A Microprogramming Animation

Steven Robbins

January, 1995

Abstract. This paper describes a successful project using computer animation to teach the concepts of microprogramming to lower division computer science majors. The students write a simulator for the Mic-1 horizontal microcontroller described in the book *Structured Computer Organization* by Andrew Tanenbaum. The simulation is enhanced by the use of a graphical representation of the machine which animates the simulation. This creative use of computer animation enables the students to see the results of their simulation without having to write an extensive user interface. They can concentrate on implementing the instruction cycle which directly enhances their understanding of the Mic-1. The XTANGO animation package is used, making this portable to any Unix system with an X display. The user interface is available via anonymous ftp.

Technical Report CS-95-9

Division of Computer Science
The University of Texas at San Antonio
San Antonio, TX 78249

1 Introduction

The University of Texas at San Antonio Computer Science Program offers a sophomore-level course called Computer Organization II which is required of all undergraduate computer science majors. We use the text *Structured Computer Organization* by Tanenbaum [8]. The background of the students taking this course includes an introductory course in which they learn to program in C, a data structures course, also in C, and a circuit design course. The course covers CPU organization, the instruction cycle, memory, microprogramming, instruction formats and types, addressing, assembly language and RISC machines. We use the Intel 80x86, the Motorola 680x0 and the SPARC microprocessors as case studies. The course includes a segment on assembly language programming using the SPARC assembly language [5].

The most difficult material in the course is the microprogramming covered in the Tanenbaum book in chapter 4. Students who understand microprogram control develop a fundamental understanding of how a CPU executes a program. This understanding is important for later courses in operating systems and architecture. Unfortunately few students are able to master the material from lecture presentations alone. They are confused by the presence of two instruction cycles (microprogram and conventional machine level) and have difficulty understanding the timing of the microprogram instructions themselves. Students clearly need hands-on experience with this material. Exercises in the Tanenbaum book include translation of assembly language into microcode and addition of new assembly language instructions to the machine. Assignments of this type do not give students a sense of the execution timing.

The traditional approach to overcoming these pedagogical difficulties is for the instructor to provide hardware or a simulator for student experimentation [1, 2, 3, 4, 6]. The student then views the state of the machine during execution. Fuchs et al. [3] describe a microprogramming simulator which they have successfully used to enhance learning in an introductory computer engineering course. They introduce an instructional microprocessor simulation which has a graphical user interface. The students can write assembly language or microcode for this target machine and the tool simulates its execution. Their approach is a definite improvement over pencil and paper simulation and is particularly useful if the goal is to teach students how to write efficient assembly language or microcode.

The goal of a sophomore level computer science course in computer organization is for the students to develop a fundamental understanding of how computers work. In particular, the goal is to achieve an understanding of the instruction cycles of a microcoded Von Neumann machine. At this level, a simulator like the one in [3] does some of the thinking for the students. We have used an alternative approach in which the students write their own simulation. Because the instruction cycle is the key concept in this material, I provide

an animation environment using a standard freely-available animation package which allows students to do a full-instruction cycle implementation in a few hundred lines of code. The simulator is organized so that it can be implemented in stages with well-defined milestones. The animated output allows students to see the execution of the machine as they develop code. The project is so clearly-defined that most students are able to complete the simulation. Student comments were extremely positive about the experience. The unique advantages of the approach described here are:

- The students write the part of the simulator which requires fundamental understanding of the process they are trying to understand.
- The students do not have to worry about the details of displaying the information on the screen.
- The machine to be simulated is described in a standard textbook.
- The simulator is animated and shows the movement of data in the machine.
- Since the students write their own simulator, they can use their creativity to produce something which is uniquely their own and in which they can take pride.
- The animations produced by the students look professional and the students enjoy working on them.
- The hard part of the simulation is provided by the instructor and is readily available for use by others via anonymous ftp. Other instructors who use the Tanenbaum book can easily incorporate this into their courses.

