A Microprogramming Simulator for Instructional Use

J. R. Parker
K. Becker
Department of Computer Science
University of Calgary
2500 University Drive N.W.
Calgary, Alberta, Canada
T2N-1N4

The teaching of computer architecture at a low level is made difficult by the complexity of the real systems which are used as examples and tools. This paper describes a processor simulation system which is intended for use at the second and third year undergraduate level for teaching techniques and concepts in the implementation of instruction sets and microprogramming. The important features of this system are in the user interface, and not necessarily in the actual processor which is simulated.

The IEEE Curriculum for undergraduate computer engineering [5], and the ACM curriculum for B.Sc. Computer Science degrees [6,8] both contain courses on computer architecture at the elementary level. Specified as a topic of study is microprogramming and the implementation of computer instruction sets. While the major differences between discrete logic implementation and microprogrammed implementations can be explained in a straightforward way, the details and the 'flavor' of creating a new instruction in microcode is not as easily conveyed. While many modern machines have microprogrammed instruction sets, not all of these possess a writeable control store, and not all are easily used. Some machines can be damaged by bad microcode. Most machines with writeable control store are complex, and most are expensive. Thus, it is not practical to allow the average undergraduate student to directly experience microprogramming on such machines.

What remains as educational tools in this area are chip sets, such as the AM 2900, which are, to first and second year undergraduates, too complex and artificial. An introduction to a new concept should involve suitable metaphors, simplifications, and abstractions to convey the major principles and concepts, rather than a complete and detailed collection of facts which, while real and relevant, are too specific and too many. The result of teaching too many details can often be a student with no grasp of the generality of an idea or its relationship with other concepts.

The apparent artificiallity of devices such as the 2900 lies in the fact that they are marketed solely as a microprogrammable chip, with no resident instruction set. Thus, the microinstructions take the place of normal instructions, as on a CPU chip like the Z80, and microprogramming looks a lot like assembler programming. While microprogramming actually IS a lot like assembler coding in some ways, the intent of using a tool such as this in teaching is to underscore the differences, to allow the student to explore the new techniques involved in microprogramming, and to see the consequences on machine architecture.

Based on the arguments above, what is proposed is a software emulation of a simplified microprogrammable processor. The processor should be simple enough that its structure can be understood after a very few lectures and labs, and yet it should provide features suitable for illustrating advanced concepts in architecture. The students understand that this is an explanitory device, and not a real system. Later, with the skills learned on the simplified system, the more complex 'real' systems may be mastered more quickly. The simulator is a metaphor for the concepts that students consider difficult, and leaves the actual details for later.

Microtool is a system of programs intented to assist in the teaching of concepts in microprogramming and computer architecture. The system provides students with a facility for writing and executing microcode without the normal complexities of dealing with an actual microprogrammable CPU with writable control store. In

© 1984 ACM-0-89791-126-1/84/002/0069 $00.75

addition, high level debugging tools and input and output utilities are provided.

Use of this utility is expected to simplify Junior and Senior level architecture courses, by providing a 'hands-on' demonstration of many important concepts. As well, since the system is written in a high-level language, it is reasonably portable, and cannot harm or even disrupt the processor on which it runs. It also allows large numbers of students to actually experience microprogramming, without incurring a large hardware expense or producing a contention problem. Many students can run the system simultaneously on a multiprocessor (as opposed to a stand alone real system). This is especially important at this time, with class sizes exceeding previous reasonable limits.

The MicroTool (or uTool) system consists of an emulator for a hypothetical microprogrammable CPU, a compiler for a 'high level' micro language, and a collection of exercises which illustrate concepts in computer architecture. Students may write microprograms for the CPU, either as bit sequences or as symbolic statements, and then 'run' them on the emulator. The processor is, of course, a simplified and somewhat idealized one, to avoid clouding the relevant concepts.

The Processor

The system is based on a processor described by Dasgupta [7] and also by Tanenbaum [1], which he calls a "hypothetical host machine", and which will be referred to as the T1a processor in this document. This processor was used, rather than inventing a new one, for a number of reasons. The most relevant rationale for the choice was to ensure that a commonly used textbook was available for use in conjunction with the system. The reader is referred to this text for an excellent detailed description of the processor.

