A Microprogrammable Simulator for Logic Design

Dick K. Blandford
University of Evansville

Abstract

A class in logic design is a mainstay of most electrical and computer engineering programs and is typically taught in the sophomore year. Such courses cover Boolean algebra, combinational and sequential logic design, logic circuit analysis, and programmable logic devices. Many also provide an introduction to register level design and computer architecture. This paper presents the details of a simulation program that allows students to write microinstructions and some machine code for a simple machine. The program presents an easy-to-use graphical user interface and has options for a two-level and three-level pipelined machine. The program is currently in use in a sophomore level course in logic design as a mechanism to illustrate how a computer works and to introduce computer architecture concepts.

I. Introduction

Microprogramming is a concept dating back to the early 1950's[1] and is widely used in the design of modern computers. The concept is easy to understand and presents a general solution to the control problem for a central processing unit. Students who understand how microprogramming works from a logical point of view have insight into the inner workings and magic of digital computer systems that is difficult to obtain otherwise. An animated simulation of a microprogrammed computer in which a user can "see" the bits interacting with the hardware provides a vehicle for teaching what microprogramming is all about. The software described in this paper runs under the Windows XP operating system and is used in several projects in a sophomore-level electrical engineering class on logic design. It allows students to microprogram a simulated 8-bit computer and to visualize such architectural features as pipelining, the stored program, and the arithmetic and logic unit (alu) loop.

II. Characteristics of the simulated machine

The simulated machine is loosely based on the Intel 8080 8-bit cpu that dates back to the mid seventies. This machine has roughly the same register set, accumulator-based architecture, and ALU functions. It is however, microprogrammed (the 8080 was not) and it can be used in a pipelined mode (the 8080 had no pipeline). The architectural features of the machine are as follows:

- 16-bit address bus
- 8-bit data bus

- 512 word x 31-bit microprogram control store
- 8-bit user modifiable input port
- Two 7-segment displays driven by an 8-bit BCD output port
- 64K byte program and data memory for user programs
- 8-bit natural instruction word width (multiple word instructions possible)
- user can single step or run microcode
- Registers
  - 16-bit program counter (PC)
  - 16-bit stack pointer (SP)
  - 16-bit memory address register (MAR)
  - 8-bit accumulator (A)
  - Six 8-bit registers for general use (B, C, D, E, H, and L)
  - One 8-bit temporary register for microprogram use only (T)
  - 4-bit flag registers (zero, carry, sign, and parity)
  - 9-bit microcontrol address register
- 16-function, 8-bit ALU
- fully animated data, control, and address paths
- built-in editor so the user can modify, save, and print
  - Microcode file
  - Instruction decoder file
  - User program file
- Can operate in pipelined or non-pipeline mode
- Pipeline can be two levels (fetch/execute) or three levels (fetch/decode/execute).

Figure 1 shows the opening screen of the simulator program. The three-level pipeline mode is shown.

III. Operation of the simulator

In operation a user creates an application and loads it into memory. This machine resets all registers to zero so zero is a convenient address of the first machine instruction. The application program appears in program memory as a sequence of bytes with a machine instruction consisting of one or more bytes. On start up, the microprogram counter is reset to zero and the microprogram sequence stored at that location is typically the machine instruction fetch. This fetch delivers the machine instruction to the instruction register (IR) where it is decoded by the instruction decoder. This decoder does a simple translation of the 8-bit machine instruction into a 9-bit starting address for the microcode. Thus, after the initial fetch sequence, the next microinstruction to be executed is determined by the value of the decoded machine instruction. Microinstructions thereafter are executed in order until some final microinstruction in the sequence triggers a reset of the microinstruction program counter setting it back to zero. Since the instruction fetch sequence is stored at location zero in the microprogram memory, this is the fetch for the next machine instruction.

To put the simulator into operation, the user must create three files: 1) the microprogram file which contains the microinstructions, 2) the program file which contains the machine instructions, and 3) the instruction decoder file which translates machine instructions into

starting addresses for the microcode. Each of these files can be created with an internal editor and each is stored as a text file.