The organization of the paper is as follows. In Section 2 an overview of the simulator is given. The animation interface is described in Section 3, and class assignments are described in Section 4. A discussion of the results is in Section 5. Some examples of Mic-1 and Mac-1 execution are provided in the appendix for readers who are unfamiliar with Tanenbaum's text.

2 Overview of the Simulator

A CPU may have its control function implemented entirely in hardware. Alternatively, the machine instructions may be interpreted by a simpler machine which directly controls the timing and flow of data within the CPU. The program which runs on this simpler machine is called a *microprogram* or the *control store* of the CPU. Insertion of a different control store results in a machine with different machine (assembly language) instructions. The

execution of the microprogram consists of loading instructions from the control store into the microinstruction register (MIR). This register directly controls the flow of information in the machine.

Tanenbaum describes a simple horizontal microcode machine called the Mic-1 shown in Figure 1. The machine executes a microprogram which is stored in the control store. The subprograms in the control store correspond to individual instructions at the conventional machine level (the assembly language of the machine). Tanenbaum implements a complete assembly language called Mac-1 in the Mic-1 control store. More detailed descriptions of the Mic-1 and Mac-1 with examples are given in the appendix.

Figure 1: The Mic-1 machine as it appears in the Tanenbaum text. (Permission from Prentice Hall for publication is pending.)

The material on microprogramming is quite complicated and requires an understanding of the conventional instruction cycle of a Von Neumann machine. Students find this difficult, in part because there are two instruction cycles to keep track of, one at the microcode level and one at the assembly language level. We have taught this material for several years, never feeling satisfied with the students' grasp of this material. Although they seem to understand the lecture presentations and could do the assigned problems, fundamental understanding is often lacking. We had considered implementing a Mic-1 simulator for the students to use, but there just was not enough real estate on the screen of an ASCII terminal to make this feasible.

Several years went by in which I was involved in other teaching projects, and so I did not teach this course for a while. In fall 1994 I was assigned to teach the course again. In the meantime the course had moved from using ASCII terminals connected to a VAX to Sun workstations. The availability of megapixel displays solved the real estate problem. Two options were available — write a simulator for student experimentation or have the students write the simulator as a class assignment. The problem with the latter approach is that most of the code for such a simulator involves boring routines to display the state of the simulated machine. While this might be good for teaching programming techniques, it is not directly related to the course. Also, it would be nice to let the students take advantage of the graphics capabilities of the workstations, but the programming required normally far exceeds the capabilities of the students at this level. These difficulties were overcome by using the XTANGO [7] computer animation system as the display vehicle for the simulations.

The idea of a Mic-1 simulator is quite simple since the machine itself is simple. The basic simulator is a program which takes two command line arguments, a control store and a memory file. The first file is a representation of the 79-word control store used in the Mic-1 machine. The second file represents the memory of the Mac-1 machine and contains the assembled Mac-1 assembly language program to execute along with its data. The simulator initializes certain data structures and then goes into a loop:

```
while(1) {
   subcycle_1();
   subcycle_2();
   subcycle_3();
   subcycle_4();
}
```

Each of the subcycle routines performs the operations corresponding to that subcycle. For example, the routine `subcycle_1()` might just consist of:

```
MIR = decode_MIR(control_store[MPC]);
```

where `control_store` is an array of 32-bit quantities and `MIR` is a structure of the type:

```
typedef struct {
    int addr;
    int Areg;
    int Breg;
    int Creg;
    int ENC;
    int WR;
    int RD;
    int MAR;
    int MBR;
    int SH;
    int ALU;
    int COND;
    int AMUX;
} MIR_decoded;
```

The `decode_MIR` routine just shifts and masks its argument to produce the required fields.

Of course to make this simulator useful, there must be some interaction with the user. The user should be able to single step through the execution of the Mic-1 at the three levels of Mic-1 subcycle, Mic-1 cycle, and Mac-1 instruction cycle. Most of the code for the simulator involves displaying what is going on in a useful way.