Figure 1 shows the major processor components, giving as well the control points which make up the microstore word. Micro instructions are 40 bits in length, and are of two types: GATE instructions, which move data from a source to a destination, and TEST instructions, which compare a bit constant against a bit in a register, and branch to a target micro address if the bits have the same value. Figure 1 gives the format of these two kinds of instruction.

The micro instructions are executed by the processor in three subcycles, which provides a very important temporal separation between parts of a micro instruction. During the first subcycle, gates 1 thru 29 may be opened, permitting data transfers to take place into the adder. The second subcycle allows gates 30 thru 37 to be opened, which allows results from the adder/shifter to be moved into destination registers. Finally, subcycle three permits either the read or write gates (38 or 39) to be opened, implying that a read or write operation will take place. Parker [4] gives a table of data transfers possible in this processor, and describes in more detail the function of each gate.

The microprogram store, or, control store contains 256 of the 40 bit micro instructions, which means that eight bit microstore addresses are needed. The control store may be written to, which allows user written microcode to be executed.

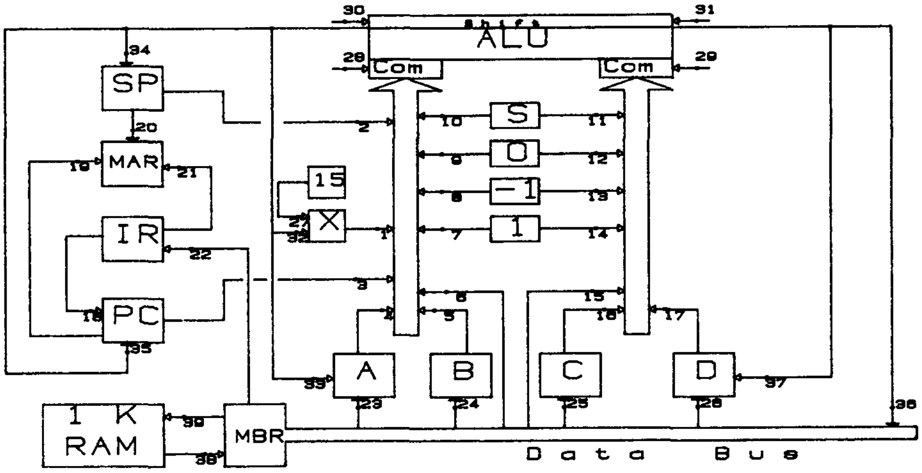
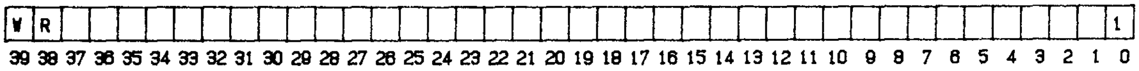


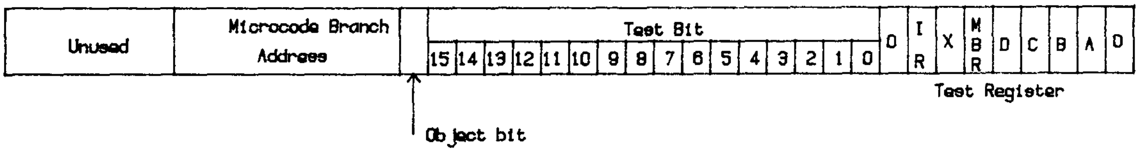
Figure 1 - The Hypothetical CPU

The GATE Instructions :



Each bit of a GATE instruction represents one of the control points (See Fig. 1) A '1' bit allows data to move across a control point, a '0' bit does not. Bit 0 of each microinstruction is the opcode - a GATE instruction has an opcode of '1'. Bits 38 and 39 control reading and writing of main memory, respectively.

The TEST Instructions :



The TEST microinstruction allows any bit of certain registers to be tested. In any named field above, only one bit may be non-zero or an error occurs. The action of the TEST instruction is to examine the indicated test register, to see if the specified bit (The TEST field above, 0-15) is equal to the the value of the Object Bit field (1 or 0). If so, a branch is made to the microinstruction specified by the Microcode Branch Address field. If not, then the next instruction in the normal sequence is executed. Opcode for the TEST instruction is '0'.

