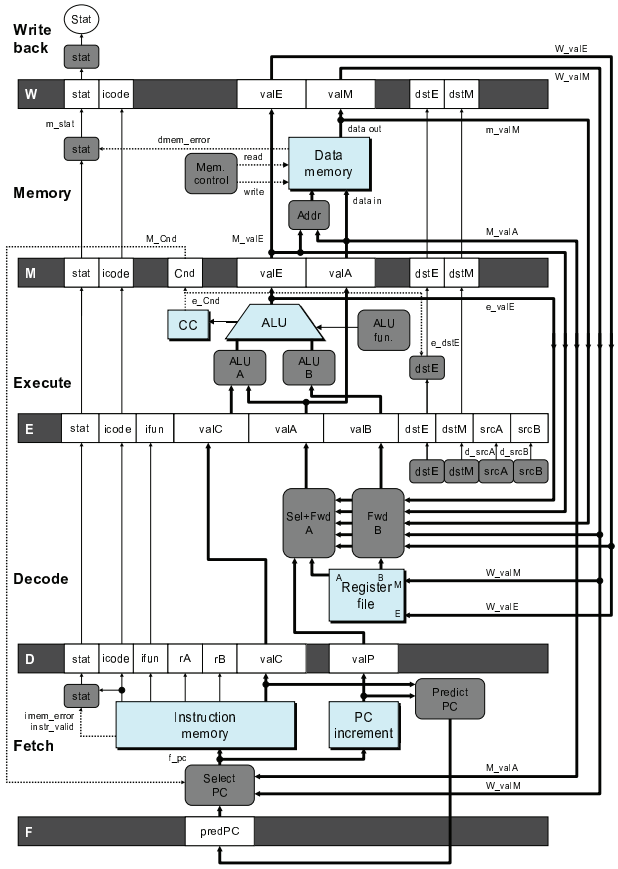
CS:APP2e Guide to Y86 Processor Simulators*



Randal E. Bryant
David R. O'Hallaron

May 2, 2011

This document describes the processor simulators that accompany the presentation of the Y86 processor architectures in Chapter 4 of *Computer Systems: A Programmer's Perspective, Second Edition*. These simulators model three different processor designs: SEQ, SEQ+, and PIPE.

1 Installing

The code for the simulator is distributed as a tar format file named `sim.tar`. You can get a copy of this file from the CS:APP2e Web site (`csapp.cs.cmu.edu`).

With the tar file in the directory you want to install the code, you should be able to do the following:

```
unix> tar xf sim.tar
unix> cd sim
unix> make clean
unix> make
```

By default, this generates GUI (graphic user interface) versions of the simulators, which require that you have Tcl/Tk installed on your system. If not, then you have the option to install TTY-only versions that emit their output as ASCII text on stdout. See file `README` for a description of how to generate the GUI and TTY versions.

The directory `sim` contains the following subdirectories:

`misc` Source code files for utilities such as YAS (the Y86 assembler), YIS (the Y86 instruction set simulator), and HCL2C (HCL to C translator). It also contains the `isa.c` source file that is used by all of the processor simulators.

`seq` Source code for the SEQ and SEQ+ simulators. Contains the HCL file for homework problems 4.49 and 4.50. See file `README` for instructions on compiling the different versions of the simulator.

`pipe` Source code for the PIPE simulator. Contains the HCL files for homework problems 4.52–4.57. See file `README` for instructions on compiling the different versions of the simulator.

`y86-code` Y86 assembly code for many of the example programs shown in the chapter. You can automatically test your modified simulators on these benchmark programs. See file `README` for instructions on how to run these tests. As a running example, we will use the program `asum.ys` in this subdirectory. This program is shown as CS:APP2e Figure 4.8. The compiled version of the program is shown in Figure 1.

