CS 347, Spring 1998
Lab 3 Prologue:
Understanding Pipeline Hazards

Assigned: Tues., March. 31
Due: Tues., April 7

1. Policy

You may work in a group of up to 3 people in solving the problems for this assignment. You should turn in a single solution for your entire group, identifying all of the group members.

For this part of the lab there is no electronic handin. Instead, you will fill out your solution by hand using the provided answer sheets (by printing the postscript file `lab3-diagrams.ps`).

2. Logistics

Any clarifications and revisions to the assignment will be posted on the class bboard and Web page.

For this assignment, you will want to retrieve the code files `pipe1.O` and `pipe2.O` and the postscript file `lab3-diagrams.ps` from the directory

```
/afs/cs.cmu.edu/academic/class/15347-s98/public/labs/lab3
```

3. Introduction

In Lab 3 you will be modifying the code to the Alpha pipeline simulator to fix some problems regarding its handling of data and control hazards. As a warmup, we will have you use two versions of this simulator—one fully funtional and the other with some slight bugs. The intention is for you to become familiar with pipeline operation, hazard handling, and the simulator interface.

The two versions of the simulator have been installed as the programs `solve_tk` and `buggy_tk` in the directory:

```
/afs/cs.cmu.edu/academic/class/15347-s98/public/sim/
```

These programs are compiled to run on the class Alpha machines. You must telnet to these machines from a host that supports the X11 interface.

4. pAlpha Implementation

Figure 1 illustrates the structure of the pAlpha implementation. This implementation is styled after the MIPS implementation described in Chapters 5 and 6 of the textbook. The figure is taken from the class handouts. The rectangular blocks in the figure denote *pipe registers*, a set of registers that hold the state used by the pipeline stages. Note that the program counter `PC` is one such pipe register, while the others are labeled by the states between which they sit. Embedded within the stages are additional state elements: the instruction memory in IF, the register file in ID, and the data memory in MEM. To keep things simple, the