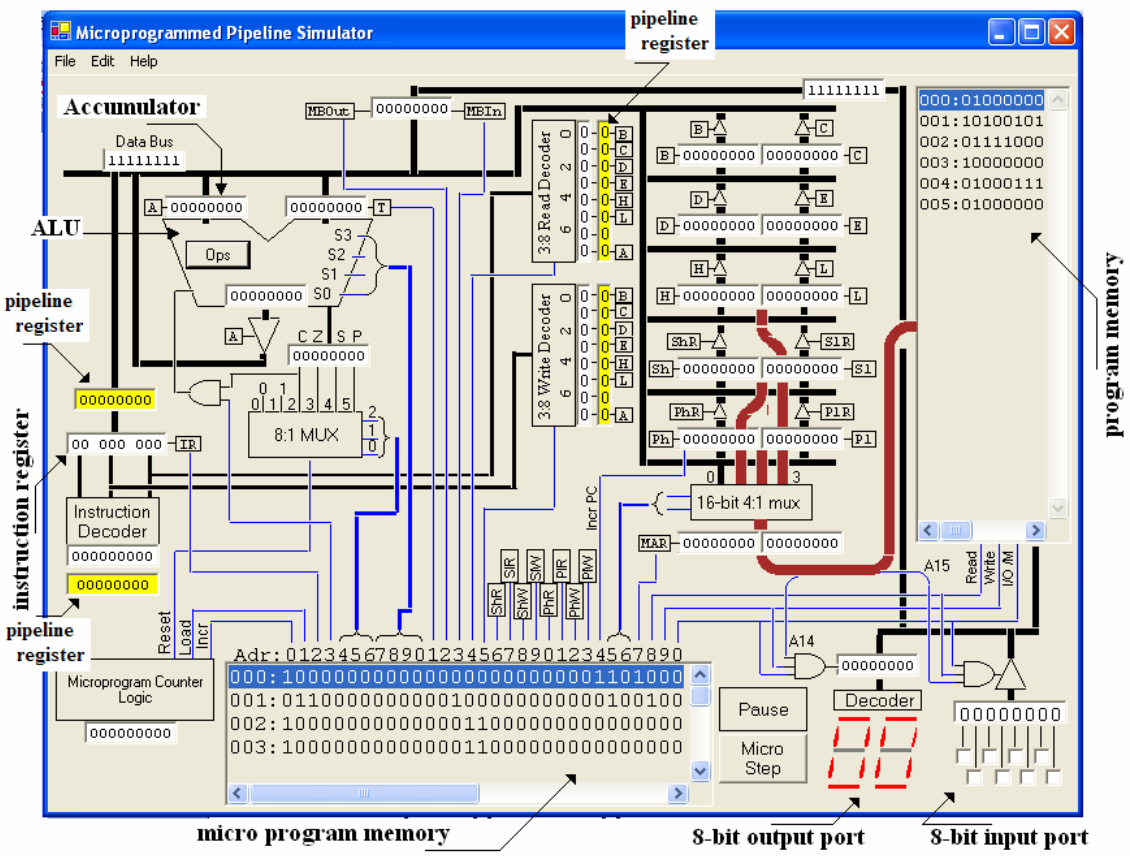

Figure 1

The opening screen of the simulator. This screen shows the main architectural features of the machine. The yellow registers are used for pipelining and disappear when not in the pipeline mode.

The microprogram file: A single microinstruction consists of 31 bits where each bit controls some aspect of the instruction fetch/execution cycle. Since the microinstruction program counter is reset to zero the microinstruction at location zero begins a two microinstruction sequence which does a fetch of a machine instruction from program memory. The first two microinstructions do the following operations:

```
Microinstruction 1: Load 16 bits from program counter to MAR,
                    Increment the program counter
Microinstruction 2: Does a memory read and copies data to the IR
                    Loads the microprogram counter with the decoded
                        address
```

Once a machine instruction is copied into the IR, it is decoded. The decoder translates the 8-bit machine instruction into a 9-bit address for the microprogram memory. The second microinstruction loads this 9-bit address into the microprogram counter. Thus, after the first two microinstructions at locations 0 and 1, the next microinstruction is determined by the decoded machine instruction address.

For example, suppose, in a very simple machine we had just three machine instructions which are add (`add`), move (`mov`), and multiply (`mpy`). We would have to figure out the microinstruction sequence to make each of these operations possible. Let's say that the `add` and `mov` instructions require two microinstructions each and the `mpy` instruction requires 10 microinstructions. The microprogram memory might look something like that shown in Figure 2.