Figure 2 - The Microinstruction Formats

The Emulation System

The T1a emulator is a computer program which simulates the T1a hardware. It 'executes' the 40-bit microinstructions one at a time, and permits editing, tracing, and dumping both the 'macro' store and the control store, and allows the examination and modification of all internal machine registers.

The entire system is written in Berkeley PASCAL[1], and runs on a VAX 11/780 under UNIX. The program itself uses only standard language features, and can be easily ported to other systems running PASCAL. In fact, the system is also running on an IBM PC, and the effort needed to make the conversion was minimal.

The emulator reads commands from the user's terminal, but allows the entry of microcode and machine code from user-specified files. All commands are single word names, possibly followed by parameters which can be either numeric or text. As well, some commands apply to microcode only, some to machine code only, and some to both. In cases where a command may apply to both micro and machine code, the micro command name will be the same as the machine code version, except it will have the letter 'u' prepended to it.

For example, the dual purpose command 'dump' is used to dump a set of memory locations. Hence, the command 'udump' is used to dump a set of control store locations.

The dual purpose commands consist mainly of data entry or tracing commands. They include BREAK, CLEAR, DUMP, ENTER, LOAD, STEP, and STORE. Single purpose commands are CHANGE, CONTINUE, DISPLAY, NOTRACE, PROGRAM, QUIT, RESTORE, RUN, and SAVE. However, it makes more sense to group the commands functionally, into input/output commands, debugging/tracing commands, and others.

Tracing and Debugging

These commands very often involve creating, deleting, or using a breakpoint, or an address (either microaddress or normal address) where execution will be suspended. When the program counter contains the address of a breakpoint, the emulator stops running the current program and returns to command level. Breakpoints are set in order to check the current contents of store, values in registers, or to modify some value. The ability to set breakpoints is extremely valuable while debugging microcode (and machine code as well).

The command break (or ubreak) takes one parameter- the address, either micro or macro, at which the breakpoint is to be placed. The emulator then prints out the address, and asks if it is correct, to which you may reply 'yes' or 'no'. Finally, the emulator asks if this is a tracepoint. A tracepoint is a special case of a breakpoint, in that when the tracepoint address is encounted, the emulator will begin to trace the execution of the user programs by dumping instructions to the screen as they are executed. The emulator does not stop executing when the tracepoint is seen, it simply begins the trace.

A micro breakpoint lasts until removed. A macro breakpoint will only exist until it is encountered and causes a break; it will then be removed. Tracepoints will, however, exist until removed.

Breakpoints are removed using the clear command. One parameter, the address of the breakpoint, must be given. The breakpoint (or tracepoint) is then removed.

After a breakpoint has been seen during emulation of user code, the user may enter commands. One useful one is the step command, which causes the emulator to execute the next machine (or micro) instruction, then stop again. Single stepping through code is very useful during debugging.

Once a breakpoint is encountered, it is often desireable to resume processing from the current state. the continue statement is used for this purpose. This command may be entered after a breakpoint is encountered during program execution. The result is that the program resumes executing where it left off. Changes in the machine registers made with a change command will be in effect. The equivalent command for tracepoints is notrace. After a tracepoint is seen, entering the notrace command turns off tracing, until another tracepoint is encountered.

Input/Output Commands

I/O commands cause memory or registers to be either loaded or saved. For example, a user may directly enter into memory from the terminal, either microcode (in its binary form) or machine code (in various forms) by use of the enter command.

For entering microcode, the user is first asked for the address. Then, the bit positions in the microword which are to be SET (ie = 1) are entered. Entry ends with a negative number. Now the emulator stores the microword, and increments and displays the next address value, and asks if you wish to continue. This process repeats until all words are entered. It is assumed that words to be entered are consecutive locations.

For entering machine code, the first address is entered as above, but code entry is input as either hexadecimal, octal, or decimal numbers, depending on the input mode specified. You will be asked for the number base, which must be one of 2, 8, 10, or 16.