3 Animation Interface

The XTANGO animation package is a powerful tool for showing the dynamic behavior of a program. A simplified interface to XTANGO is provided by the `animator` which was written by John Stasko of Georgia Tech [7] specifically for use by undergraduates doing computer animation. This program simply reads ASCII text, one command per line. Since the `animator` 's input comes from standard input, a C program can send commands to it with simple `printf` statements. The simulator might be run by the following command:

```
mic1 control.store assembly.prog | animator
```

The simulator executable is called `mic1`. The file `control.store` contains the control store and the contents of the main memory are held in the file `assembly.prog`. The standard output of the simulator is redirected to standard input of the `animator` through the pipeline `|`.

The `animator` allows for drawing text, lines, rectangles, triangles, and circles in various colors. For example, the following `animator` command would create a rectangle on the screen:

```
rectangle 1234 0.3 0.2 0.4 0.5 blue outline
```

Positions and distances in the default `animator` display are referenced with rectangular coordinates in the range from 0.0 to 1.0. The above command creates a rectangle whose lower left corner has coordinates (0.3, 0.2) and whose width and height are 0.4 and 0.5 respectively. The outline of the rectangle is displayed in blue. The rectangle can be referenced by its ID, 1234. The `animator` provides commands to move objects smoothly from one location to another making animation relatively painless. For example, the `animator` command:

```
moveto 1234 0.4 0.7
```

moves the rectangle with ID 1234 so that its lower left corner now has with coordinates (0.4, 0.7).

The `animator` interface has been successfully used in our upper division Analysis of Algorithms course, but it is too complicated for the lower division Computer Organization students given the complexity of the display to be produced. We solved this problem by writing an `initialize_animator` routine which draws and labels all of the boxes and lines in the diagram. The sample `animator` display from the first version of the Mic-1 simulator shown in Figure 2 corresponds to the drawing from the Tanenbaum book shown in Figure 1. Color is automatically used on a color display. Approximately 40 additional routines are provided to fill values in the various boxes and to move values from one place to another.

These library routines hide the details of the animation from the students. A sample of some of these library routines is shown in Table 1. Their functions should be evident from their names with a small amount of explanation. For example, the routine:

```
ani_move_value_from_reg_to_alatch(value,reg);
```

takes the given value, highlights it in the given register, smoothly moves it to and then along the A bus to the A latch, and changes the value displayed in the A latch.