Microprogram memory	
Addr	**31-bit microinstruction**
0	first word of fetch
1	second word of fetch
2	first word of `add` instruction
3	second word of `add` instruction
4	first word of `mov` instruction
5	second word of `mov` instruction
6	first word of `mpy` instruction
…	…
15	tenth word of `mpy` instruction

Figure 2

One possible configuration for microprogram memory for a machine with just three instructions. `Add` begins at location 2, `mov` at 4, and `mpy` at 6. The last microinstruction in each sequence resets the microprogram counter to zero to begin the fetch anew.

To write the microprogram, the user must set the 31 ones and zeros in each microinstruction word to their proper values for the microinstruction function. Figure 3 below shows the assigned function for each of the 31 bit positions in the microinstruction. All bits are active high.

Note that the register timing is assumed to be such that the user can enable one register to get onto the bus and clock another in the same microinstruction thus moving data from one register to another in a single micro cycle.

The machine code file: This is the application program which the user writes to carry out the desired set of operations. There is no assembler or high level language so the application programs are short and written in machine language. The instruction register (IR) is just 8-bits wide and it is arranged so that the least significant 6-bits are directed to the read and write register decoders. This is convenient for instructions which operate on two registers using a single machine code byte. From looking at the read and write register decoders we can deduce the following bit codes for the registers:

```
000 → B
001 → C
010 → D
011 → E
100 → H
101 → L
110 → not used
111 → A
```

bit	function
0	increments the microprogram counter
1	loads the microprogram counter from the instruction decoder
2	clocks the data bus to the instruction register
3	forwards the current carry flag to the ALU
4	3:8 mux input which chooses 0, 1, carry, zero, sign, or parity flag to go the microprogram counter reset.
5	
6	
7	chooses one of 16 ALU functions. See table xxx
8	
9	
10	
11	clocks the data bus to the T register
12	enables the memory buffer register output the external data bus
13	enables the memory buffer register output to the internal data bus
14	enables the 3:8 read decoder which decodes the least significant 3 bits of the instruction register. The decoder output clocks the decoder register.
15	enables the 3:8 write decoder which decodes bits 3, 4, and 5 of the instruction register. The decoded output enables the decoded register output to the data bus.
16	enables the stack pointer high byte register to the data bus
17	enables the stack pointer low byte register to the data bus
18	clocks the stack pointer high byte register
19	clocks the stack pointer low byte register
20	enables the program counter high byte register to the data bus
21	enables the program counter low byte register to the data bus
22	clocks the program counter high byte register
23	clocks the program counter low byte register
24	increments the 16-bit program counter register
25	4:1 mux which selects whether the program counter, stack pointer, HL register pair, or nothing is sent to the memory address register
26	
27	clocks the 16-bit memory address register
28	enables program memory read
29	enables program memory write
30	$IO / \overline{M}$ this bit is 0 for I/O operation and 1 for memory operation

Figure 3
Bit assignments for the microinstruction word.

Note that the instruction register is arranged so that bits 2, 1, and 0 go to the read decoder and bits 5, 4, and 3 go to the write decoder. Bits 2, 1, and 0 thus become the source bits and bits 5, 4, and 3 become the destination bits. This leaves just 2 bits for an op code for instructions arranged in this manner. Following Intel's[2] lead for the 8080 we choose 01 for the op code for a `mov` instruction. Thus the 8-bit register to register `mov` instruction would be put together in the following manner:

```
01 DDD SSS
```

In this format, DDD is a 3-bit code for the destination register and SSS is a 3-bit code for a source register. For example, 01 111 001 would be translated as move the accumulator register to the C register or `mov A, C`. Bit code 110 (6) is not assigned to a register. Intel used this bit code to designate a memory location whose address was in the HL register pair and this feature can be duplicated on this machine.

The Intel 8080 had an accumulator architecture meaning that instructions doing arithmetic or logical operations always used the accumulator as one of the operands. This meant that the A register need not be specified as part of the instruction since it is implied. It makes good sense then to write all of the logical and arithmetic instructions with up to a 5-bit op code and a 3-bit source register. For example the instruction `ADD D` might be encoded as 10000 010 where the first five bits are an op code for the `add` instruction and the last three bits are the register code for the D register. The instruction would perform the following operations A ← A + D.

