
is-xts (i)    → load-stk (x)
                adv-ctr
is-stx (i)    → x : pop-stk
                adv-ctr
is-adx (i)    → x: add (x, a)
                    a: pop-stk
                adv-ctr
is-sbx (i)    → x : sub (x, a)
                    a : pop-stk
                adv-ctr
is-ret (i)    → cc : pop-stk
is-pop (i)    → pop-stk
                adv-ctr
is-stop (i)   → sw(s) : 0
                adv-ctr
```

6. *is-branch* (*i*) =
   PASS : *is-tra* (*i*) ∨ *is-tpl* (*i*) ∨ *is-tmi* (*i*) ∨
   is-tze (*i*) ∨ *is-tnz* (*i*)

7. *cond*(*i*) =
   PASS: *is-tra* (*i*) ∨
   (*is-tpl* (*i*) ∧ (*a*[0] = 0)) ∨
   (*is-tmi* (*i*) ∧ (*a*[0] = 1)) ∨
   (*is-tze* (*i*) ∧ (0 = 2⊥*a*)) ∨
   (*is-tnz* (*i*) ∧ (0 ≠ 2⊥*a*))
   a : *mem* [2 ⊥ *stk* [8 + ι 24];]

8. *store-word* (*a*, *t*) =
   mem [2 ⊥ *t*[8 + ι 24[;] : *a*

9. *load-stk* (*a*) = *store-word* (*a*, *stk*)
   push-stk

10. *push-stk* =
    stk : (32ρ2) ⊤ (2*32)| ¯1 + 2 ⊥ *stk*

11. *pop-stk* = *stk* : (32ρ2) ⊤ (2*32) | 1 + 2 ⊥ *stk*
    PASS : *fetch-word* (*stk*)

12. *adv-ctr* = *cc* : (32ρ2) ⊤ (2*32) | 1 + 2 ⊥ *cc*

13. *calc-addr* (*i*) = *calc-id* (*i*, *a*)
    a: *calc-ix(i)*

14. *calc-ix(i)* =
    (1 = *s-ix* (*i*)) → PASS : (32ρ2) ⊤ (2*32) | *b* + *c*
    b: 2 ⊥ *x*
    c: 2 ⊥ *s-ad*(*i*)
    else → PASS: 8ρ0, *s-ad*(*i*)

15. *calc-id* (*a*, *i*) =
    (1 = *s-id*(*i*)) → PASS: 8ρ0, *b*[8 + ι 24]
    b : *fetch-word* (*a*)
    else → PASS: *a*

16. *incr-x* =
    x : (32ρ2) ⊤ (2*32) | 1 + 2 ⊥ *x*

17. *test-loop* (*i*) =
    (0 ≠ 2 ⊥ *x*) → *cc* : *calc-addr* (*i*)
    else → *adv-ctr*

18. *ls* (*a*, *N*) =
    PASS: (*a*, *N*ρ0)[*N* + ι 32]

19. *rs* (*a*, *N*) =
    PASS: ((*N*ρ*a*[0]), *a*) [ι 32]

20. *is-bin* (*i*) =
    PASS: *is-add* (*i*) ∨ *is-sub* (*i*) ∨ *is-and* (*i*) ∨
    is-or (*i*) ∨ *is-eor* (*i*)

21. *bin* (*a*, *b*, *i*) =
    is-add (*i*) → PASS: *add* (*a*, *b*)
    is-sub (*i*) → PASS: *sub* (*a*, *b*)
    is-and (*i*) → PASS: *a* ∧ *b*
    is-or (*i*) → PASS: *a* ∨ *b*
    is-eor (*i*) → PASS: (*a* ∧ ~ *b*) ∨ (~ *a* ∧ *b*)

22. *add* (*a*, *b*) =
    PASS: (32ρ2) ⊤ (2*32)|(2 ⊥ *a*) + 2 ⊥ *b*

23. *sub* (*a*, *b*) =
    PASS: (32ρ2) ⊤ (2*32) | (2 ⊥ *a*) − 2 ⊥ *b*

24. *is-run* (*S*) =
    PASS: *sw*(*S*) = 1

```
25. is-load (i) =
      PASS: 0 = 2 ⊥ s-op (i)
    is-ldi (i) =
      PASS: 1 = 2 ⊥ s-op (i)
    is-store (i) =
      PASS: 2 = 2 ⊥ s-op (i)
    is-tra (i) =
      PASS: 3 = 2 ⊥ s-op (i)
    is-tpl (i) =
      PASS: 4 = 2 ⊥ s-op (i)
    is-tmi (i) =
      PASS: 5 = 2 ⊥ s-op (i)
    is-tze (i) =
      PASS: 6 = 2 ⊥ s-op (i)
    is-tnz (i) =
      PASS: 7 = 2 ⊥ s-op (i)
    is-enter (i) =
      PASS: 8 = 2 ⊥ s-op (i)
    is-ldx (i) =
      PASS: 9 = 2 ⊥ s-op (i)
    is-ldxi (i) =
      PASS: 10 = 2 ⊥ s-op (i)
    is-loop (i) =
      PASS: 11 = 2 ⊥ s-op (i)
    is-ls (i) =