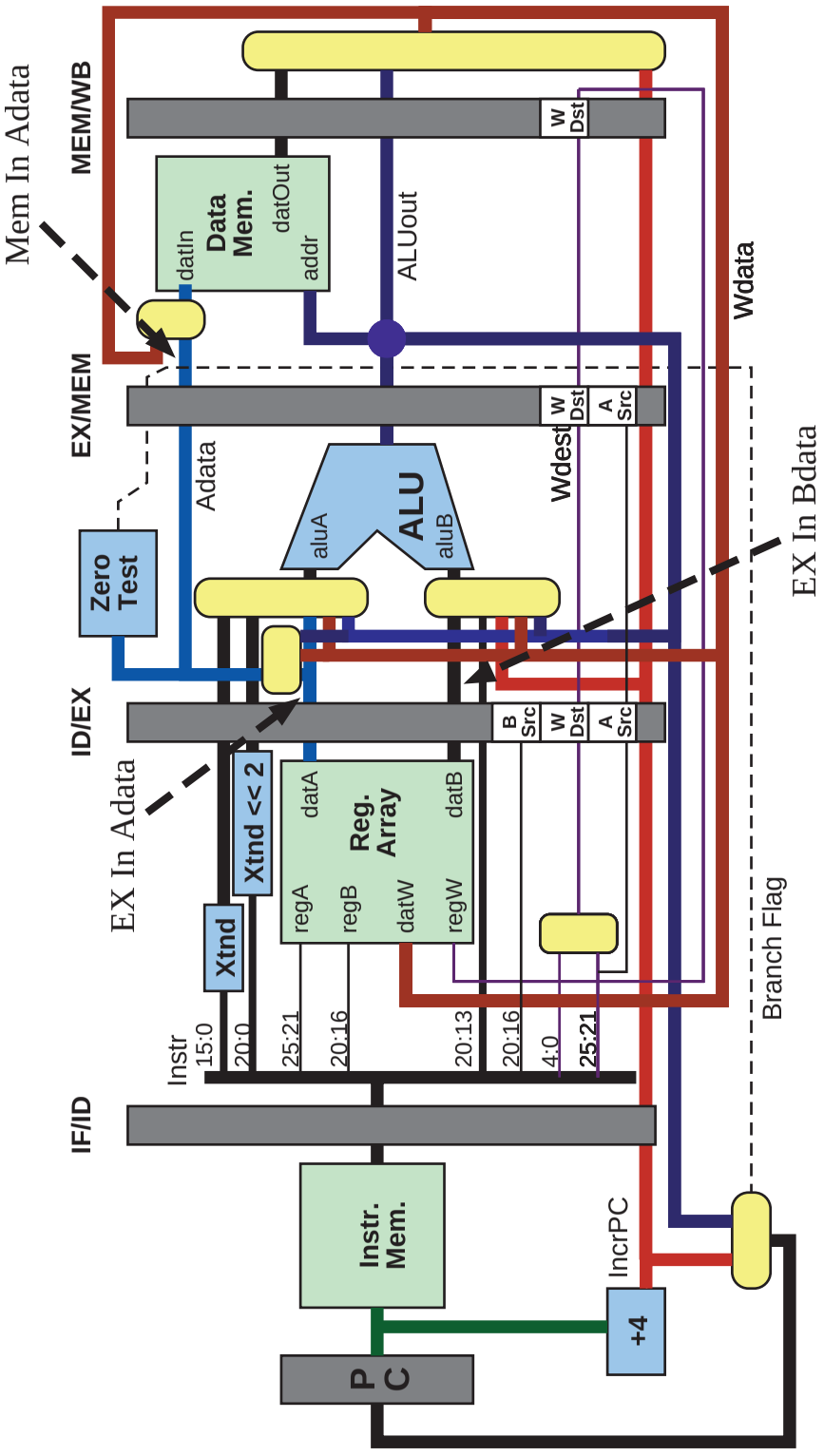

Figure 1: Detailed pAlpha Pipeline Organization (From class handouts)

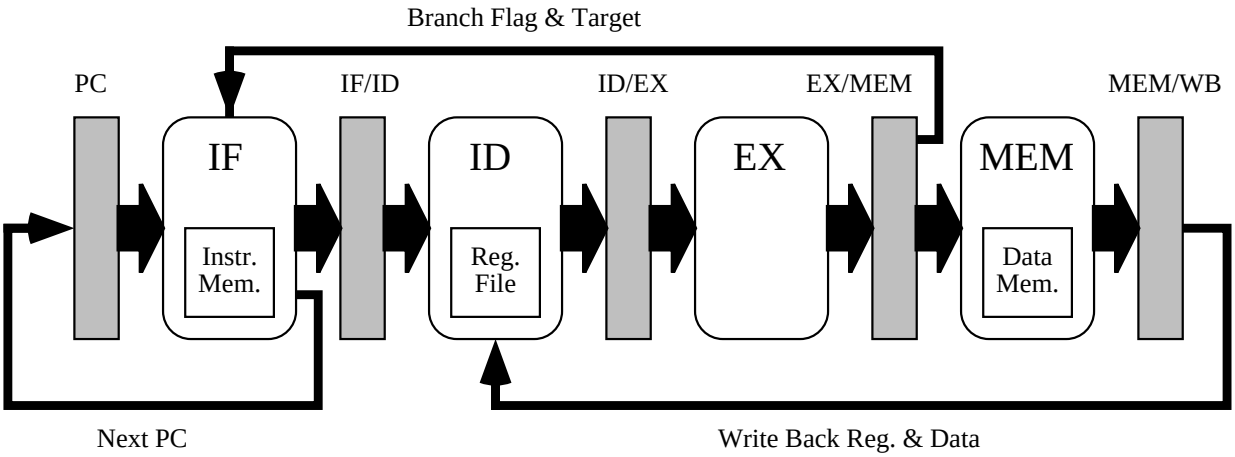

Figure 2: Simplified pAlpha Pipeline Organization

instruction data memories are distinct. In the actual processor, there are indeed separate instruction and data caches, but these both access a common main memory. Also shown are the major functional units: an adder in IF to increment the program counter, and an ALU in EX to compute data values, effective addresses, and branch targets, and a "Zero Test" block in EX to compute branch conditions. Also shown are the bypass paths used to support register forwarding.

Figure 2 shows a simplified version of the pipeline structure. The rounded rectangles denote the logic of each of the pipeline stages, while the arcs denote the signal connections.

A pipe register has a current state and a next state. Each pipeline stage takes the current state of one or more pipe registers and generates the next state of one or more pipe registers. One cycle of the pipeline consists of two phases: during the *operate* phase the pipe stages compute new values for the registers, while during the *update* phase, the pipe registers store these values and deliver them as inputs to the next stage.

