An Integrated Hardware Simulator

Delmar E. Searls
Asbury College
1 Macklem Drive
Wilmore, KY 40390-1198
(606)-858-3511 x2238

Introduction

Computer systems courses include a discussion of hardware and programming at a number of levels often using a simplified example as a focus of the discussion. Andrew S. Tanenbaum's widely adopted text, *Structured Computer Organization*, utilizes this approach with the Mic-1 microarchitecture and Mac-1 machine language levels. While simulators for this example machine have been available for a number of years they tend to concentrate on the microarchitecture. Over the years we have developed a menu-driven simulator for IBM-compatible machines that, we believe, offers a uniquely integrated approach that includes micro-programming, machine language programming, and assembly language programming. The utility of the simulator can even be extended to programming in a simplified version of Pascal.

The Microarchitecture Level

The user of the simulator has full access to the 256 x 32 bit control store. When the program is first run, a slightly modified version of Tanenbaum's microprogram (which interprets his Mac-1 machine language instruction set) is loaded by default. The simulator includes a built-in editor (see figure 1) that allows the user to view and/or modify the contents of the control store. A help screen is available to remind students which codes to use in which fields to

```
 Control    M  C   A       M  M        E          [F1] = Help
 Address    U  N   L       B  A  R  W  N
Dec   Hex   X  D   U   SH  R  R  D  R  C  C     B     A     ADDR
----  --    -  --  --  --  -  -  -  -  -  ----  ----  ----  ----------
   0  00  +-------------- Microinstruction Fields ------------------+
   1  01  |                                                         |
   2  02  | AMUX:  0 => A latch     COND: 00 => No jump             |
   3  03  |        1 => MBR               01 => Jump if N=1         |
   4  04  |                               10 => Jump if Z=1         |
   5  05  |                               11 => Jump always         |
   6  06  |                                                         |
   7  07  |  ALU: 00 => A + B         SH: 00 => No shift            |
   8  08  |       01 => A and B           01 => Shift right         |
   9  09  |       10 => A                 10 => Shift left          |
  10  0A  |       11 => not A             11 => (not used)          |
```

Figure 1. Part of a screen snapshot of the control store editor with the help screen superimposed on top. Data is entered in binary in the appropriate columns. (The screen snapshots here and below do not faithfully duplicate the text mode line graphics of the actual screen display.)

accomplish which tasks. The control-store can be totally cleared, allowing students to write their own microprograms from scratch. For example, a student might try to write a microprogram that interprets a two-address instruction set rather than the Mac-1 instruction set provided by Tanenbaum. The contents of the control store can be saved to disk and reloaded at any time.

Initially, we ask our students to write rather simple microprograms such as adding the value in main memory location zero to memory location one and store the result in memory location two. To emphasize the ability to do several things at once at the microprogramming level, we make a contest of it by awarding bonus points to those who can do the job with the fewest number of microinstructions. Even a simple assignment like this gives the students more of a feel for the flexibility of microinstructions than does a reading of the text alone.

To facilitate assignments like this (as well as machine language programming), a second built-in editor is provided expressly for the purpose of viewing and modifying main memory. As with the control store, the contents of main memory can be saved and reloaded at any time.

During the execution of a microprogram the movement of data through the CPU is graphically displayed on the screen (see figure 2). CGA text mode graphics are used so the program should run on any MS-DOS-based machine. In Mic-1, microinstruction execution is broken into four subcycles: (1) load the microinstruction into the microinstruction register, (2) gate data from registers to the A and B latches, (3) produce a stable output in the ALU and shifter and (4) store the output (if necessary) in an internal register and/or the memory data register (MBR). The simulator can run a microprogram in a number of different modes, the first of which single steps