      PASS: 12 = 2 ⊥ s-op (i)
    is-rs (i) =
      PASS: 13 = 2 ⊥ s-op (i)
    is-add (i) =
      PASS: 32 = 2 ⊥ s-op (i)
    is-sub (i) =
      PASS: 33 = 2 ⊥ s-op (i)
    is-and (i) =
      PASS: 34 = 2 ⊥ s-op (i)
    is-or (i) =
      PASS: 35 = 2 ⊥ s-op (i)
    is-eor (i) =
      PASS: 36 = 2 ⊥ s-op (i)
    is-not (i) =
      PASS: 37 = 2 ⊥ s-op (i)
    is-xts (i) =
      PASS: 38 = 2 ⊥ s-op (i)
    is-stx (i) =
      PASS 39 = 2 ⊥ s-op (i)
    is-adx (i) =
      PASS: 40 = 2 ⊥ s-op (i)
    is-sbx (i) =
      PASS: 41 = 2 ⊥ s-op (i)
    is-ret (i) =
      PASS: 42 = 2 ⊥ s-op (i)
    is-pop (i) =
      PASS: 43 = 2 ⊥ s-op (i)
    is-stop (i) =
      PASS = 44 = 2 ⊥ s-op (i)
```

```
26. is-mem (t) =
      is-binmatrix (t) → ρt = (2*24 32) → PASS : 1
                                 else   → PASS : 0
      else             → PASS : 0
    is-reg (t) =
      is-binvector (t) → (ρt) = 32  → PASS: 1
                                 else → PASS: 0
      else             → PASS: 0
```

Appendix B: Definition of μS

• *Abstract syntax of μS*

```
is-μS = (⟨mem : is-mem⟩,
         ⟨stk    : is-reg⟩,
         ⟨cc     : is-reg⟩,
         ⟨sw     : is-bit⟩,
         ⟨cs     : is-controlstore⟩,
         ⟨mdr    : is-reg⟩,
         ⟨x      : is-reg⟩,
         ⟨a      : is-reg⟩,
         ⟨b      : is-reg⟩,
         ⟨mar    : is-adreg⟩,
         ⟨csar   : is-csreg⟩,
         ⟨ir     : is-ireg⟩,
         ⟨s-control: is-control⟩,
         ⟨s-lib: is-lib⟩)
```

• *Initial state*

```
stk: (32ρ2) ⊤ 2*24
csar: 12ρ0
cs : MCODE1 (Appendix C)
s-control (μS): exec-μpgm
```

• *Macro library for μS*

```
1. exec-μpgm =
     is-run (μS)  → exec-μpgm
                      exec-irem
                        exec-μcycle
     else         → Ω
2. exec-irem =
     is-iexec(μS) → exec-irem
                      exec-μcycle
     else         → Ω
3. is-iexec (α) =
     PASS: (12ρ0) ≠ csar (α)
4. is-run (α) =
     PASS: 1 = sw(α)
5. exec-μcycle = exec-μi (i)
                 i : fetch-μi
6. fetch-μi = build-μi (a)
              a : cs(μS)[2 ⊥ csar(μS);]
```

```
7. build-μi (t) =
     (1 = t[0]) → PASS: μ0(⟨s-branch : a⟩,
                          ⟨s-cond   : b⟩,
                          ⟨s-addr   : c⟩)
                            a : t [0]
                            b : t [1, 2, 3]
                            c : t [4 + ι 12]
     else       → PASS: μ0(⟨s-branch : a⟩,
                          ⟨s-memf   : b⟩,
                          ⟨s-in1    : c⟩,
                          ⟨s-in2    : d⟩,
                          ⟨s-f      : e⟩,
                          ⟨s-out    : f⟩)
                            a : t[0]
                            b : t[1, 2]
                            c : t[3, 4, 5]
                            d : t[6, 7, 8]
                            e : t[9 + ι 4]
                            f : t[13, 14, 15]