The corresponding output command is dump. This command allows main store or microstore to be displayed on the screen. Two parameters, the starting address and the final address, must be specified. All of the memory locations between these two addresses (inclusive) will be written to the terminal.

It will not be often that microcode will be directly entered into memory using enter - it is too slow and clumsy. More often, we will want to read code from data files. The load command requires a file name argument, and this file is expected to contain the required memory words. In the case of microcode, this file may be produced by the MPL compiler. In the case of machine code, it may be entered by a text editor, or by a user program.

Store permits saving the current contents of either microstore or main memory. It requires a file name argument, the name of the file which will contain the memory contents. Later, store may be reloaded from the same file using the load command. Saving both main store and microstore in this way requires two separate files. If the entire processor state, including tracepoints, breakpoints, all registers, and all memorys needs to be saved, then this can be done too. The command save keeps all of this information on a file called 'utool.SAVE' in the current UNIX working directory. The machine state may be recalled using the restore command. Note that there is only one file - many states may be kept only by renaming old versions of the file. program :

The command program is a unique feature of the microtool system. It is used by students to load standard microcode and machine code sequences written by the instructor. The command will request one integer argument, between 0 and 99. The integer argument represents a code, created by the instructor, to identify assignments, lab exercises, etc. This allows students to experiment with standard code sequences before writing their own firmware. The instructor is free to change these code sequences at will, and on some systems (such as UNIX) a usage count for each student may be maintained. Numbers entered by students which do not correspond to a real code file cause a message to the student's terminal.

Quite often during the debugging of microcode, a 'bad' value finds itself in a register. The 'change' command permits the modification of processor registers during program execution. Two arguments are expected: the first is the name of the register to be changed, and the second is the value to be stored in the register. Legal register names (Upper case only!) are :

A	B	C	D
MBR	X	MAR	IR
PC	SP		

The Change command is also used to ini-reasonable values at the outset.

The display command is used to display all of the machine registers, probably after a breakpoint.

Other Commands

The quit command terminates the emulation system and causes a return to UNIX command level.

Run will run the current microprogram on the current contents of main memory. No parameters are expected.

The MPL Compiler

The Microcode Programming Language (MPL) compiler is provided for more convenient programming of the T1a processor. Rather than dealing with individual bits, fields, and opcodes, mnemonics are used to specify register transfers, microcode addresses, and memory access. The syntax is much like that of an assembler in some ways, and like a compiler in others. The compiler described here was inspired by [3], but includes many new features.

There are two basic statement types, because of the two basic instruction types. GATE statements consist of one or more assignments, possibly including simple expressions. TEST statements yield a test microinstruction, and include a test bit and branch address.

A gate statement may be composed of many individual assignment statements, because a gate instruction may perform many simultaneous data transfers. The general syntax of an assignment is

```
DEST = SOURCE;
     or
DEST = Simple_expression;
```

A gate consists of some number of transfer statements, one computational statement, or both kinds mixed, with only one computational statement per gate statement. The gate statement is ended by two semicolons (;;), so that one gate statement may span many lines.

Destinations and sources are specified as key words, representing T1a registers or operations. Registers, etc. may be written in either upper or lower case. Possible destinations are:

A,B,C,D	A thru D are 16 bit registers.
X	The X register.
MAR	Memory Address Register
MBR	Memory Data Register.
IR	Instruction Register.
PC	Program Counter.
SP	Stack Pointer.
memory	Main store.

The T1a processor has a simple arithmetic-logic unit (ALU), which performs simple additions, complements, and shifts. As well, there are constants stored in internal registers wich can be used in computations. Constants available are:

1	- A 16 bit register, = integer 1.
0	- 16 bit integer zero.
-1	- 16 bit two's complement
sign	- 16 bit register, only the sign bit is set.
15	- The integer constant 15.

A simple expression takes the general form :

```
shift_op (source1 + source2)
```

The operator 'shift_op' may be absent or may be one of either 'shift_left' or 'shift_right'. In any case, the addition indicated would be performed, followed by a shift of only one bit in the specified direction.