While the examples for the `mov` and `add` instructions shown here make good sense, there are many other sensible choices including instructions which have multiple bytes.

After the user has created an instruction set and written an application program the program can be arranged in a simple text file to be loaded into the simulator.

The Instruction Decoder File: This is the third required text file which the user must create. It is used by the simulator to translate op codes from machine instructions into starting addresses for the microcode sequence. In our example in Figure 2 we had an `add`, `mov`, and `mpy` instruction. If the respective 8-bit op codes for these instructions are 10000SSS, 01DDDSSS, and 00100SSS then the decoder text file might look something like that shown in Figure 4.

```
//Initial Text
//       012345678
00000000 000000000  //nop
11111111 000000000  //undefined memory
01xxxxxx 000000100  //mov dest, src
10000xxx 000000010  //add src
00100xxx 000000110  //mpy src
```

Figure 4

A portion of a sample decoder file. The double slash indicates a comment and is ignored by the simulator. The 8-bit number on the left is the op code and the 9-bit number on the right is the corresponding microprogram memory address. The sequence xxx indicates a "don't care".

IV. Running a Simulation

With the three text files in place it is possible to run a simulation. The simulator runs in either a single step mode or a run mode. In the run mode, the user can set the run speed and determine how fast instructions are executed. The single step mode allows for careful checking and debugging of the text files.

For this simulated machine, the user can choose whether or not to make use of pipelining (using the Edit → Preferences menu). Figure 5 shows the architecture for all three modes of operation. In Figure 5A there is no pipelining; in Figure 5B there is an overlap of fetch and execute cycles and an additional pipeline register has been added; in Figure 5C there is an overlap of fetch, execute, and instruction decode and four pipeline registers have been added.

Each microinstruction can be thought of as existing in one of three states (or cycles): read cycle, write cycle, and pause cycle. In the read cycle all microinstruction bits which are set to one for the current microinstruction enable register outputs effectively connecting them to the bus.

Figure 5A
This is the non pipelined mode. Fetch and execute are separate non-overlapped operations. The first two microinstructions can be configured to do an instruction fetch.

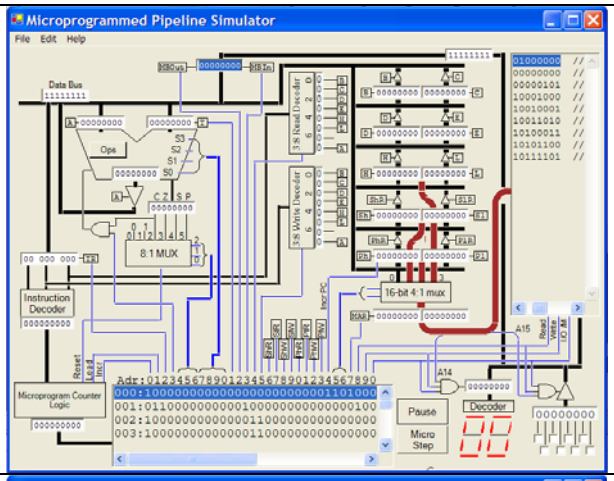

Figure 5B
This is the two-stage pipeline mode. In this mode fetch and execute are overlapped. A pipeline register has been added just before the instruction register at mid left to make this overlap possible. The first two microinstructions do an initial instruction fetch but subsequent microinstructions may have a fetch built into their sequence.

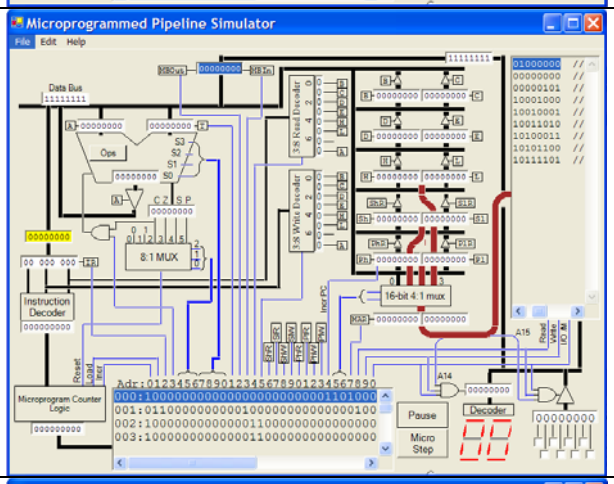

Figure 5C
This is the three-stage pipeline mode. Fetch, execute, and instruction decode may be overlapped. In this mode there are four pipelined registers. Two of these registers follow the register decode units and one pipeline register has been added immediately after the instruction decoder.

