CS 740, Fall 1997
Assignment 2:
Handling Pipeline Hazards

Assigned: Wed., Sept. 24
Due: Mon., Oct. 6

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 report for your entire group, identifying all of the group members.

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 file:

`/afs/cs.cmu.edu/academic/class/15740-f97/public/asst/asst2/files.tar`

You will hand in a hard copy document for this assignment. You should also provide us with a version of your code file `stages.c`. Do this by naming your file *last*`-stages.c`, where *last* is the last name of one of your group members, and copying this file to the directory

`/afs/cs.cmu.edu/academic/class/15740-f97/public/asst/asst2/handin`

Include as comments near the beginning of this file the identities of all members of your group.

Formatted text is preferred to hand written.

3. Introduction

The purpose of this assignment is to gain a deeper understanding of how pipelined processors are implemented. Our method of doing this will be to create a C program that "simulates" pMIPS, a pipelined implementation of a subset of the MIPS architecture. Although C is not an ideal language for describing and simulating hardware designs, one can get a high level view of how the hardware operates by using an appropriate coding style.

You can pick up the entire set of files by copying and untarring the file `files.tar`. The source files include the following:

`mips.h` Macros defining the pMIPS instruction format. These are generally consistent with the MIPS architecture.

`sim.h`, `sim.c` Simulator framework.

`stages.h`, `stages.c` A partial implementation of the pipeline stages. You will need to fix and extend the code in the file `stages.c`.

In addition, there are some files containing utility routines for the simulator, user interface code, plus subdirectories `tests`, `demos`, and `exc` containing some sample machine programs.

The code can be compiled to generate two different user interfaces:

`mips_tty` A batch-oriented interface that prints all kinds of trace information out as it executes.

`mips_tk` A graphic-user interface based on Tcl/Tk that lets you watch and control the simulator execution.

The GUI version is far more pleasant to use. The batch interface is provided as the fall-back on systems that do not support Tcl/Tk. Also, the batch version works better for doing systematic testing of your solution.

You may want modify the file `mips_tk` to include pathnames for the directory in which you install your code. Also, you can find out which version of Tk is available on your machine by running `tk_version`. Change your Makefile accordingly, and give the command `make mips_tk`.

A correctly working version of the simulator has been installed as:

`/afs/cs.cmu.edu/academic/class/15740-f97/public/sim/solve_tk`

You might find it useful as an aid in debugging your own code. Versions have been generated for most of the Andrew machine types. See the class WWW pages for a more detailed discussion of platforms.

4. pMIPS Instruction Set

Figure 1 illustrates the subset of the MIPS instruction set implemented by pMIPS. In this figure, the field names match those in the file `mips.h`. The notation `L2` means that the value will be shifted left by two when computing the effective address, and `DPC` means the address of the instruction in the branch delay slot.

Note that it includes only 7 classes of instructions:

Register Operations Shown in the chart as `Fun`. These are `SLL`, `ADDU`, `BREAK` (used to halt the simulator), `NOR`, `OR`, `SLT`, `SLTU`, `SUBU`, and `XOR`.

Immediate data Shown in the format as `FunI`. These are `ADDIU`, and `LUI` (Load Upper Immediate).

Load The `LW` instruction.

Store The `SW` instruction.

Branches Shown in the chart as `Bx`. These are `BEQ` and `BNE`.

Fixed Jumps Shown in the chart as `Jx`. These are `J` and `JAL`. The particular jump type is encoded in the `Op` field. The jump target is given by concatenating the high order 4 bits of `DPC`, 26 bits from the instruction, and 2 zero bits.

Register Jumps These are `JR` and `JALR`. The particular jump type is encoded in the `Fun` field of the instruction. (Observe that they use the same opcode as the register operations). For `JALR`, field `Rs` designates the jump target, while field `Rd` designates the register in which to save the return program counter. Note that if you write the assembly command: '`jal $4`', the assembler will generate the code for '`JALR $31, $4`'. The `JAL` instruction always stores the return program counter in register 31.