The source operands above may simply be constants, registers, or may be a complemented constant or register. The one's complement of register A is written

```
complement (A)
```

To shift the A register left one bit, we would enter :

```
A = shift_left (A + 0);;
 / Sets 4, 13, 33 /
```

The procedure for accessing main memory is fairly simple. The address for a memory read operation is always placed in the MAR prior to the read operation being started. The result of the read is a 16 bit memory word, which is placed in the MBR. Remember that reads and writes are performed during subcycle 3 of the micro-sequence, and so the address can be moved to the MAR during the same micro instruction.

A memory read could be written as :

```
MBR = memory ( MAR );;
   or as :
MBR = memory;;
```

since the use of the MAR is understood. MBR must, however, always be the destination of the read.

Any MPL statement may be preceeded by one or more labels, which are symbolic names representing the address in microstore of that statement. For example, :

```
lab1:
tst1:      MBR = memory;;
```

Here, 'lab1' and 'tst1' represent the same location in microstore. Later, these names may be the object of a branch.

Labels need not be defined before they are used, but all labels must be defined somewhere in the program.

When it is necessary to branch to a particular microinstruction out of the normal sequence, then a <u>test</u> statement would be used. There is always a condition involved, which is expressed as the equality of a bit constant (1 or 0) with a specified bit in a particular register. If the two are equal, then the branch will be made; otherwise, the next instruction in the control store will be executed.

The general syntax is

```
if <bit expression>
   then goto <microaddress> ;;
```

The keywords 'then' and 'goto' are always optional, and either, both, or neither may be present.

The general form of a bit expression is

```
bit (register, bit-number)
    =   bit-constant
```

where:

```
bit-number    is a constant between
                 0 and 15.
register      is one of A,B,C,D,
                 MBR, X, IR, or 0.
bit-constant  is 0 or 1. If omitted,
                 1 is assumed.
```

In any place where a comma (,) is seen, the key word 'of' may be used. All equal signs (=) may be replaced by the key word 'is'. All parentheses may be omitted, and extra ones are ignored.

All of the following bit expressions test bit number 14 of the instruction register to see if it is set (=1) :

```
bit (14, IR) = 1
bit (14, IR)
bit 14 of IR is 1
bit 14, IR is 1
bit 14 of IR
bit ((14)of(IR)) is ((1))
```

There are obviously many forms possible for the same test statement. It is assumed that individual programmers will adopt a convention.

Often a simple unconditional branch instruction would be useful. There is no such instruction in our processor, but the MPL compiler can construct one. The GOTO statement is a variation of the test statement, where the intent is to branch unconditionally to a destination address. The condition used is to test any bit, say bit number 0, of the zero register, which has all bits set to 0, against the constant bit 0. This always results in a true condition, and therefore will always result in a branch. The test statement would be :

```
if bit (0, 0) = 0
  then goto <dest>;;
```

where <dest> is some label. In MPL, this can be shortened to

```
goto <dest>;;
```

An Example MPL Program:
A stack machine.

Here, a simple stack machine is written in T1a microcode. There are only six instructions, which are:

Opcode	Instruction	Description
000	PUSH	the low 13 bits of the instruction
001	POP	the top stack value into the A register.
100	ADD	the top two stack values, result on top.
101	SUBtract	the second-from top value from the top value, result on top.
110	STOre	the (Top-1) value into the memory location given by Top.
111	GET	the contents of the address given by the top of the stack, result goes on Top.

The opcodes occupy the top 3 bits of the word. The code for this instruction set could be written in many ways, and the sample solution is not guarenteed to be optimal in any sense.

The solution is given in Figure 3 as an MPL compilation listing. Points to note are, first, that the code written above for the FEC is used without modification, and second, that the execute portion of the fetch-execute cycle must first decode the instruction opcode to determine which instruction to execute. this decoding is performed bit-by-bit, a limitation enforced by the architecture of the micro machine.

A coding sheet has been devised for use as a guide for beginning students coding in binary. It is also useful as a reminder of the fields, transfers, and formats found in the T1a processor. More advanced students will probably write their micro programs in MPL, or some other higher level representation, but it is probably instructional for students to write their first few microprograms in machine format.