Note that there is no explicit write-back stage WB. Instead, the write-back logic is incorporated into the decode stage ID, to avoid a conflict at the register file.

The version of the simulator obtained by executing `solve_tk` is fully functional. The version obtained by executing `buggy_tk` has four bugs. Part of your assignment will be to identify these bugs.

5. GUI Version of the Simulator

For this part of the assignment you will be working with the GUI version of the simulator.

When you invoke the simulator with a code file (in the '.O' format described below) as a command line argument, two windows will appear on your machine as illustrated in Figures 3 and 4. The first provides the overall control for the simulator as well as displaying the state of the pipeline and the registers. The second provides a listing of the code and tracks the instructions as they progress through the pipeline.

Viewing the control panel in Figure 3 from top to bottom, we find the following regions:

Run Controls A set of buttons that control the simulator activity:

Quit Exits the simulator

Go Starts (or restarts) the simulator. Simulation continues until either an exception condition is encountered, or the **Stop** button is pressed.

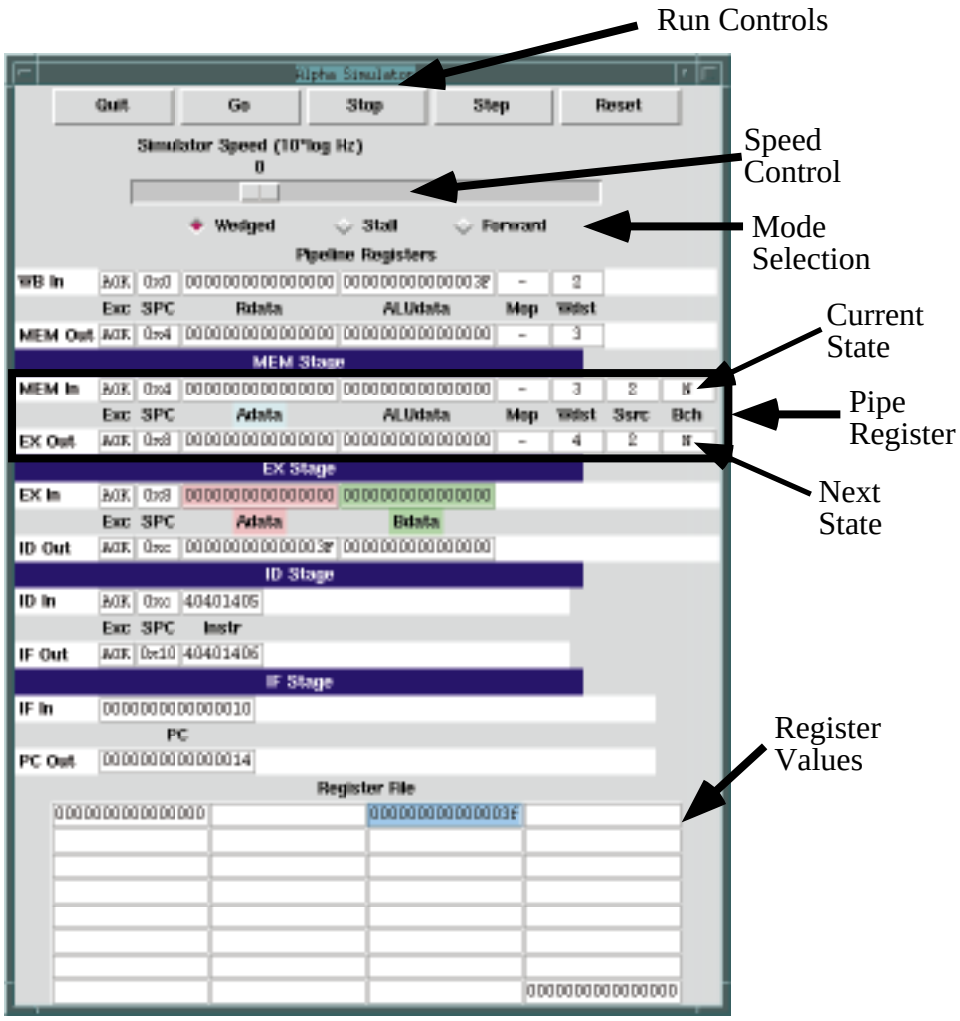

Figure 3: Main Control Panel for Alpha Simulator

Stop Stops the simulation.