8. exec-μi (i) =
     (1 = s-branch (i)) → exec-branch (i)
     else               → exec-assign (i)
9. exec-branch (i)
     is-trm (i)   → csar (μS): s-addr (i)
     is-tϕ (i)    → (mdr(μS)[0] = 0) → csar(μS) : s-addr (i)
                    else             → adv-csar
     is-t1 (i)    → (mdr(μS)[1] = 0) → csar(μS) : s-addr (i)
                    else             → adv-csar
     is-t2 (i)    → (mdr(μS)[2] = 0) → csar(μS): s-addr (i)
                    else             → adv-csar
     is-tmdr (i)  → (mdr(μS) = 32ρ0) → csar(μS): s-addr (i)
                    else              → adv-csar
     is-ti(i)     → csar(μS) : s-addr(i) ∨ (7ρ0, ir(μS))
10. exec-assign (i) =
     is-stop (i) → sw(μS) : 0
                   csar(μS) : 12ρ0
     else        → exec-assign-1(i)
11. exec-assign-1(i) = mem (i)
                         set (i, a)
                         a: result (i, b, c)
                         b: select-1 (i)
                         c: select-2 (i)
                       adv-csar
12. select-1 (i)  =
     is-mdr-1 (i)  → PASS: mdr (μS)
     is-x-1 (i)    → PASS:   x (μS)
     is-mone-1 (i) → PASS: 32ρ1
     is-one-1 (i)  → PASS: 31ρ0, 1
     is-zero-1 (i) → PASS: 32ρ0
13. select-2 (i)  =
     is-one-2 (i)  → PASS: 31ρ0, 1
     is-zero-2 (i) → PASS: 32ρ0
     is-mask-2 (i) → PASS: 8ρ0, 24ρ1
     is-a-2 (i)    → PASS: a(μS)
     is-b-2 (i)    → PASS: b(μS)
     is-stk-2 (i)  → PASS: stk(μS)
     is-cc-2 (i)   → PASS: cc(μS)
14. result (i, a, b) =
     is-add (i)  → PASS: (32ρ2) ⊤ (2*32) | (2 ⊥ a)
                                + 2 ⊥ b
     is-sub (i)  → PASS: (32ρ2) ⊤ (2*32) | (2 ⊥ a)
                                − 2 ⊥ b
     is-and (i)  → PASS: a ∧ b
     is-or (i)   → PASS: a ∨ b
     is-not (i)  → PASS: ∼ (32ρ2) ⊤ (2*32) | (2 ⊥ a)
                           + 2 ⊥ b
     is-eor (i)  → PASS: (a ∧ ∼ b) ∨ (b ∧ ∼ a)
     is-rs (i)   → PASS: c[0], c[ι31]
                     c : (32ρ2) ⊤ (2*32) | (2 ⊥ a)
                       + 2 ⊥ b
     is-ls (i)   → PASS: c[1 + ι31], 0
                     c : (32ρ2) ⊤ (2*32) | (2 ⊥ a)
                       + 2 ⊥ b
15. set (i, a)  =
     is-out-mar (i) → mar(μS): a[8 + ι24]
     is-out-mdr (i) → mdr(μS): a
     is-out-x (i)   → x(μS): a
     is-out-a (i)   → a(μS): a
     is-out-b (i)   → b(μS): a
     is-out-stk (i) → stk(μS): a
     is-out-cc (i)  → cc(μS): a
     is-out-ir (i)  → ir(μS): a[3 + ι5]
16. mem (i)  =
     is-read (i)  → mdr(μS): mem(μS) [2 ⊥ mar(μS);]
     is-write (i) → mem(μS)[2 ⊥ mar(μS);] : mdr (μS)
     is-p (i)     → Ω
17. adv-csar  =
     csar (μS): (12ρ2) ⊤ (2*12) | 1 + 2 ⊥ csar (μS)
18. is-trm (i)  = PASS: 0 = 2 ⊥ s-cond (i)
     is-tϕ (i)  = PASS: 1 = 2 ⊥ s-cond (i)
     is-t1 (i)  = PASS: 2 = 2 ⊥ s-cond (i)
     is-t2 (i)  = PASS: 3 = 2 ⊥ s-cond (i)
     is-tmdr (i) = PASS: 4 = 2 ⊥ s-cond (i)