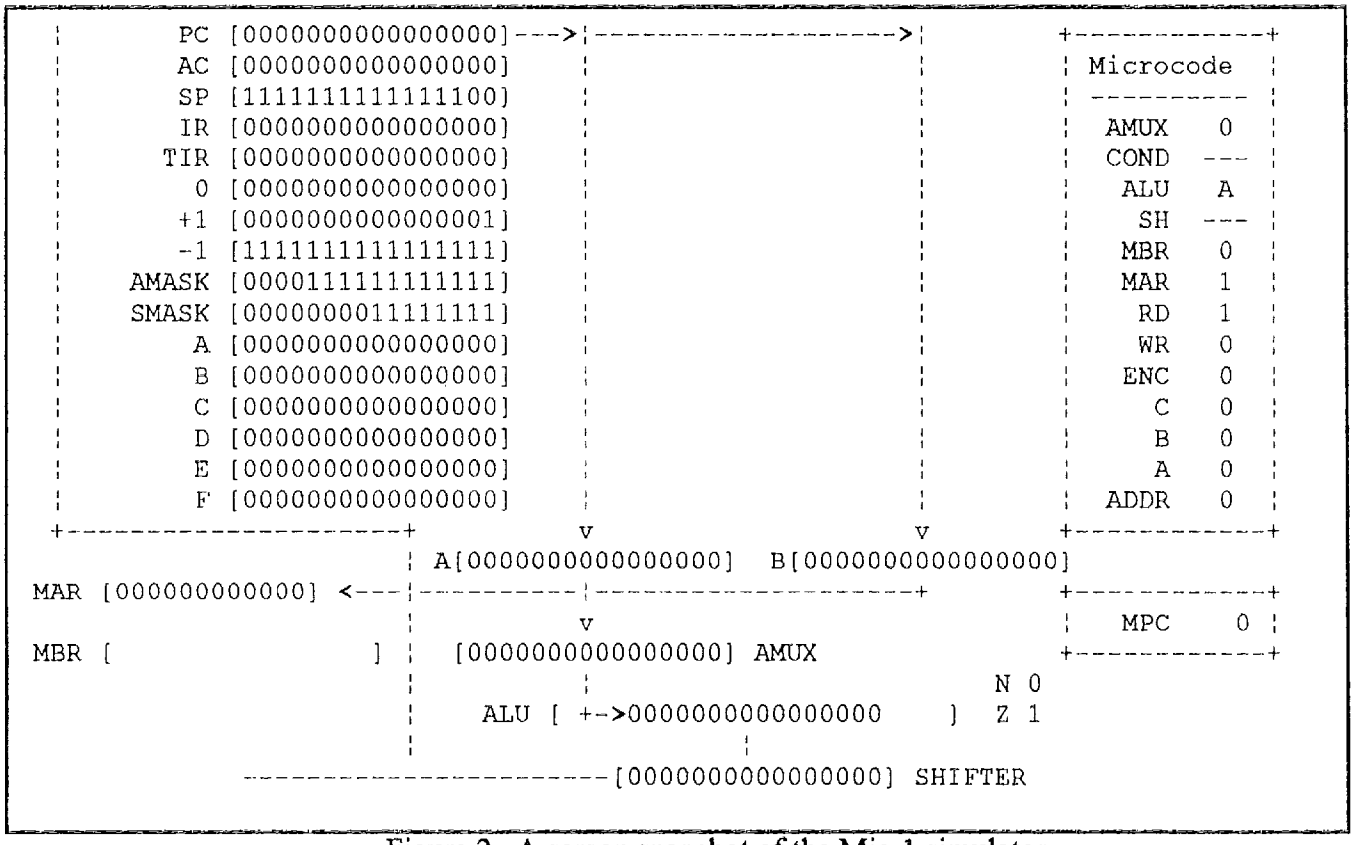

Figure 2. A screen snapshot of the Mic-1 simulator.

through the program one subcycle at a time. This allows the student to clearly understand what happens during each subcycle as the movement of data is illustrated on the screen.

The second, more common, execution mode single steps through the program one complete microinstruction at a time. This mode is very useful in helping students understand how the machine language interpreter works or for debugging their own microprograms.

A third instruction mode single steps one machine language instruction at a time. (Actually, it pauses every time the microinstruction program counter reaches zero but since the machine language interpreter always goes to microinstruction zero to load the next machine language instruction, the net effect is that the simulator single steps through machine language programs. It would continue to do so even if some other machine language instruction set were implemented, as long as the loading of the next machine language instruction begins at zero in the microprogram!)

The final execution mode does no single-stepping at all and just runs the microprogram to completion. A microprogram is considered complete when the microprogram counter points to a null (all zero) instruction.

During microprogram execution, the user can freely switch from one execution mode to another or even terminate program execution entirely.