`ptest` Scripts that generate systematic regression tests of the different instructions, the different jump possibilities, and different hazard possibilities. These scripts are very good at finding bugs in your homework solutions. See file `README` for instructions on how to run these tests.

```
1                       | # Execution begins at address 0
2   0x000:              |         .pos 0
3   0x000: 30f400010000 | init:   irmovl Stack, %esp      # Set up stack pointer
4   0x006: 30f500010000 |         irmovl Stack, %ebp      # Set up base pointer
5   0x00c: 8024000000   |         call Main               # Execute main program
6   0x011: 00           |         halt                    # Terminate program
7                       |
8                       | # Array of 4 elements
9   0x014:              |         .align 4
10  0x014: 0d000000     | array:  .long 0xd
11  0x018: c0000000     |         .long 0xc0
12  0x01c: 000b0000     |         .long 0xb00
13  0x020: 00a00000     |         .long 0xa000
14                      |
15  0x024: a05f         | Main:   pushl %ebp
16  0x026: 2045         |         rrmovl %esp,%ebp
17  0x028: 30f004000000 |         irmovl $4,%eax
18  0x02e: a00f         |         pushl %eax              # Push 4
19  0x030: 30f214000000 |         irmovl array,%edx
20  0x036: a02f         |         pushl %edx              # Push array
21  0x038: 8042000000   |         call Sum                # Sum(array, 4)
22  0x03d: 2054         |         rrmovl %ebp,%esp
23  0x03f: b05f         |         popl %ebp
24  0x041: 90           |         ret
25                      |
26                      |         # int Sum(int *Start, int Count)
27  0x042: a05f         | Sum:    pushl %ebp
28  0x044: 2045         |         rrmovl %esp,%ebp
29  0x046: 501508000000 |         mrmovl 8(%ebp),%ecx     # ecx = Start
30  0x04c: 50250c000000 |         mrmovl 12(%ebp),%edx    # edx = Count
31  0x052: 6300         |         xorl %eax,%eax          # sum = 0
32  0x054: 6222         |         andl   %edx,%edx        # Set condition codes
33  0x056: 7378000000   |         je     End
34  0x05b: 506100000000 | Loop:   mrmovl (%ecx),%esi      # get *Start
35  0x061: 6060         |         addl %esi,%eax          # add to sum
36  0x063: 30f304000000 |         irmovl $4,%ebx          #
37  0x069: 6031         |         addl %ebx,%ecx          # Start++
38  0x06b: 30f3ffffffff |         irmovl $-1,%ebx         #
39  0x071: 6032         |         addl %ebx,%edx          # Count--
40  0x073: 745b000000   |         jne    Loop             # Stop when 0
41  0x078: 2054         | End:    rrmovl %ebp,%esp
42  0x07a: b05f         |         popl %ebp
43  0x07c: 90           |         ret
44                      |
45                      | # The stack starts here and grows to lower addresses
46  0x100:              |         .pos 0x100
47  0x100:              | Stack:
```

Figure 1: **Sample object code file.** This code is in the file `asum.yo` in the `y86-code` subdirectory.

2 Utility Programs

Once installation is complete, the `misc` directory contains two useful programs:

YAS The Y86 assembler. This takes a Y86 assembly code file with extension `.ys` and generates a file with extension `.yo`. The generated file contains an ASCII version of the object code, such as that shown in Figure 1 (the same program as shown in CS:APP2e Figure 4.8). The easiest way to invoke the assembler is to use or create assembly code files in the `y86-code` subdirectory. For example, to assemble the program in file `asum.ys` in this directory, we use the command:

```
unix> make asum.yo
```

YIS The Y86 instruction simulator. This program executes the instructions in a Y86 machine-level program according to the instruction set definition. For example, suppose you want to run the program `asum.yo` from within the subdirectory `y86-code`. Simply run:

```
unix> ../misc/yis asum.yo
```

YIS simulates the execution of the program and then prints changes to any registers or memory locations on the terminal, as described in CS:APP2e Section 4.1.

3 Processor Simulators

For each of the three processors, SEQ, SEQ+, and PIPE, we have provided simulators SSIM, SSIM+, and PSIM respectively. Each simulator can be run in TTY or GUI mode:

TTY mode Uses a minimalist, terminal-oriented interface. Prints everything on the terminal output. Not very convenient for debugging but can be installed on any system and can be used for automated testing. The default mode for all simulators.

GUI mode Has a graphic user interface, to be described shortly. Very helpful for visualizing the processor activity and for debugging modified versions of the design. However, it requires installation of Tcl/Tk on your system. Invoked with the `-g` command line option.

3.1 Command Line Options

You can request a number of options from the command line:

`-h` Prints a summary of all of the command line options.

`-g` Run the simulator in GUI mode (default TTY mode).

-t Runs both the processor and the ISA simulators, comparing the resulting values of the memory, register file, and condition codes. If no discrepancies are found, it prints the message "ISA Check Succeeds." Otherwise, it prints information about the words of the register file or memory that differ. This feature is very useful for testing the processor designs.

-l m Sets the instruction limit, executing at most m instructions before halting (default 10000 instructions).

-v n Sets the verbosity level to n, which must be between 0 and 2 with a default value of 2.

Simulators running in GUI mode must be invoked with the name of an object file on the command line. In TTY mode, the object file name is optional, coming from `stdin` by default.

Here are some typical invocations of the simulators (from the `y86-code` subdirectory):

```
unix> ../seq/ssim -h
unix> ../seq/ssim -t < asum.yo
unix> ../pipe/psim -t -g asum.yo
```

The first case prints a summary of the command line options for SSIM. The second case runs the SEQ simulator in TTY mode, reading object file `asum.yo` from `stdin`. The third case runs the PIPE simulator in GUI mode, executing the instructions object file `asum.yo`. In both the second and third cases, the results are compared with the results from the higher-level ISA simulator.

3.2 SEQ and SEQ+ Simulators

The GUI version of the SEQ processor simulator is invoked with an object code filename on the command line:

```
unix> ../seq/ssim -g asum.yo &
```

where the "`&`" at the end of the command line allows the simulator to run in background mode. The simulation program starts up and creates three windows, as illustrated in Figures 2–4.

The first window (Figure 2) is the main control panel. If the HCL file was compiled by HCL2C with the `-n name` option, then the title of the main control window will appear as "`Y86 Processor: name`" Otherwise it will appear as simply "`Y86 Processor`."