     is-ti (i)  = PASS: 7 = 2 ⊥ s-cond (i)
19. is-mdr (i)  = PASS: 1 = 2 ⊥ s-in1 (i)
     is-x-1 (i)  = PASS: 2 = 2 ⊥ s-in1 (i)
     is-mone-1 (i) = PASS: 3 = 2 ⊥ s-in1 (i)
     is-one-1 (i) = PASS: 7 = 2 ⊥ s-in1 (i)
     is-zero-1 (i) = PASS: 0 = 2 ⊥ s-in1 (i)
20. is-one-2 (i)  = PASS: 1 = 2 ⊥ s-in2 (i)
     is-zero-2 (i) = PASS: 0 = 2 ⊥ s-in2 (i)
     is-mask-2 (i) = PASS: 2 = 2 ⊥ s-in2 (i)
     is-a-2 (i)  = PASS: 3 = 2 ⊥ s-in2 (i)
     is-b-2 (i)  = PASS: 5 = 2 ⊥ s-in2 (i)
     is-stk-2 (i) = PASS: 6 = 2 ⊥ s-in2 (i)
     is-cc-2 (i) = PASS: 4 = 2 ⊥ in2 (i)
21. is-add (i)  = PASS: 0 = 2 ⊥ s-f (i)
     is-sub (i)  = PASS: 1 = 2 ⊥ s-f (i)
```

```
      is-and (i)      = PASS: 2 = 2 ⊥ s-f (i)
      is-or (i)       = PASS: 3 = 2 ⊥ s-f (i)
      is-not (i)      = PASS: 4 = 2 ⊥ s-f (i)
      is-eor (i)      = PASS: 5 = 2 ⊥ s-f (i)
      is-rs (i)       = PASS: 6 = 2 ⊥ s-f (i)
      is-ls (i)       = PASS: 7 = 2 ⊥ s-f (i)
      is-stop (i)     = PASS: 10 = 2 ⊥ s-f (i)
22.   is-out-mar (i)  = PASS: 0 = 2 ⊥ s-out (i)
      is-out-mdr (i)  = PASS: 1 = 2 ⊥ s-out (i)
      is-out-x (i)    = PASS: 2 = 2 ⊥ s-out (i)
      is-out-a (i)    = PASS: 3 = 2 ⊥ s-out (i)
      is-out-b (i)    = PASS: 5 = 2 ⊥ s-out (i)
      is-out-stk (i)  = PASS: 6 = 2 ⊥ s-out (i)
      is-out-cc (i)   = PASS: 4 = 2 ⊥ s-out (i)
      is-out-ir (i)   = PASS: 7 = 2 ⊥ s-out (i)
23.   is-read (i)     = PASS: 1 = 2 ⊥ memf (i)
      is-write (i)    = PASS: 2 = 2 ⊥ memf (i)
      is-p (i)        = PASS: 0 = 2 ⊥ memf (i)
24.   is-mem (t) =
        is-binmatrix (t) → ρt = (2*24 32) → PASS: 1
                           else           → PASS: 0
        else             → PASS: 0
25.   is-reg (t) =
        is-binvector(t) → (ρt) = 32 → PASS: 1
                          else      → PASS: 0
        else            → PASS: 0
26.   is-controlstore(t) =
        is-binmatrix(t) → ρt = (2*12 16) → PASS: 1
                          else           → PASS: 0
        else            → PASS: 0
27.   is-adreg(t) =
        is-binvector (t) → (ρt) = 24 → PASS: 1
                           else      → PASS: 0
        else             → PASS: 0
28.   is-csreg(t) =
        is-binvector(t) → (ρt) = 12 → PASS: 1
                          else      → PASS: 0
        else            → PASS: 0
29.   is-ireg(t) =
        is-binvector (t) → (ρt) = 5 → PASS: 1
                           else     → PASS: 0
        else             → PASS: 0
```

Appendix C: Microcode

```
          ∇MCODE1[□]∇
        ∇ MCODF1
[1]       1 ADDM ZERO,CC,MAR,R
[2]       2 ADDM ONE,CC,CC,P
[3]       3 TRM 168
[4]       4 T2 7
[5]       5 TI 32
[6]       7 T1 9
[7]       8 ADDM X,B,B,P
[8]       9 T0 12
[9]      10 ADDM ZERO,B,MAR,R
[10]     11 ANDM MDR,MASK,B,P
[11]     12 TI 64
[12]     20 ADDM MONE,STK,STK,P
[13]     21 ADDM ZERO,STK,MAR,W
[14]     22 TRM 1