The Machine Language Level

Since the student can modify main memory, as indicated above, the microarchitecture simulator can be used as a machine language simulator as well. But the program also includes a separate Mac-1 simulator (see figure 3). By

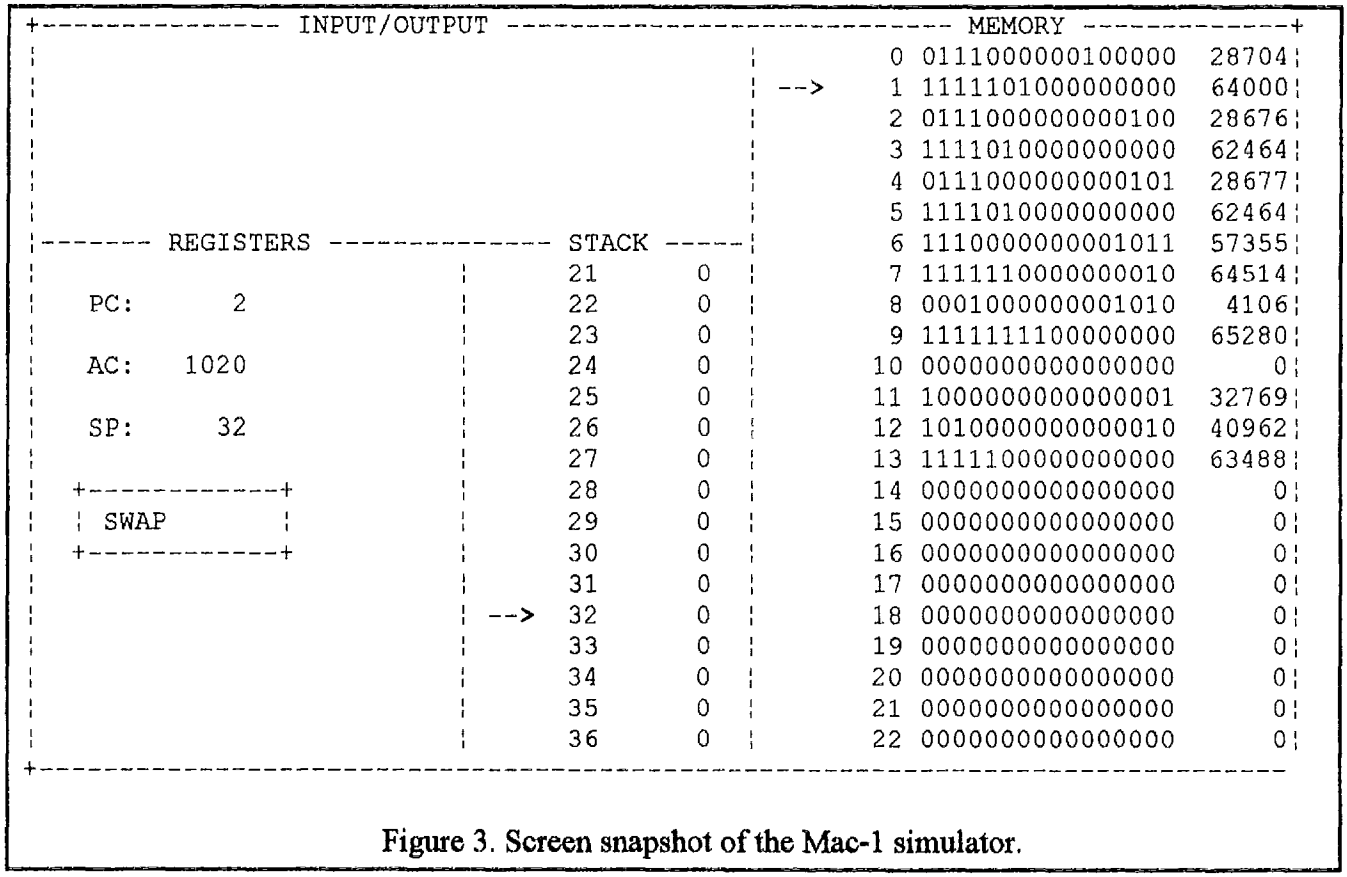

Figure 3. Screen snapshot of the Mac-1 simulator.

using the memory editor, the student can write machine language programs in binary (see Figure 4). A help screen containing machine language formats and an explanation of each instruction can be called up while editing memory.

```
Main Memory
 Address          Machine Language
Dec   Hex           Instruction
----  ---   ----------------------
   0  000 : 0000  0000  0000  0110
   1  001 : 0010  0000  0000  0111
   2  002 : 0001  0000  0000  1000
   3  003 : 0011  0000  0000  0111
   4  004 : 0001  0000  0000  1001
   5  005 : 1111  1111  0000  0000
   6  006 : 0000  0000  0000  1000
   7  007 : 0000  0000  0000  0011
   8  008 : 0000  0000  0000  0000
```

Figure 4. Part of a screen snapshot of the memory editor.