The main control window contains buttons to control the simulator as well as status information about the state of the processor. The different parts of the window are labeled in the figure:

Control: The buttons along the top control the simulator. Clicking the Quit button causes the simulator to exit. Clicking the Go button causes the simulator to start running. Clicking the Stop button causes the simulator to stop temporarily. Clicking the Step button causes the simulator to execute one instruction and then stop. Clicking the Reset button causes the simulator to return to its initial state, with the program counter at address 0, the registers set to 0s, the memory erased except for the program, the condition codes set with ZF $= 1$, CF $= 0$, and OF $= 0$, and the program status set to . AOK.

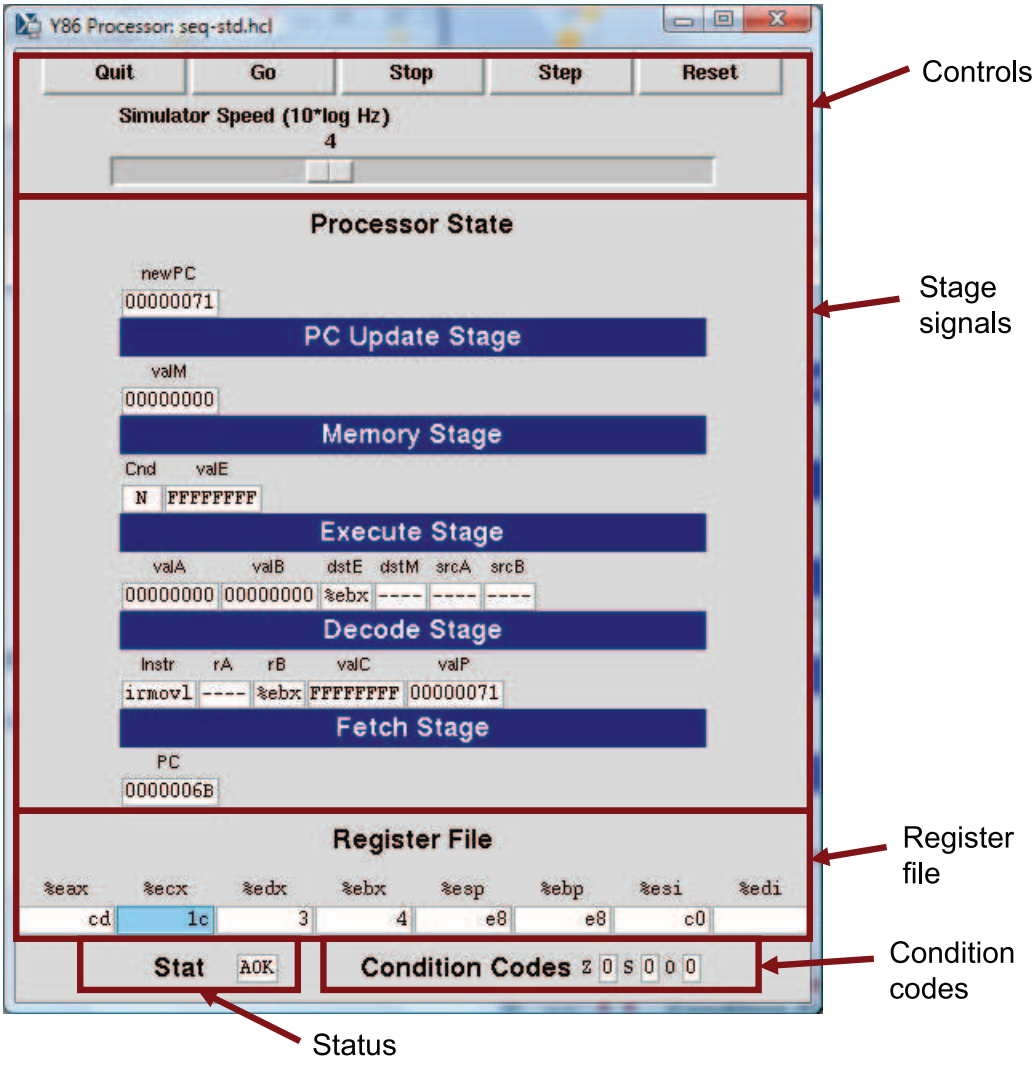


Figure 2: Main control panel for SEQ simulator

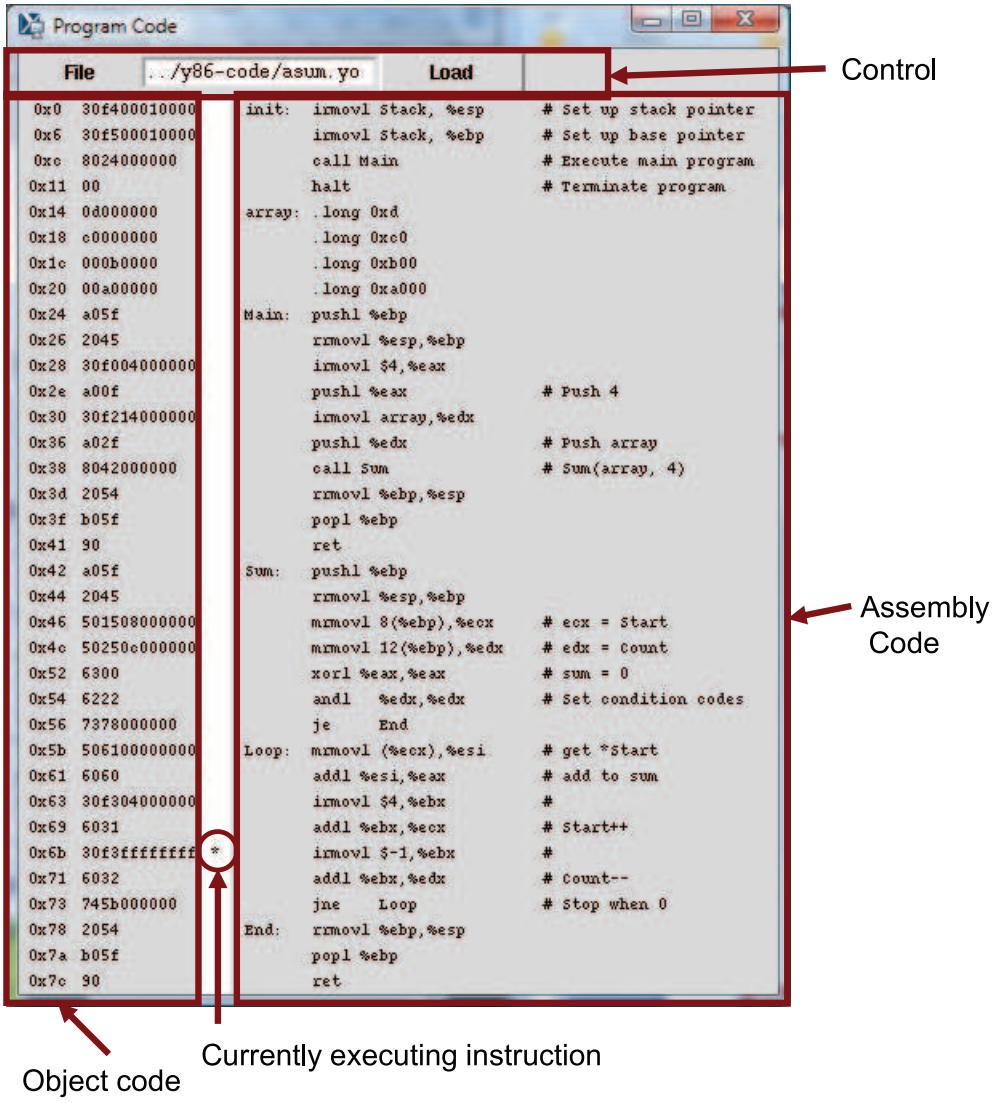


Figure 3: Code display window for SEQ simulator

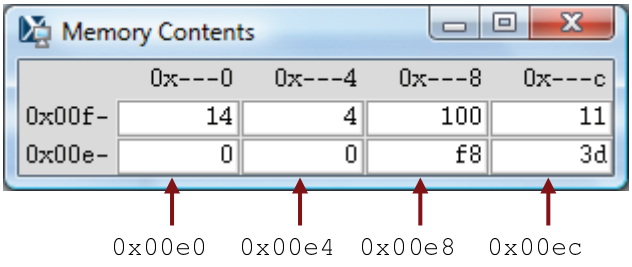


Figure 4: Memory display window for SEQ simulator

The slider below the buttons controls the speed of the simulator when it is running. Moving it to the right makes the simulator run faster.

Stage signals: This part of the display shows the values of the different processor signals during the current instruction evaluation. These signals are almost identical to those shown in CS:APP2e Figure 4.23. The main difference is that the simulator displays the name of the instruction in a field labeled Instr, rather than the numeric values of icode and ifun. Similarly, all register identifiers are shown using their names, rather than their numeric values, with "`----`" indicating that no register access is required.