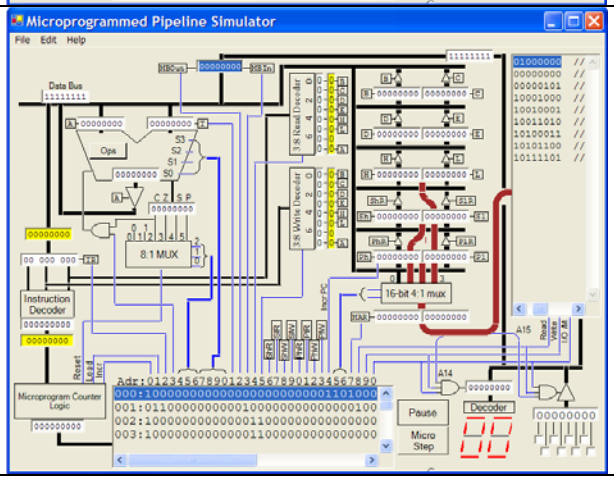

Figure 5
This figure shows the simulated architecture for all three modes of operation. The operation mode is chosen from the Edit → Preferences menu.

Typically only one register can be connected to the bus at any one time and the register that is enabled has its enable line highlighted in red. During the write cycle any clock line that becomes a 1 for the current microinstruction is activated and its line is highlighted in red. Note that the enable and the clock overlap so that one register may be enabled onto the bus while another is clocked to take the data from the bus, thus completing a register-to-register move. Each micro

cycle completes with a pause cycle in which enables and clocks become inactive and the results can be viewed.

To edit the machine code file, the decoder file, or the microinstruction file, the user double clicks on the program memory, the instruction decoder, or the microprogram memory. (Alternatively, the user can select the editor from the edit menu.) The editor enables the user to alter a file and reload it. The program can then be reset or continued from that point on.

V. Pipelining

The simulator can be set to allow either two stage or three stage pipelining (as well as the standard non-pipelined starting mode). A two-stage pipe overlaps the instruction fetch with the instruction execute. In a three-stage pipe, one instruction can be fetched while a second is being decoded, and a third is being executed.

If the machine is not pipelined, typically the first two microinstructions do an instruction fetch. Every instruction then executes a short sequence of microinstructions followed by a reset of the microprogram counter to get to this fetch sequence. If the machine is pipelined then the fetch and execute can be overlapped. In this case, it is typical to make the first two microinstructions the instruction fetch for the initial instruction. After the initial instruction each instruction may contain a fetch sequence for the next instruction as part of its normal execution. For example, a move instruction which moves data from one register to another, can consist of just two microinstructions. The first moves the data between registers and simultaneously puts the program counter onto the address bus and does a memory read. The second microinstruction then moves the next instruction to the CPU.

VI. Use in the Classroom

This simulator is used during the last month of a first course in logic design which is typically taken during the spring of the sophomore year. The course is a mix of both electrical and computer engineering majors. Students download the program from a web site along with a user's manual and they work in teams of two. Initially students work on the non-pipelined machine and are given microinstruction sequences for particular instructions. They are asked to improve or debug these sequences. This evolves into creating a full assembly language instruction set with each team developing different instructions. Pipelining is added in, beginning with the two-stage pipe. The delayed branch is introduced and students write and simulate code for this instruction. Conditional branches are explored in detail and students compute the loss in efficiency that occurs when the pipeline must be cleared due to a branch instruction. Finally, students are asked to improve the machine by making suggestions for added hardware or additional hardware functions.

This program can be downloaded as an executable file from the author's web site at `http://csserver.evansville.edu/~blandfor`. The source code is available for noncommercial purposes on request.

Bibliography

[1] MV Wilkes. The Best Way to Design an Automated Calculating Machine. Manchester Univ. Computer Inaugural Conf, 1951, pp. 16-18.
[2] MCS-85 User's Manual, Intel Corporation, June, 1977

DICK K. BLANDFORD
Dr. Dick K. Blandford is the Chair of the Electrical Engineering and Computer Science Department at the University of Evansville.