Figure 3 shows an audited session with microtool. The session shows how to load both control store and main memory, illustrates the dump formats for both memories, and shows how to dump the registers. Note that the example is still the stack machine. A breakpoint is set at the last executable address in main store, and the program is run- the result is that the program stops after executing the last instruction. Since there is no proper 'halt' instruction, this is the suggested method of halting a program.

References

[1] Joy, W.N., Graham, S.L., Haley, C.B., "Berkeley PASCAL User's Manual Version 2.0", Oct. 1980.

[2] Andrews, M., "Principles of Firmware Engineering in Microprogram Control", Computer Science Press. 1980.

[3] Tannenbaum, A., "Structured Computer Organization", Prentice-Hall, 1976.

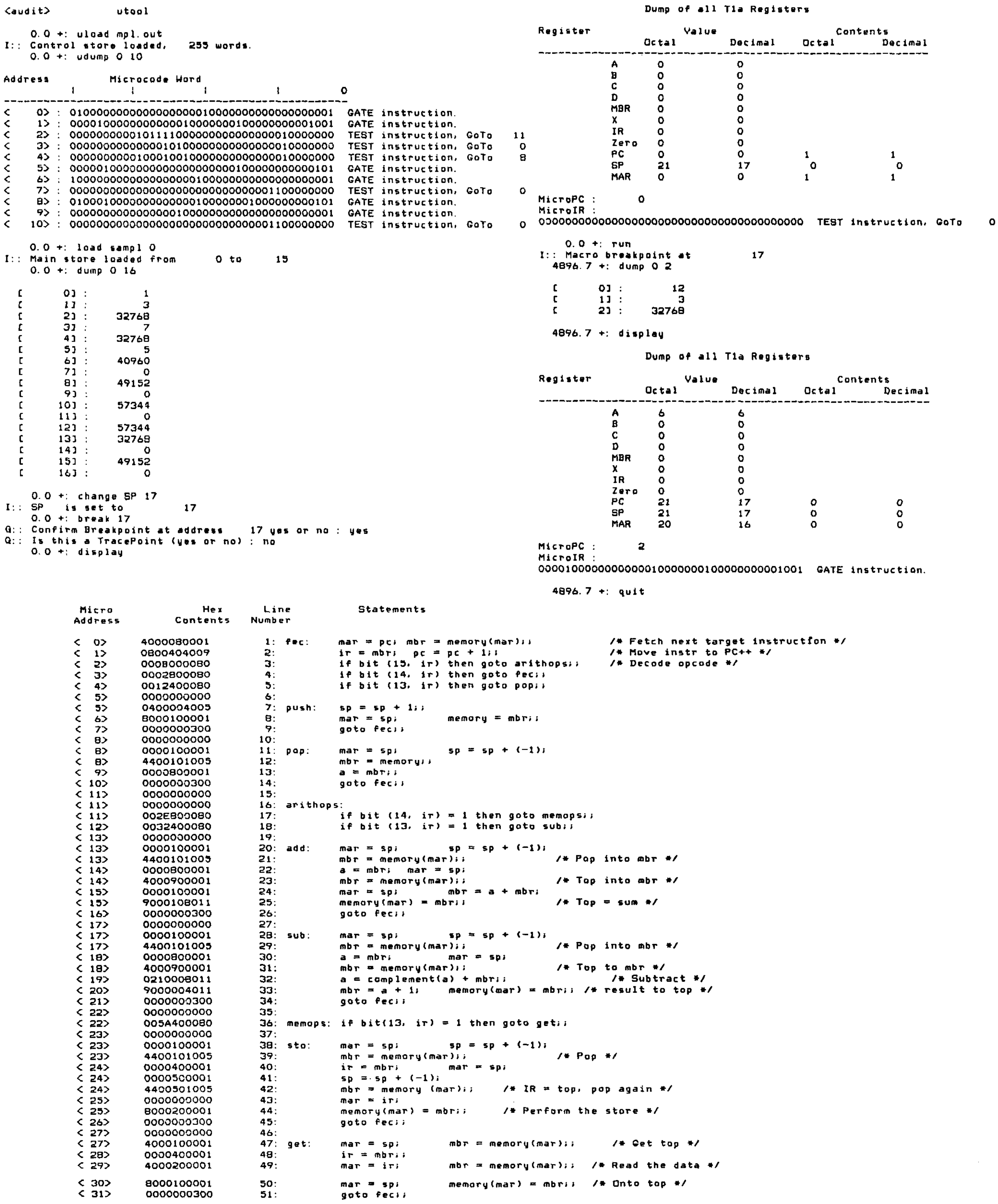
[4] Parker, J.R., "The Microtool Processor Emulation System", University of Calgary Computer Science research report number 82/110/29, January 1983.