Register file: This section displays the values of the eight program registers. The register that has been updated most recently is shown highlighted in light blue. Register contents are not displayed until after the first time they are set to nonzero values.

Remember that when an instruction writes to a program register, the register file is not updated until the beginning of the next clock cycle. This means that you must step the simulator one more time to see the update take place.

Stat: This shows the status of the current instruction being executed. The possible values are:

`AOK`: No problem encountered.

`ADR`: An addressing error has occurred either trying to read an instruction or trying to read or write data. Addresses cannot exceed `0x0FFF`.

`INS`: An illegal instruction was encountered.

`HLT`: A `halt` instruction was encountered.

Condition codes: These show the values of the three condition codes: `ZF`, `SF`, and `OF`.

Remember that when an instruction changes the condition codes, the condition code register is not updated until the beginning of the next clock cycle. This means that you must step the simulator one more time to see the update take place.

The processor state illustrated in Figure 2 is for the second execution of line 38 of the `asum.yo` program shown in Figure 1. We can see that the program counter is at `0x06b`, that it has processed the instruction `addl %ebx %ecx`, that register `%eax` holds `0xcd`, the sum of the first two array elements, and `%edx` holds 3, the count that is about to be decremented. Register `%ecx` holds `0x1c`, the address of the third array element. Register `%ebx` still holds the value 4 (from line 36) but there is a pending write of `0xFFFFFFFF` to this register (since dstE is set to `%ebx` and valE is set to `0xFFFFFFFF`). This write will take place at the start of the next clock cycle.