[15]     23 ADDM MONE,STK,STK,P
[16]     24 ADDM ZERO,STK,MAR,W
[17]     25 TRM 1
[18]     26 ADDM ZERO,B,MAR,W
[19]     27 TRM 54
[20]     29 T0 70
[21]     30 TRM 1
[22]     32 ADDM ONE,STK,MAR,R
[23]     33 TRM 115
[24]     34 ADDM ONE,STK,MAR,R
[25]     35 TRM 115
[26]     36 ADDM ONE,STK,MAR,R
[27]     37 TRM 115
[28]     38 ADDM ONE,STK,MAR,R
[29]     39 TRM 115
[30]     40 ADDM ONE,STK,MAR,R
[31]     41 TRM 115
[32]     42 ADDM ZERO,STK,MAR,R
[33]     43 TRM 142
[34]     44 ADDM MONE,STK,STK,P
[35]     45 TRM 144
[36]     46 ADDM ZERO,STK,MAR,R
[37]     47 TRM 158
[38]     48 ADDM ZERO,STK,MAR,R
[39]     49 TRM 147
[40]     50 ADDM ZERO,STK,MAR,R
[41]     51 TRM 149
[42]     52 ADDM ZERO,STK,MAR,R
[43]     53 TRM 156
[44]     54 ADDM ONE,STK,STK,P
[45]     55 TRM 1
[46]     56 STOPM ZERO,ZERO,MAR,P
[47]     64 ADDM ZERO,B,MAR,R
[48]     65 TRM 20
[49]     66 ADDM ZERO,B,MDR,P
[50]     67 TRM 23
[51]     68 ADDM ZERO,STK,MAR,R
[52]     69 TRM 26
[53]     70 ADDM ZERO, B, CC, P
[54]     71 TRM 1
[55]     72 ADDM ZERO,STK,MAR,R
[56]     73 TRM 29
[57]     74 ADDM ZERO,STK,MAR,R
[58]     75 TRM 97
[59]     76 ADDM ZERO,STK,MAR,R
[60]     77 TRM 99
```

```
[61]    78 ADDM ZERO,STK,MAR,R
[62]    79 TRM 101
[63]    80 ADDM ZERO,CC,MDR,P
[64]    81 TRM 103
[65]    82 ADDM ZERO,B,MAR,R
[66]    83 TRM 106
[67]    84 ADDM ZERO,B,X,P
[68]    85 TRM 1
[69]    86 ADDM X,ONE,X,P
[70]    87 TRM 108
[71]    88 ADDM ZERO,STK,MAR,R
[72]    89 TRM 110
[73]    90 ADDM ZERO,STK,MAR,R
[74]    91 TRM 110
[75]    96 TRM 1
[76]    97 TO 1
[77]    98 TRM 70
[78]    99 TMDR 70
[79]   100 TRM 1
[80]   101 TMDR 1
[81]   102 TRM 70
[82]   103 ADDM MONE,STK,STK,P
[83]   104 ADDM ZERO,STK,MAR,W
[84]   105 TRM 70
[85]   106 ADDM MDR,ZERO,X,R
[86]   107 TRM 1
[87]   108 ADDM X,ZERO,MDR,P
[88]   109 TRM 101
[89]   110 ADDM MDR,ZERO,A,P
[90]   111 ADDM ZERO,B,MDR,P
[91]   112 TRM 124
[92]   115 ADDM MDR,ZERO,A,P
[93]   116 ADDM ZERO,STK,MAR,R
[94]   117 ADDM ONE,STK,MAR,P
[95]   118 TI 128
[96]   120 LSM ZERO,A,A,P
[97]   121 TRM 123
[98]   122 RSM ZERO,A,A,P
[99]   123 SUBM MDR,ONE,MDR,P
[100]  124 TMDR 126
[101]  125 TI 120
[102]  126 ADDM ZERO,A,MDR,W
[103]  127 TRM 1
[104]  128 ADDM MDR,A,MDR,W
[105]  129 TRM 54
[106]  130 SUBM MDR,A,MDR,W
[107]  131 TRM 54
[108]  132 ANDM MDR,A,MDR,W
[109]  133 TRM 54
[110]  134 ORM MDR,A,MDR,W
[111]  135 TRM 54
[112]  136 EORM MDR,A,MDR,W
[113]  137 TRM 54
[114]  142 NOTM MDR,ZERO,MDR,W
[115]  143 TRM 1
[116]  144 ADDM ZERO,STK,MAR,P
[117]  145 ADDM X,ZERO,MDR,W
[118]  146 TRM 1
[119]  147 ADDM MDR,ZERO,A,P
[120]  148 TRM 160
[121]  149 ADDM MDR,ZERO,A,P
[122]  150 TRM 162
[123]  156 ADDM MDR,ZERO,CC,P
[124]  157 TPM 54
[125]  158 ADDM MDR,ZERO,X,P
[126]  159 TRM 54
[127]  160 ADDM X,A,X,P
[128]  161 TRM 54
[129]  162 SUBM X,A,X,P
[130]  163 TRM 54