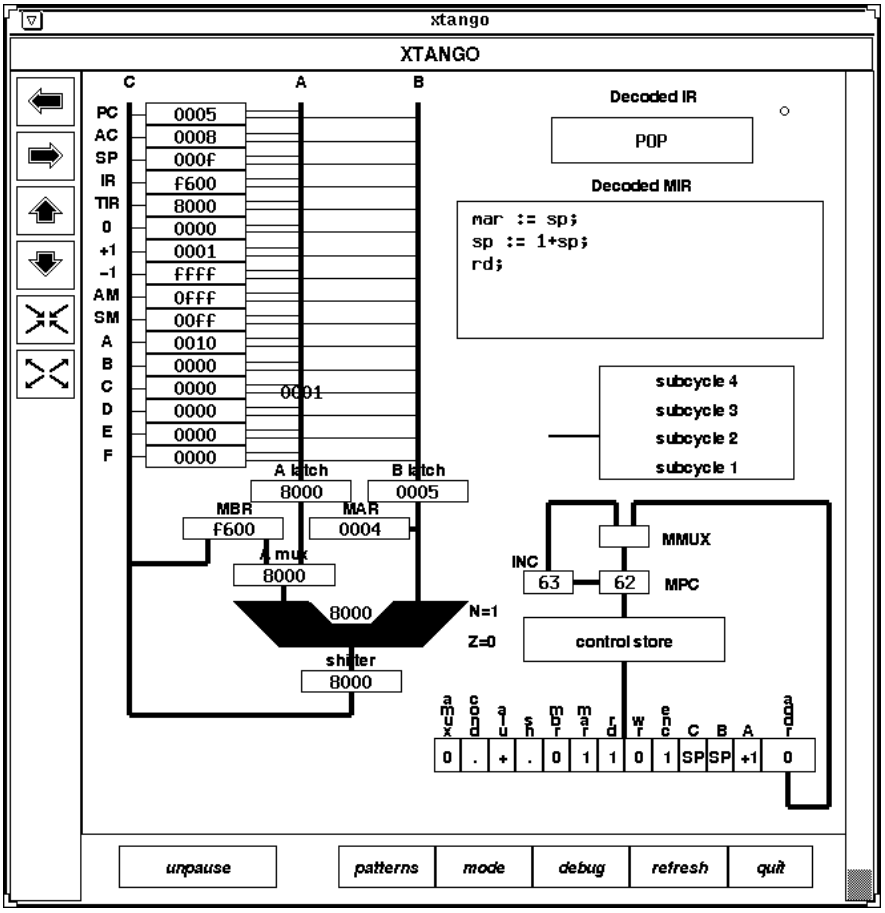

Figure 2: The animator display for the first version of the simulator.

The students are shown a very simple Mic-1 simulator for demonstration purposes so that they can see what is expected of them. The sample simulation allows for single stepping through the Mic-1 subcycles, dynamically showing the execution of the Mic-1 machine. For

`void ani_set_mir_A_value(int val);`
`void ani_set_mir_amux_value(char ch);`
`void ani_set_register_value(int value, int reg);`
`void ani_move_value_from_reg_to_alatch(int value, int reg);`
`void ani_move_value_from_blatch_to_mar(int value);`
`void ani_move_value_from_mbr_to_memory(int value, int address);`

Table 1: Some of the routines used to modify the animator display.

example, on the second Mic-1 subcycle, data in one of the registers is highlighted, moved from the register to the A bus and down the A bus into the A latch. Data in another register is highlighted and moved to the B bus and down into the B latch. The entire program (excluding the animator library) can be implemented in about 300 lines of code.

Figure 2 shows the animator output. The snapshot of the display was taken in the middle of executing the second subcycle of the first Mic-1 instruction which executes the Mac-1 POP instruction. The details of this execution are described in the appendix. The simulator displays the decoded MIR by calling a routine from the animation library package for each field of the MIR. For example:

```
ani_set_mir_A_value(6);
```

displays $+1$ in the A field of the MIR, since the animator library knows that register 6 of the Mic-1 contains the constant $+1$. The instruction POP is shown in the decoded IR box of the display with the command:

```
ani_set_decoded_ir(registers[IR]);
```

The animator library knows about the Mac-1 assembly language and can disassemble its instructions. The display in the decoded MIR box can be set with:

```
ani_set_decoded_mir(control_store[MPC]);
```

In subcycle 2, the routines:

```
ani_move_value_from_reg_to_alatch(registers[MIR.Areg],MIR.Areg);
ani_move_value_from_reg_to_blatch(registers[MIR.Breg],MIR.Breg);
```

are used. The first of these highlights the value in register 6, which is 0001 and shows it moving to the A bus, down the A bus, and into the A latch. The A latch is then filled with that value. The snapshot of Figure 2 was captured just as this value was to enter the A latch.

4 Class Assignments

For the first assignment, the class is divided into small groups and each group is responsible for hand-assembling ten instructions in the control store. By requiring that all don't-care bits be zero, there is only one correct answer for each word of microcode. The students hand in their results by email and these answers are collected into a single file to form the control store input for the simulator.