The window depicted in Figure 3 shows the object code file that is being executed by the simulator.The edit box identifies the file name of the program being executed. You can edit the file name in this window and click the Load button to load a new program. The left hand side of the display shows the object code being executed, while the right hand side shows the text from the assembly code file. The center has an asterisk (`*`) to indicate which instruction is currently being simulated. This corresponds to line 38 of the `asum.yo` program shown in Figure 1.

The window shown in Figure 4 shows the contents of the memory. It shows only those locations between the minimum and maximum addresses that have changed since the program began executing. Each row shows the contents of four memory words. Thus, each row shows 16 bytes of the memory, where the addresses of the bytes differ in only their least significant hexadecimal digits. To the left of the memory values is the "root" address, where the least significant digit is shown as "`-`". Each column then corresponds to words with least significant address digits `0x0`, `0x4`, `0x8`, and `0xc`. The example shown in Figure 4 has arrows indicating memory locations `0x00e0`, `0x00e4`, `0x00e8`, and `0x00ec`.

The memory contents illustrated in the figure show the stack contents of the `asum.yo` program shown in Figure 1 during the execution of the `Sum` procedure. Looking at the stack operations that have taken place so far, we see that `%esp` and `%ebp` were initialized to `0x100` (lines 3 and 4). The call to `Main` on line 5 pushes the return pointer `0x011`, which is written to address `0x00fc`. Procedure `Main` starts by pushing `%ebp`, writing `0x100` to `0x00f8`. It then pushes the value of `%eax` (line 18), writing `0x4` to `0x00f4` and `%edx` (line 20), writing `0x14` (the address of the array) to `0x00f0`. The call to `Sum` on line 21 causes the return pointer `0x3d` to be written to address `0x00ec`. Within `Sum`, pushing `%ebp` causes `0xf8` to be written to address `0x00e8`. That accounts for all of the words shown in this memory display, and for the stack pointer being set to `0xe8`.

Figure 5 shows the control panel window for the SEQ+ simulator, when executing the same object code file and when at the same point in this program. We can see that the only difference is in the ordering of the stages and the different signals listed. These signals correspond to those in CS:APP2e Figure 4.40. The SEQ+ simulator also generates code and memory windows. These have identical format to those for the SEQ simulator.

3.3 PIPE Simulator

The PIPE simulator also generates three windows. Figure 6 shows the control panel. It has the same set of controls, and the same display of the register file and condition codes. The middle section shows the state of the pipeline registers. The different fields correspond to those in CS:APP2e Figure 4.52. At the bottom of this panel is a display showing the number of cycles that have been simulated (not including the initial cycles required to get the pipeline flowing), the number of instructions that have completed, and the resulting CPI.

As illustrated in the close-up view of Figure 7, each pipeline register is displayed with two parts. The upper values in white boxes show the current values in the pipeline register. The lower values with a gray background show the inputs to pipeline register. These will be loaded into the register on the next clock cycle, unless the register bubbles or stalls.

The flow of values through the PIPE simulator is quite different from that for the SEQ or SEQ+ simulator. With SEQ and SEQ+, the control panel shows the values resulting from executing a single instruction. Each step of the simulator performs one complete instruction execution. With PIPE, the control panel shows the values for the multiple instructions flowing through the pipeline. Each step of the simulator performs just one stage's worth of computation for each instruction.

Figure 8 shows the code display for the PIPE simulator. The format is similar to that for SEQ and SEQ+, except that rather than a single marker indicating which instruction is being executed, the display indicates which instructions are in each state of the pipeline, using characters `F`, `D`, `E`, `M`, and `W`, for the fetch, decode,