When running the Mac-1 simulator, the user can see the contents of the three registers available to the machine language programmer (program counter, accumulator, and stack pointer), the region of memory containing the instruction currently being executed, and the region of memory pointed to by the stack pointer. As the program counter and stack pointer change values these memory windows dynamically adjust to show the corresponding memory regions. A separate input\output window allows the user to program memory-mapped I/O.

The Assembly Language Level

There are a number of important concepts that can be introduced using the Mac-1 machine language instruction set (e.g. parameter passing via the stack). Since coding longer programs in binary is tedious and perhaps counterproductive, the simulator also includes a built-in assembler and an assembly language editor (see Figure 5). The assembler is not fancy (no macros, for example) but fully functional and includes an EQU directive along with DCLS (declare storage) and DCLW (declare word) directives. A HALT statement has been added to the machine language instruction set to provide an orderly termination of a program.

The assembler requires a fixed-format source and generates the corresponding binary machine language code. As with the control store and main memory, assembly language programs can be saved to and loaded from the disk as ASCII text files. The assembler is a traditional two-pass assembler and the generated listing can be sent to the screen or to a printer. Error messages are inserted directly into the source code and will be automatically removed when the corrected code is re-assembled. The user can access a help screen that lists the assembler mnemonics and describes the meaning of each.

The simulator allows the user to print out a copy of the current assembly language program, the contents of the control store, or the contents of main memory; each in a nicely formatted style. When printing the contents of either type of memory, only data up to (and including) the last nonzero value is printed. This prevents wasting a lot of paper when only a small fraction of memory is actually used.

```
File:  CALLSUB.ASM                                        [F1] = HELP

LINE  LABEL   OP   OPRND  COMMENTS
----  ------  ---- ------ --------
   0  ************************************************
   1  * CALLSUB.ASM
   2  *
   3  * THIS PROGRAM TESTS AND ILLUSTRATES THE SUBROUTINE
   4  * CALLING MECHANISM OF MAC-1 AND THE PASSING OF
   5  * VALUE PARAMETERS ON THE STACK.
   6  ************************************************
   7
   8         LOCO 32
   9         SWAP          SP := 32
  10         LOCO 4        --*
  11         PUSH            | PUSH PARAMETERS
  12         LOCO 5          | ONTO THE STACK
  13         PUSH          --*
  14         CALL ADDSB    CALL ADD SUBROUTINE
  15         INSP 2        REMOVE PARAMETERS
```

Figure 5: Part of a screen snapshot of the assembly language program editor.

The Pascal Level

We follow our computer systems course with a systems programming course and we continue to use the simulator as our target machine. We progress through a number of programming assignments (done as team projects) that allow the students to apply what they are studying in class. Each assignment is a stand-alone project.

The first project is an assembler which translates the Mac-1 assembly code into an object file format. The program duplicates the work of the simulator's assembler except for the format of the output. Another difference is that the students' assembler must provide for global defines and external references. The teams are provided with a linker that will convert their object files to machine language files which can then be loaded into the main memory of the simulator and tested. Asking them to write a functional clone (of the simulator's assembler) is helpful in that they have a model to follow and are already familiar with most of the features they are expected to include. Getting through this project helps give them confidence that they can complete the remaining projects.

After completing the assembler, the students are required to write a macro processor. (Actually, it would be more appropriate to call it a pre-processor.) An assembly language program containing macro definitions and macro invocations is supplied as the source file to the macro processor. The output is an assembly language program with the macro invocations expanded into assembly language code. This output can then be used as input for the assembler.

The third project involves the writing of a linker. This is their implementation of the linker they have already been using in the previous two projects. The input is in the form of one or more object files and the output is an executable machine language file.

The next three projects together comprise a compiler for a greatly simplified version of Pascal. The students first write a tokenizer that takes a Pascal source file and creates a tokenized file. Then they write a grammar checker that takes the tokenized file as input and checks for syntax errors. The final component is a code generator that takes the tokenized file and generates a Mac-1 assembly language output file that (as you have already guessed) can be assembled by the students' assembler and linked with a Pascal library object file by their linker and run on the simulator.

Conclusion

Our students have remarked that the highly integrated hardware simulator has been helpful to them in learning the concepts presented in Tanenbaum's text. They appreciate the continuity of example hardware as they move into the systems programming course. I am making the simulator available to a broader audience in the hope that others will find it equally useful. If you would like a copy of the program just write to me. I will require $5.00 to cover the cost of the disk, the mailer, and postage. You may freely copy the program for distribution to students and/or colleagues.

The simulator is less than 100K in size and requires nothing fancier than a CGA color display on an IBM-PC or compatible. A reference manual, a tutorial, and sample programs are also included on the disk.