[5] IEEE Education Committee (Model Curriculum Subcommittee of the IEEE Computer Society), "A Curriculum in Computer Science and Engineering", preliminary version. Pub. EH0119-8 Jan. 1977.

[6] ACM Curriculum Committee on Computer Science, "Curriculum 68 - Recommendations for Academic Programs in Computer Science", CACM Vol. 11 Number 3, March 1968.

[7] Dasgupta, D., "The Organization of Microprogram Stores", Computing Surveys, Vol. 11 Number 1, March, 1979.

[8] ACM Curriculum Committee on Computer Science, "Curriculum Computer Science", CACM, Vol. 22, No. 3, March, 1979.



```
<audit>         utool

    0.0 +: uload mpl.out
I:: Control store loaded,   255 words.
    0.0 +: udump 0 10

Address          Microcode Word
           |         |           |           |          0
------------------------------------------------------------
<    0> : 0100000000000000000010000000000000000001  GATE instruction.
<    1> : 0000100000000000000100000001000000001001  GATE instruction.
<    2> : 0000000001011110000000000000000010000000  TEST instruction, GoTo    11
<    3> : 0000000000000010100000000000000010000000  TEST instruction, GoTo     0
<    4> : 0000000001000100100000000000000010000000  TEST instruction, GoTo     8
<    5> : 0000010000000000000000001000000000000101  GATE instruction.
<    6> : 1000000000000000000010000000000000000001  GATE instruction.
<    7> : 0000000000000000000000000000001100000000  TEST instruction, GoTo     0
<    8> : 0100010000000000000100000001000000000101  GATE instruction.
<    9> : 0000000000000000100000000000000000000001  GATE instruction.
<   10> : 0000000000000000000000000000001100000000  TEST instruction, GoTo     0

    0.0 +: load sampl 0
I:: Main store loaded from      0 to      15
    0.0 +: dump 0 16

  [      0] :        1
  [      1] :        3
  [      2] :    32768
  [      3] :        7
  [      4] :    32768
  [      5] :        5
  [      6] :    40960
  [      7] :        0
  [      8] :    49152
  [      9] :        0
  [     10] :    57344
  [     11] :        0
  [     12] :    57344
  [     13] :    32768
  [     14] :        0
  [     15] :    49152
  [     16] :        0

    0.0 +: change SP 17
I:: SP   is set to          17
    0.0 +: break 17
Q:: Confirm Breakpoint at address    17 yes or no : yes
Q:: Is this a TracePoint (yes or no) : no
    0.0 +: display
```

```
                    Dump of all T1a Registers

Register              Value                    Contents
               Octal          Decimal    Octal          Decimal
---------------------------------------------------------------
               A      0        0
               B      0        0
               C      0        0
               D      0        0
               MBR    0        0
               X      0        0
               IR     0        0
               Zero   0        0
               PC     0        0          1              1
               SP     21       17          0              0
               MAR    0        0          1              1
MicroPC :       0
MicroIR :
0000000000000000000000000000000000000000  TEST instruction, GoTo    0

     0.0 +: run
I:: Macro breakpoint at          17
   4896.7 +: dump 0 2

   [      0] :       12
   [      1] :        3
   [      2] :    32768

   4896.7 +: display

                    Dump of all T1a Registers

Register              Value                    Contents
               Octal          Decimal    Octal          Decimal
---------------------------------------------------------------
               A      6        6
               B      0        0
               C      0        0
               D      0        0
               MBR    0        0
               X      0        0
               IR     0        0
               Zero   0        0