Step Simulates for one clock cycle.

Reset Empties the pipeline and resets the program counter to 0.

Speed Control Controls how fast the simulator will execute. The control is logarithmic—at the extreme left the simulator runs at 0.1 cycles per second, while at the extreme right it runs at 1000 cycles per second. The default value is 1 cycle per second.

Mode Selection Controls the simulation mode. These are:

Wedged This is the version you will be given for Lab 3. It lacks any mechanism for handling control or data hazards.

Stall This (should) be the version that handles hazards by stalling the pipeline appropriately until the hazard can be resolved.

Forward This (should) be the version that handles hazards by forwarding whenever possible.

Pipeline State This region displays the state of all of the *pipe registers*, oriented with the PC on the bottom up to the MEM/WB register on the top.

Each pipeline register is represented by two rows of boxes. The upper row indicates the current state of the register and is named after the primary stage which uses it. The lower row indicates the next state of the register and is named after the stage which updates it. For example, the figure shows a box around pipe register EX/MEM. This register is updated by the EX stage and used (primarily) by the MEM stage. The top row, labeled **MEM In**, indicates the current state of the register, while the lower row, labeled **EX Out** indicates the next state of the register.

The *pipeline stages* are indicated in blue: each stage takes input from the current state of the preceding pipe register, and computes the next state of the pipe register following the stage.

Register State These are shown as 8 rows of 4 registers each, with the values displayed in hexadecimal. A blank entry is one that has never been updated. Its value is 0. The most recently updated register is indicated in blue.

The fields of the pipeline registers are as follows:

PC The program counter register (hex).

Exc The exception status for the stage. As an instruction progresses through the pipeline, its condition can go from `AOK`, meaning everything is fine, to some exceptional condition. Once the exception reaches the WB stage, the simulator will halt.

SPC Indicates which instruction is at this stage in the pipeline (hex). This information would not normally be maintained by the hardware. It is included here to aid debugging. A bubble in a stage is indicated by `----`.

Adata This is the data read from the A port of the register file (hex).

Bdata This is the data read from the B port of the register file (hex).

ALUdata This is the data generated by the ALU (hex).

Mop Indicates the memory operation to be performed, either '`R`' (read), '`W`' (write), or '`-`' (none).

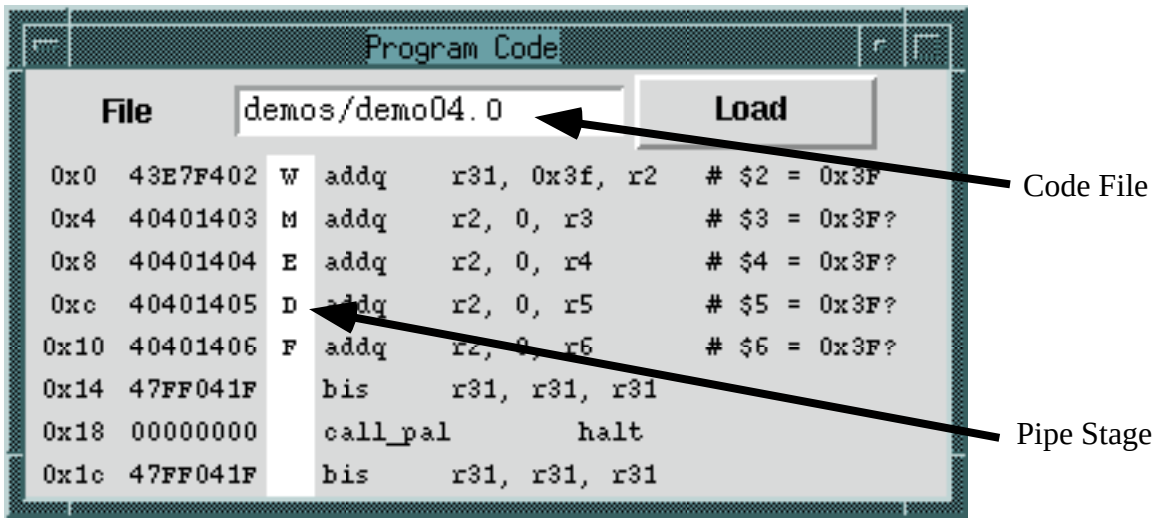

Figure 4: Code Window for Alpha Simulator

Wdst The destination register for write back (decimal).