The simulator assignment consists of writing a basic simulator which allows for single stepping at each of three levels: Mic-1 subcycle, Mic-1 cycle, and Mac-1 instruction cycle. The simulator program essentially consists of a loop which, aside from handling standard input, calls the four procedures to handle the Mic-1 subcycles. For example, subcycle one consists of taking the element from the control store array indexed by the MPC and decoding it. The decoding involves shifting and masking so that each bit field of the MIR can be stored in the appropriate field of an MIR structure. The simulator shows each field on the display by calling a routine such as `ani_set_mir_A_value(value,reg)`.

The basic simulator assignment is divided into milestones. The first milestone consists of writing a main program which just calls the `initialize_animator` routine and is executed with its standard output redirected to the `animator`. These few lines of code allow the students to see how easy it is to bring up the graphical interface. The second milestone consists of setting up and initializing the data structures used for storing the state of the Mic-1 machine and sending the initial values of the sixteen registers to the `animator`. At this point students have written very little code, but they can see how the animation will progress. They can try out several of the routines for controlling the graphical display and get a feel for what will be involved in the simulation. The milestones progress in simple steps until the entire simulator is complete.

The simulator takes simple input commands for single stepping and for turning on and off the dynamic updating of the display. Even with fast hardware, it takes several seconds to show a full Mac-1 instruction cycle so for testing more than a few Mac-1 instructions, it is convenient to not show the detailed execution at each Mic-1 subcycle. The simulator can execute about a thousand Mac-1 instructions per second with the display updating turned off when run on a low-end Sun workstation.

The Mac-1 assembly language is cumbersome for manipulating arrays since it does not have indirect addressing except through the stack pointer and accumulator. In the next assignment students add a number of instructions for indexed addressing. The new assembly language is called the Mac-1a. The students are given a Mic-1 assembler at this point and told to modify the control store to implement the new instructions. The opcode encoding for the new instructions is left to the students and they are challenged to find the most efficient encoding and control store program which maintains binary compatibility with the old ma-

chine. This leads to a good discussion of the consequences of binary compatibility over several generations. They also make a few minor modifications to their simulator to keep track of the number of Mic cycles and Mac instructions executed. Since the `animator` displays the decoded instruction register, some routines are included so that the student's simulators can communicate the information about the new Mac instructions to the `animator`.

One of the problems with the original simulator was the display of the Mac-1 memory. It was left to the students to display this memory with an appropriate command to their simulator. This was done with text sent to standard error and was independent of the graphical display. Towards the end of the semester, I modified the `animator` display so it could show a part of the Mac memory as well as the cycle and instruction counts. The new animator display is shown in Figure 3.

5 Discussion

The students are encouraged to add features to the basic simulator. Some of the features added by the students or suggested by the instructor include:

- Execute until a Mac-1 fetch and decode are complete.
- Modify the control store so that the simulator stops when a particular Mac-1 opcode is detected.
- The ability to unexecute instructions.
- Allow breakpoints.
- The ability to modify the memory while the simulator is running.
- Loading information about the assembly language encoding from a file.
- Analyze the program before execution. That is, run the code for a while and determine where in memory the program is and what memory locations are accessed. This allows for displaying the memory locations in a format appropriate to their use.
- Add specific additional Mac-1 instructions chosen by the instructor, keeping binary compatibility with the original Mac-1 machine.
- Add general purpose Mac-1 instructions chosen by the students which are designed to make certain functions efficient, such as accessing arrays.
- Add an interrupt capability.
- Add (memory mapped) I/O capability.