               PC     21       17         0              0
               SP     21       17         0              0
               MAR    20       16         0              0

MicroPC :       2
MicroIR :
0000100000000000000100000001000000001001  GATE instruction.

   4896.7 +: quit
```

```
 Micro          Hex      Line         Statements
Address       Contents   Number

< 0>        4000080001     1: fec:     mar = pc; mbr = memory(mar);;          /* Fetch next target instructfon */
< 1>        0800404009     2:          ir = mbr;  pc = pc + 1;;               /* Move instr to PC++ */
< 2>        0008000080     3:          if bit (15, ir) then goto arithops;;   /* Decode opcode */
< 3>        0002800080     4:          if bit (14, ir) then goto fec;;
< 4>        0012400080     5:          if bit (13, ir) then goto pop;;
< 5>        0000000000     6:
< 5>        0400004005     7: push:    sp = sp + 1;;
< 6>        8000100001     8:          mar = sp;         memory = mbr;;
< 7>        0000000300     9:          goto fec;;
< 8>        0000000000    10:
< 8>        0000100001    11: pop:     mar = sp;         sp = sp + (-1);
< 8>        4400101005    12:          mbr = memory;;
< 9>        0000800001    13:          a = mbr;;
< 10>       0000000300    14:          goto fec;;
< 11>       0000000000    15:
< 11>       0000000000    16: arithops:
< 11>       002E800080    17:          if bit (14, ir) = 1 then goto memops;;
< 12>       0032400080    18:          if bit (13, ir) = 1 then goto sub;;
< 13>       0000000000    19:
< 13>       0000100001    20: add:     mar = sp;         sp = sp + (-1);
< 13>       4400101005    21:          mbr = memory(mar);;              /* Pop into mbr */
< 14>       0000800001    22:          a = mbr;  mar = sp;
< 14>       4000900001    23:          mbr = memory(mar);;              /* Top into mbr */
< 15>       0000100001    24:          mar = sp;         mbr = a + mbr;
< 15>       9000108011    25:          memory(mar) = mbr;;              /* Top = sum */
< 16>       0000000300    26:          goto fec;;
< 17>       0000000000    27:
< 17>       0000100001    28: sub:     mar = sp;         sp = sp + (-1);
< 17>       4400101005    29:          mbr = memory(mar);;              /* Pop into mbr */
< 18>       0000800001    30:          a = mbr;          mar = sp;
< 18>       4000900001    31:          mbr = memory(mar);;              /* Top to mbr */
< 19>       0210008011    32:          a = complement(a) + mbr;;             /* Subtract */
< 20>       9000004011    33:          mbr = a + 1;    memory(mar) = mbr;; /* result to top */
< 21>       0000000300    34:          goto fec;;
< 22>       0000000000    35:
< 22>       005A400080    36: memops: if bit(13, ir) = 1 then goto get;;
< 23>       0000000000    37:
< 23>       0000100001    38: sto:     mar = sp;         sp = sp + (-1);
< 23>       4400101005    39:          mbr = memory(mar);;              /* Pop */
< 24>       0000400001    40:          ir = mbr;         mar = sp;
< 24>       0000500001    41:          sp = sp + (-1);
< 24>       4400501005    42:          mbr = memory (mar);;    /* IR = top, pop again */
< 25>       0000000000    43:          mar = ir;
< 25>       8000200001    44:          memory(mar) = mbr;;     /* Perform the store */
< 26>       0000000300    45:          goto fec;;
< 27>       0000000000    46:
< 27>       4000100001    47: get:     mar = sp;         mbr = memory(mar);;     /* Get top */
< 28>       0000400001    48:          ir = mbr;;
< 29>       4000200001    49:          mar = ir;         mbr = memory(mar);;  /* Read the data */

< 30>       8000100001    50:          mar = sp;         memory(mar) = mbr;;  /* Onto top */
< 31>       0000000300    51:          goto fec;;
```

Figure 3 - MPL Listing of Stack Machine