Ssrc Identifies the source register for a store operation (decimal). You'll notice that this information is not currently used by the simulator, but you might find it useful in implementing some of the forwarding.

Bch Indicates whether (`Y`) or not (`N`) a branch will be taken.

To help visualize forwarding there are three colors defined, corresponding to the `Adata` (pink) and `Bdata` (green) fields of the EX stage, and the `Adata` (blue) field of the MEM stage. During normal operation, these operands are taken from the corresponding field of the ID/EX or EX/MEM pipeline register, as indicated by the dashed arrows in Figure 1. In the event of bypassing, however, these operands may be supplied from other locations, using the multiplexors indicated in Figure 1. The color moves to indicate the identity of the source field.

6. Object Code

The simulator reads in an ASCII version of object code. We denote these object code files with the suffix '`.O`'. Each line consists of an address, the instruction (both in hexdecimal) optionally followed by text, containing, for example, an assembly code version of the instruction. The simulator executes using the hexadecimal coded instruction. You can't change the code by simply editing the assembly code comments in the .O file.

Using an Alpha machine, you can generate this code automatically. First, create a `.s` file. Then assemble this file to get a `.o` file. Finally, disassemble this code using the program `dis`, e.g., with the command:

```
dis -h test.o > test.O
```

The code is displayed in a window such as that shown in Figure 4. At the top is an entry box where you can specify the file name and load a file by pressing the **Load** button. Each line of code is displayed giving its location, the hexadecimal code, and any text that appeared on that line in the .O file. Also indicated is the pipeline stage for any instruction being executed.

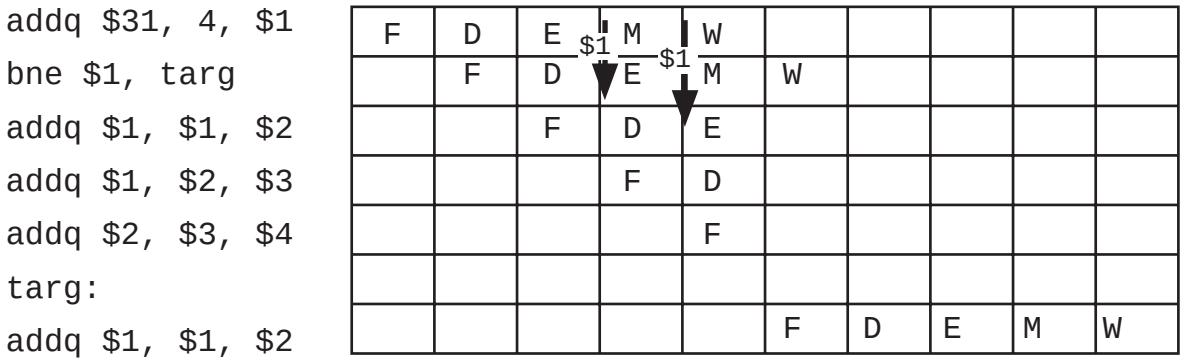

Figure 5: Example Pipeline Diagram

7. Your Task

Your job is to run 8 different simulations, performing all combinations of the following:

- Two different versions of the simulator (solve and buggy).
- Two different operating modes (Stall and Forward).
- Two different code examples (`pipe1.0` and `pipe2.0`).

For each, you should fill out the pipeline diagram. Note that you only need to diagram a subset of the two programs. In particular, the diagram for `pipe1.0` starts with the third instruction and ends with the ninth. The diagram for `pipe2.0` starts with the first instruction and ends with the ninth.

Fill in the appropriate boxes with the letters F, D, E, M or W. Where forwarding happens, draw a vertically downward arrow from the forwarding source stage to the destination stage. Label the arrow with the register that is being forwarded. Where a stage gets stalled, just fill in the same letter twice. When an instruction gets canceled, leave the remaining boxes blank. Figure 5 shows an example of a pipeline diagram in the desired format for a fragment of code running in Forwarding mode.

For the buggy versions you should identify the source of the bug and its effect on the program. There are 4 bugs: one for each of the buggy simulation runs. Some of these bugs are “functional” bugs, causing the program to yield incorrect results, while the others are “performance” bugs, causing the program to run for more cycles than is required, but eventually producing the correct results.