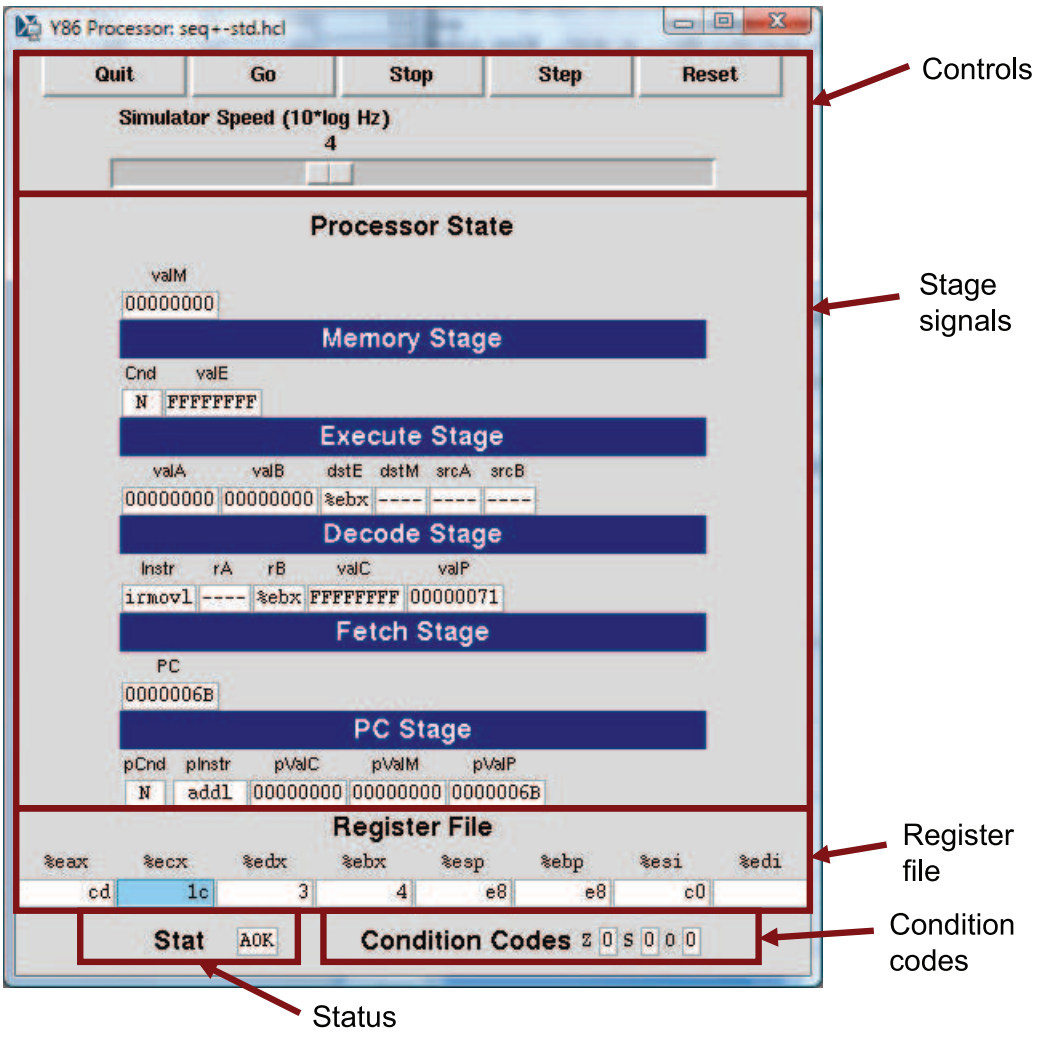


Figure 5: Main control panel for SEQ+ simulator

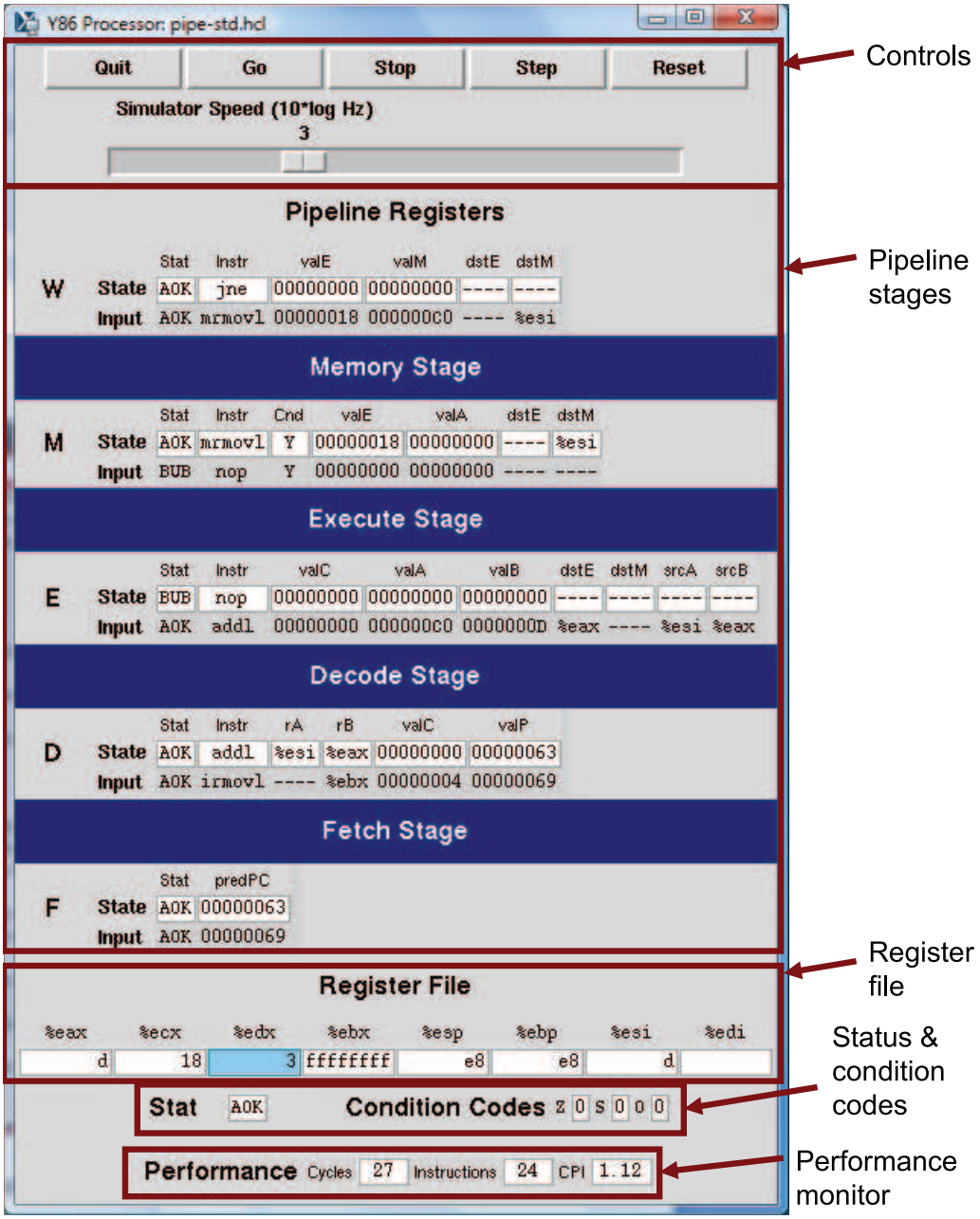


Figure 6: Main control panel for PIPE simulator

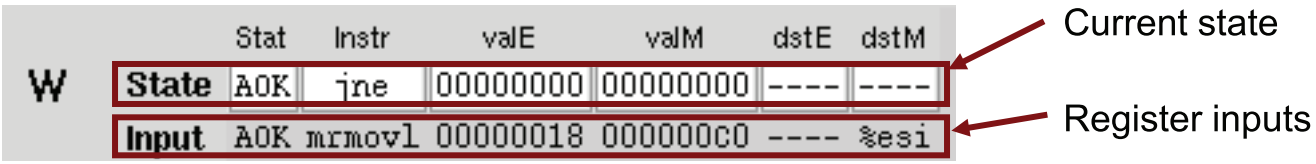