[131]  168 ANDM MDR,MASK,B,P
[132]  169 NOTM ZERO,MASK,A,P
[133]  170 ANDM MDR,A,MDR,P
[134]  171 LSM MDR,ZERO,IR,P
[135]  172 TRM 4
    ∇
```


Acknowledgments

I acknowledge useful discussions with W. C. Carter, W. H. Joyner, and G. B. Leeman, Jr. I also thank K. Haralson and R. Polivka for providing the S-machine simulator and the microprogram, and the referees for some very helpful comments.

References

1. A. Birman, "Correctness in Design: The S-Machine Experiment," Research Report 4193, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 1973.
2. C. W. Gear, *Computer Organization and Programming*, McGraw-Hill Book Co., Inc., New York, 1969.
3. K. Haralson and R. Polivka, "Microprogram Training – An APL Application," Proc. 4th. Int. APL Users' Conf., 1972.
4. P. C. Gilmore, "An Abstract Computer with a LISP-Like Machine Language without a Label Operator," *Computer Programming and Formal Systems*, edited by P. Braffort and D. Hirschberg, North-Holland Publishing Co., Amsterdam, 1963, p. 71.
5. C. C. Elgot and A. Robinson, "Random-access Stored-program Machines, An Approach to Programming Languages," *J. ACM* **11**, 365 (1964).
6. P. J. Landin, The Mechanical Evaluation of Expressions," *Computer J.* **6**, 308 (1964).
7. K. Walk et al., "Abstract Syntax and Interpretation of PL/I," TR 25.082, IBM Laboratory, Vienna, 1968.
8. P. Lucas, P. Lauer, H. Stigleitner, "Method and Notation for the Formal Definition of Programming Languages," TR 25.087, IBM Laboratory, Vienna, 1968.
9. P. Lucas and K. Walk, "On the Formal Description of PL/I" in *Annual Review in Automatic Programming* **6**, Pergamon Press, New York, 1970.
10. J. A. N. Lee, *Computer Semantics*, Van Nostrand Reinhold Co., New York, 1972.
11. A. D. Falkoff, K. E. Iverson and E. H. Sussenguth, "A Formal Description of System/360," *IBM Syst. J.* **3**, 198 (1964).

12. C. G. Bell and A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill Book Co., Inc., New York, 1971.
13. F. J. Neuhold, "The Formal Description of Programming Languages," *IBM Syst. J.* **10,** 86 (1971).
14. P. Lauer, "Formal Definition of ALGOL 60," TR 25.088, IBM Laboratory, Vienna, 1968.
15. P. Wegner, "The Vienna Definition Language," *ACM Computing Surveys* **4,** 5 (1972).
16. R. Milner, "An Algebraic Definition of Simulation Between Programs," Report CS 205, Stanford University, Calif., February 1971.
17. K. E. Iverson, *A Programming Language*, John Wiley & Sons, Inc., New York, 1962.
18. R. W. Floyd, "Assigning Meanings to Programs," Proceedings of Symposia in Applied Math., Vol. 19, American Mathematical Society, 1967.
19. A. V. Aho and J. D. Ullman, "Optimization of Straight Line Programs," *SIAM J. Computing* **1,** (1972).

Received November 29, 1973

The author is located at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.