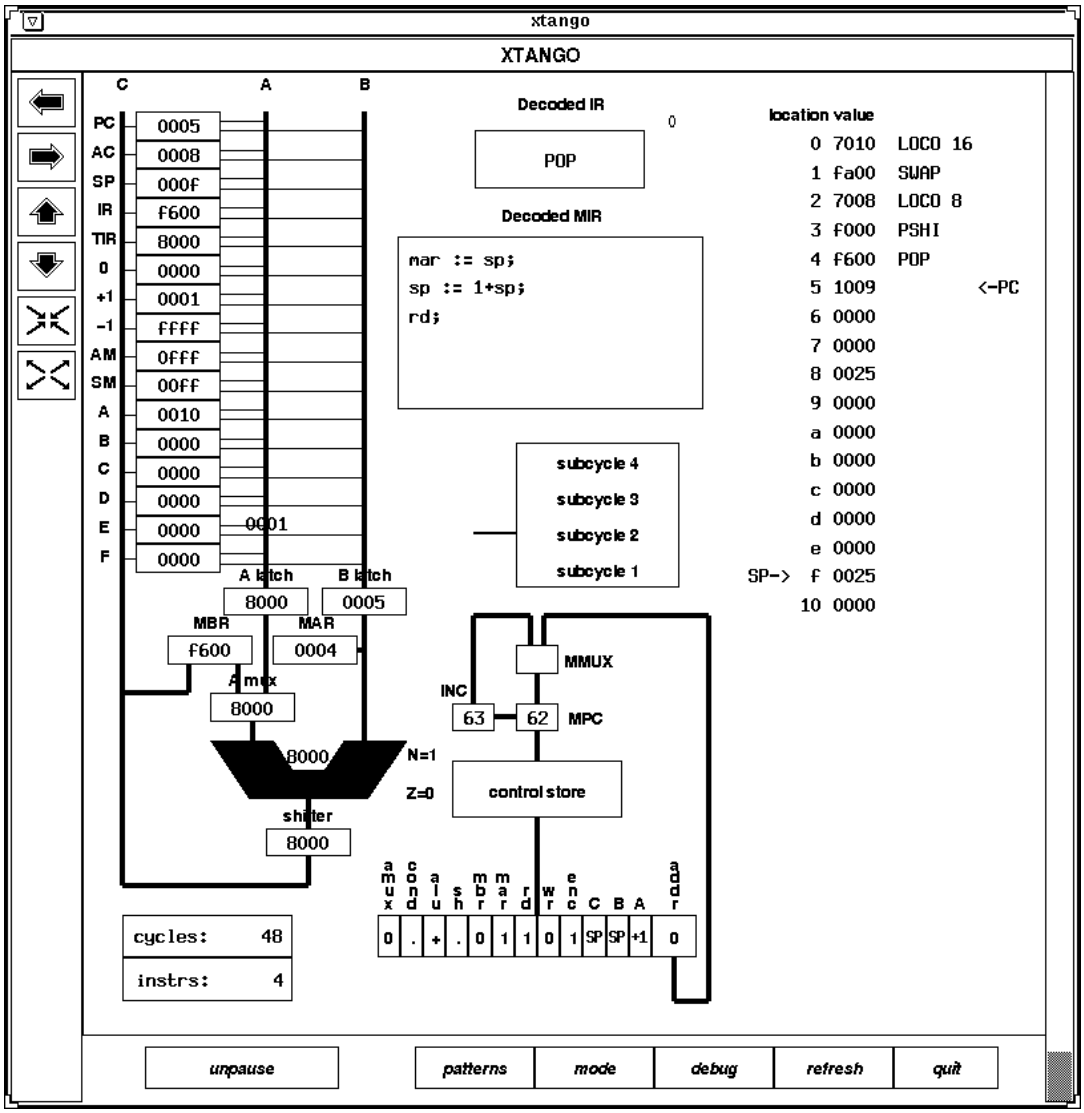

Figure 3: The animator display for the second version of the simulator.

In addition, the simulator can be modified to simulate other microcoded machines. Some of the possible changes include:

- Change word length.
- Change the memory access time.
- Add instruction decoding hardware.

A survey was distributed on the last day of class. Students indicated that they thought the simulator project enhanced their understanding of microprogramming and how computers worked in general. They also found the project interesting and enjoyed working on it. The latter may be the most important benefit in that students learn best when working on something they enjoy.

Other instructors are encouraged to use this simulator project. The following are available via anonymous ftp [9]. Essentially everything is available except for the simulator source code which the students should write.