Figure 7: View of single pipe register in control panel for PIPE simulator

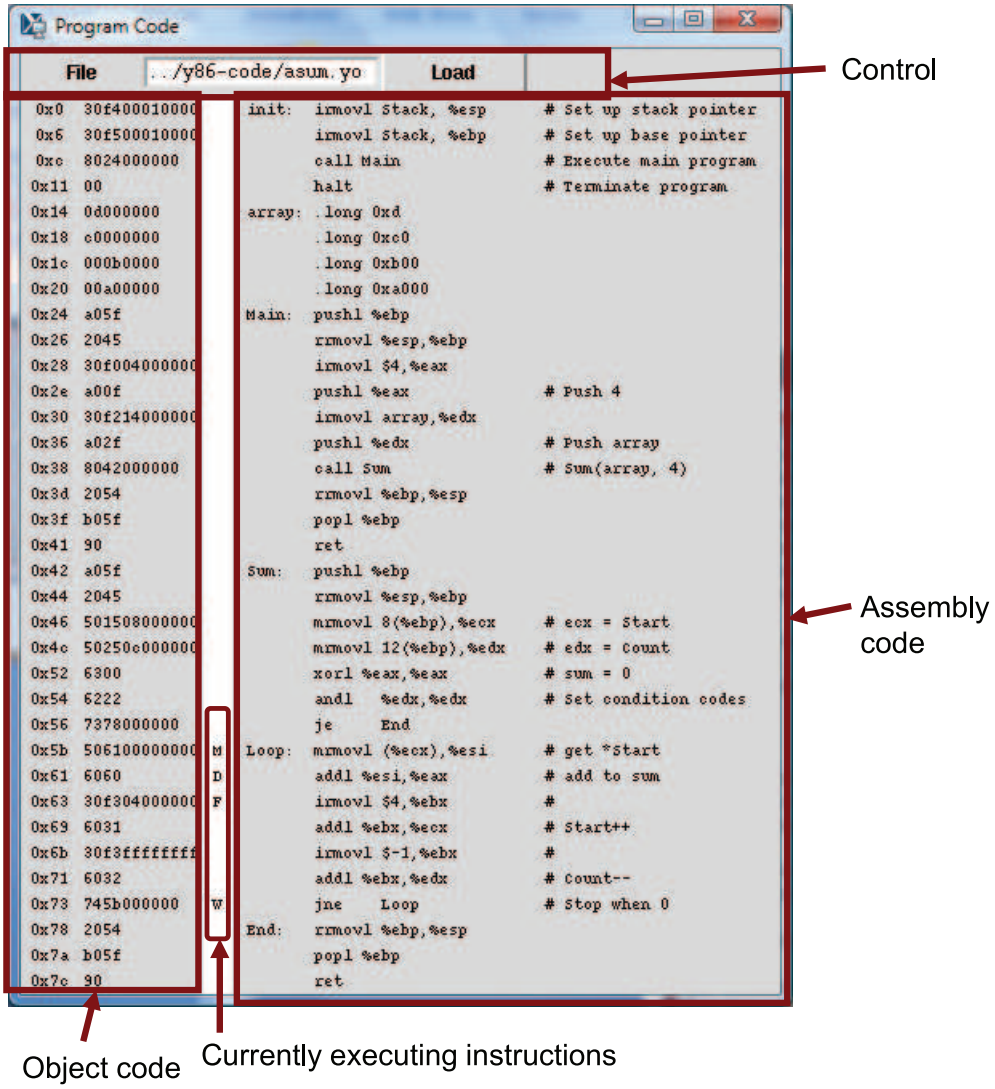


Figure 8: Code display window for PIPE simulator

execute, memory, and write-back stages.