- A simple Mic-1 assembler for generating the control store for the simulator.
- The source code for the XTANGO `animator` routines for the Mic-1 simulator.
- A makefile for compiling and linking the simulator and `animator` routines.
- A simple Mac-1 assembler for generating the memory image for use by the simulator.
- Copies of the assignments given to the students in one semester.
- A simple Mic-1 simulator executable for Sun Solaris 2.
- A simple Mic-1 simulator executable for Linux.
- Information about obtaining the XTANGO package.

Appendix: Details of the MIC-1 and MAC-1 Machines

This appendix provides an overview of the Mic-1 and Mac-1 from Tanenbaum's book. Refer to Figure 1 in the text during the discussion.

The Mic-1 instruction cycle consists for four subcycles. On the first subcycle the microprogram counter (MPC) picks out one of the 256 possible words in the control store, and this value is stored in the microinstruction register (MIR). On the second subcycle one of the sixteen registers is stored in the A latch and one in the B latch. The A and B fields of the MIR determine which registers are chosen. The AMUX bit in the MIR determines

whether the left input to the ALU comes from the memory buffer register (MBR) or the A latch. The ALU bits control the ALU functions (add, and, pass A, complement A) and the SH bits control the Shifter which can shift one bit to the left or right, or not shift at all. The result of this combinational logic is assumed to be available by the start of subcycle 4. In the meantime, during subcycle 3 the low 12 bits of the B latch are stored in the memory address register (MAR) if the MAR bit of the MIR is set.

On subcycle 4, the result from the Shifter is stored in the MBR if the MBR bit of the MIR is set. The result is also stored in the register specified by the C field of the MIR if the ENC (enable C bus) bit is set. Additionally on subcycle 4, the COND bits of the MIR along with the negative (N) and zero (Z) flags of the ALU determine the next address to be loaded into the microprogram counter (MPC). The options are to increment the MPC or to use the branch address stored in the ADDR field of the MIR. The possibilities are branch never, branch always, branch on zero, or branch on negative.

A Mic-1 cycle can also initiate either a read or a write operation as determined by the RD and WR bits of the MIR. Main memory is accessed by writing the address in the MAR. For a read, the RD control line is activated for two Mic-1 cycles. The value read is then available in the MBR and can be moved into one of the sixteen Mic-1 registers. For a write, the value to be written is moved to the MBR, and the WR line is activated for two Mic-1 cycles. This stores the value in memory.

Overview of the Mac-1

Tanenbaum gives a control store for the Mic-1 (just 79 words) which implements a simple assembly language instruction set called the Mac-1. The 23 instructions of the Mac-1 include load, store, add, and subtract instructions for a simple accumulator machine using direct and stack addressing. Other instructions include push, pop, call, return, and jump. Four of the Mic-1 registers represent the program counter (PC), accumulator (AC), the stack pointer (SP), and the instruction register (IR) of the Mac-1 machine. Two other registers are used for scratch registers. Five registers hold constants and the remaining five of the sixteen Mic-1 registers are not used by the control store which implements the Mac-1.

The basic instruction cycle for a Von Neumann machine such as the Mac-1 consists of instruction fetch, increment program counter, decode instruction, and execute instruction. The Mac-1 instruction cycle is started with the execution of microinstruction zero. The instruction whose address is contained in the PC is read into the instruction register, the PC is incremented, and the instruction is decoded. The decoding takes a long time on the Mic-1 because it is done by examining the bits of the instruction one at a time. The decoding is not described here.

As an example of instruction execution, consider the Mac-1 POP instruction which removes the memory value at the top of the stack and puts it into the accumulator (AC). POP can be described functionally as: `ac := m[sp]; sp = sp+1;`

The Mac-1 POP is implemented by three instructions in the Mic-1 control store:

```
mar := sp; sp := sp + 1; rd;
rd;
ac := mbr; goto 0;
```

The first of these instructions does two operations. It starts the memory read necessary to obtain the word on the top of the stack, and it increments the stack pointer. Since memory operations take two Mic-1 cycles, the next instruction just waits for the read to complete by keeping the RD line active. In the third instruction, the result read from memory is moved from the MBR into the AC. The instruction then transfers control (`goto 0`) to the part of the microprogram which starts the next Mac-1 instruction cycle.

Let us look in detail at how the first of these instructions is executed in a single cycle of the Mic-1 machine. Since the Mic-1 is also a Von Neumann machine, the instruction cycle of the Mic-1 also consists of the operations of (micro)instruction fetch, increment (micro)program counter, decode (micro)instruction, and execute. Here is how this is done on the four subcycles of the Mic-1.

On the first subcycle, the instruction pointed to by the MPC is moved from the control store into the MIR. Since each collection of bits in the control store has an independent control function, the instruction is already decoded. A control store with this property is called horizontal. The next three subcycles perform the execution and the incrementing of the MPC. The microinstruction: `mar := sp; sp := sp + 1; rd;` is given below:

AMUX	COND	ALU	SH	MBR	MAR	RD	WR	ENC	C	B	A	ADDR
1	00	00	00	0	1	1	0	1	0010	0010	0110	xxxxxxxx

On subcycle 2, the register in the A field of the MIR is moved into the A latch and the register in the B field is moved into the B latch. Register 6=$(0110)_2$ has the constant one, and register 2=$(0010)_2$ is the SP. The AMUX field of 1 selects the A latch as the left input of the ALU. The ALU field of 0 specifies that the ALU should add its two inputs, and the SH field of 0 indicates that the Shifter should not shift. After some delay determined by the speed of this combinational logic, the output of the Shifter will be SP+1.

On subcycle 3, since the MAR field of the MIR is 1, the contents of the B latch (the SP register) is stored in the MAR. Since the RD field is one, a read cycle is started.

On subcycle 4, since the ENC (enable C bus) field of the MIR is one, the Shifter output is stored in the register in the C field of the MIR. This is register 2, which is the SP register. Thus, the SP register has been incremented. Because the MBR field is 0, the Shifter output is not stored in the MBR. The COND field of 0 indicates that no jump is to be performed and so the left input of the Mmux is used to load the new value of the MPC. This increments the MPC. The ADDR field is not used by this microinstruction since no branch occurred.

References

[1] M. Cutler and R. R. Eckert, "Microprogrammed Computer Simulator Tools," *IEEE Trans. Educ.*, vol. 33, pp 212—220, May 1990.

[2] R. J. Distler, "A Simulator for a Bit-Slice Computer," *IEEE Trans. Educ.*, vol. 33, pp 363—365, Nov. 1990.

[3] W. K. Fuchs, W. Page, J. H. Patel, P. Tobin, "Workstation-Based Logic Animation and Microarchitecture Emulation for Teaching Introduction to Computer Engineering," *IEEE Trans. Educ.*, vol. 32, pp 218—224, August 1989.

[4] J. O. Hamblen, A. Parker, and G. A. Rohling, "An Instructional Laboratory to Support Microprogramming,", *IEEE Trans. Educ.*, vol. 33, pp 333—336, Nov. 1990.

[5] R. P. Paul *Sparc Architecture, Assembly Language Programming, & C*, Prentice Hall, 1994.

[6] G. Puvvada and M. A. Breuer, "Teaching Computer Hardware Design Using Commercial CAD Tools," *IEEE Trans. Educ.*, vol. 36, pp 258—163, Feb. 1993.

[7] J. T. Stasko, "Animating algorithms with XTANGO," SIGACT News, vol. 23, number 2, pp 67-71, 1992

[8] A. S. Tanenbaum, *Structured Computer Organization*, Third Edition, Prentice Hall, 1990.

[9] Anonymous ftp from `ringer.cs.utsa.edu` in `pub/simulators/mic1`.