The PIPE simulator also generates a window to display the memory contents. This has an identical format to the one shown for SEQ (Figure 4).

The example shown in Figures 6 and 8 show the status of the pipeline when executing the loop in lines 34–40 of Figure 1. We can see that the simulator has begun the second iteration of the loop. The status of the stages is as follows:

Write back: The loop-closing `jne` instruction (line 40) is finishing.

Memory: The `mrmovl` instruction (line 34) has just read `0x0C0` from address `0x018`. We can see the address in valE of pipeline register M, and the value read from memory at the input of valM to pipeline register W.

Execute: This stage contains a bubble. The bubble was inserted due to the load-use dependency between the `mrmovl` instruction (line 34) and the `addl` instruction (line 35). It can be seen that this bubble acts like a `nop` instruction. This explains why there is no instruction in Figure 8 labeled with "E."

Decode: The `addl` instruction (line 35) has just read `0x00D` from register `%eax`. It also read `0x00D` from register `%esi`, but we can see that the forwarding logic has instead used the value `0x0C0` that has just been read from memory (seen as the input to valM in pipeline register W) as the new value of valA (seen as the input to valA in pipeline register E).

Fetch: The `irmovl` instruction (line 38) has just been fetched from address `0x063`. The new value of the PC is predicted to be `0x069`.

Associated with each stage is its status field `Stat` This field shows the status of the instruction in that stage of the pipeline. Status `AOK` means that no exception has been encountered. Status value `BUB` indicates that a bubble is in this stage, rather than a normal instruction. Other possible status values are: `ADR` when an invalid memory location is referenced, `INS` when an illegal instruction code is encountered, `PIP` when a problem arose in the pipeline (this occurs when both the stall and the bubble signals for some pipeline register are set to 1), and `HLT` when a halt instruction is encountered. The simulator will stop when any of these last four cases reaches the write-back stage.

Carrying the status for an individual instruction through the pipeline along with the rest of the information about that instruction enables precise handling of the different exception conditions, as described in CS:APP2e Section 4.5.9.

4 Some Advice

The following are some miscellaneous tips, learned from experience we have gained in using these simulators.

- *Get familiar with the simulator operation.* Try running some of the example programs in the `y86-code` directory. Make sure you understand how each instruction gets processed for some small examples. Watch for interesting cases such as mispredicted branches, load interlocks, and procedure returns.

- *You need to hunt around for information.* Seeing the effect of data forwarding is especially tricky. There are seven possible sources for signal valA in pipeline register E, and six possible sources for signal valB. To see which one was selected, you need to compare the input to these pipeline register fields to the values of the possible sources. The possible sources are:

  R[**d_srcA**] The source register is identified by the input to srcA in pipeline register E. The register contents are shown at the bottom.

  R[**d_srcB**] The source register is identified by the input to srcB in pipeline register E. The register contents are shown at the bottom.

  D_valP This value is part of the state of pipeline register D.

  e_valE This value is at the input to field valE in pipeline register M.

  M_valE This value is part of the state of pipeline register M.

  m_valM This value is at the input to field valM in pipeline register W.

  W_valE This value is part of the state of pipeline register W.

  W_valM This value is part of the state of pipeline register M.

- *Do not overwrite your code.* Since the data and code share the same address space, it is easy to have a program overwrite some of the code, causing complete chaos when it attempts to execute the overwritten instructions. It is important to set up the stack to be far enough away from the code to avoid this.

- *Avoid large address values.* The simulators do not allow any addresses greater than `0x0FFF`. In addition, the memory display becomes unwieldy if you modify memory locations spanning a wide range of addresses.

- *Be aware of some "features" of the GUI-mode simulators (*SSIM*,* SSIM+*, and* PSIM*.)*

  – You must must execute the programs from their home directories. In other words, to run SSIM or SSIM+, you must be in the `seq` directory, while you must be in the `pipe` subdirectory to run PSIM. This requirement arises due to the way the Tcl interpreter locates the configuration file for the simulator.

  – If you are running in GUI mode on a Unix box, remember to initialize the DISPLAY environment variable:

  ```
  unix> setenv DISPLAY myhost.edu:0
  ```

  – With some Unix X Window managers, the "Program Code" window begins life as a closed icon. If you don't see this window when the simulator starts, you'll need to expand the expand manually by clicking on it.

  – With some Microsoft Windows X servers, the "Memory Contents" window does not automatically resize itself when the memory contents change. In these cases, you'll need to resize the window manually to see the memory contents.

  – The simulators will terminate with a segmentation fault if you ask them to execute a file that is not a valid Y86 object file.