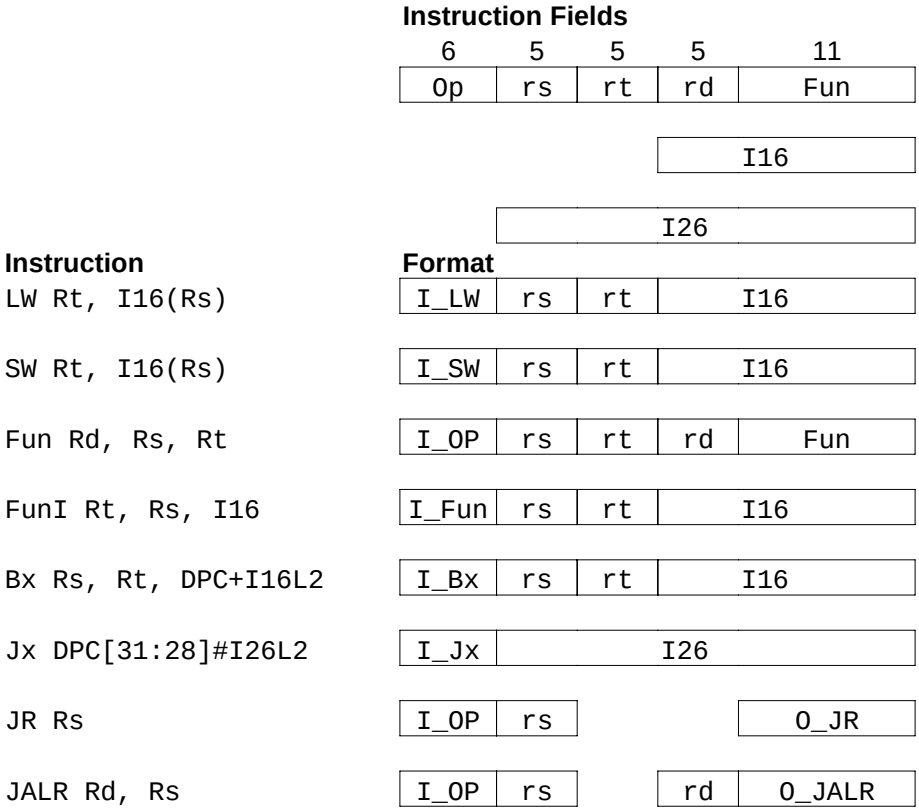

Figure 1: Instruction Format for pMIPS

The intention of the implementation is to make it follow the same semantics as with MIPS. This includes having a one cycle delay slot for both jumps and branches, requiring memory references to be aligned, etc. Of the instructions listed, all are to be fully implemented except for `SLL` (shift left logical). For this instruction, we ignore the shift amount altogether. Its main use is as a NOP instruction: an instruction word consisting of all zeros corresponds to the assembly code statement: `sll $0,$0, 0`.

Note: We will allow the instruction following a load to use the loaded result as an operand. This was not allowed in MIPS prior to the MIPS R4000.

5. pMIPS Implementation

Figure 2 illustrates the structure of the pMIPS implementation. This figure is taken from Fig 6.12 of Hennessy & Patterson, *Computer Organization and Design: The Hardware/Software Interface*, 2nd edition, Morgan-Kaufmann Publishers, 1997, an undergraduate version of our textbook. The vertical rectangles 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 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, an adder in EX to compute branch targets, and an ALU in EX to compute data values and effective addresses.

Figure 3 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 pipeline provided to you has serious deficiencies. In particular, there are no interlocks, stalls, or forwarding. Therefore both data and control hazards are handled incorrectly. Furthermore, none of the jumps are implemented. They would be handled as illegal instructions. Otherwise, the design should be correct. **Please notify us if you find any bugs in the code**.

6. Simulator Operation

The simulator is a "behavioral" simulator, meaning that it mimics the actions of the pipeline stages without modeling their detailed circuit structure.

Each pipe register is implemented by a data structure maintaining two states: `current` and `next`. Each pipe stage is implemented by a procedure that updates the next state fields of a pipe register (or two, in case of IF) based on the current states of one or more pipe registers. One cycle is simulated by invoking each of these stage procedures, and then for every pipe register copying the next state to the current state.

Extra features are included to facilitate implementing pipeline stalls. In particular, it is possible to set a pipe register to either *stall* or *bubble* on the next update phase. When the register is set to stall, the current state will not be changed the next time the pipe registers are updated. When the register is set to bubble,

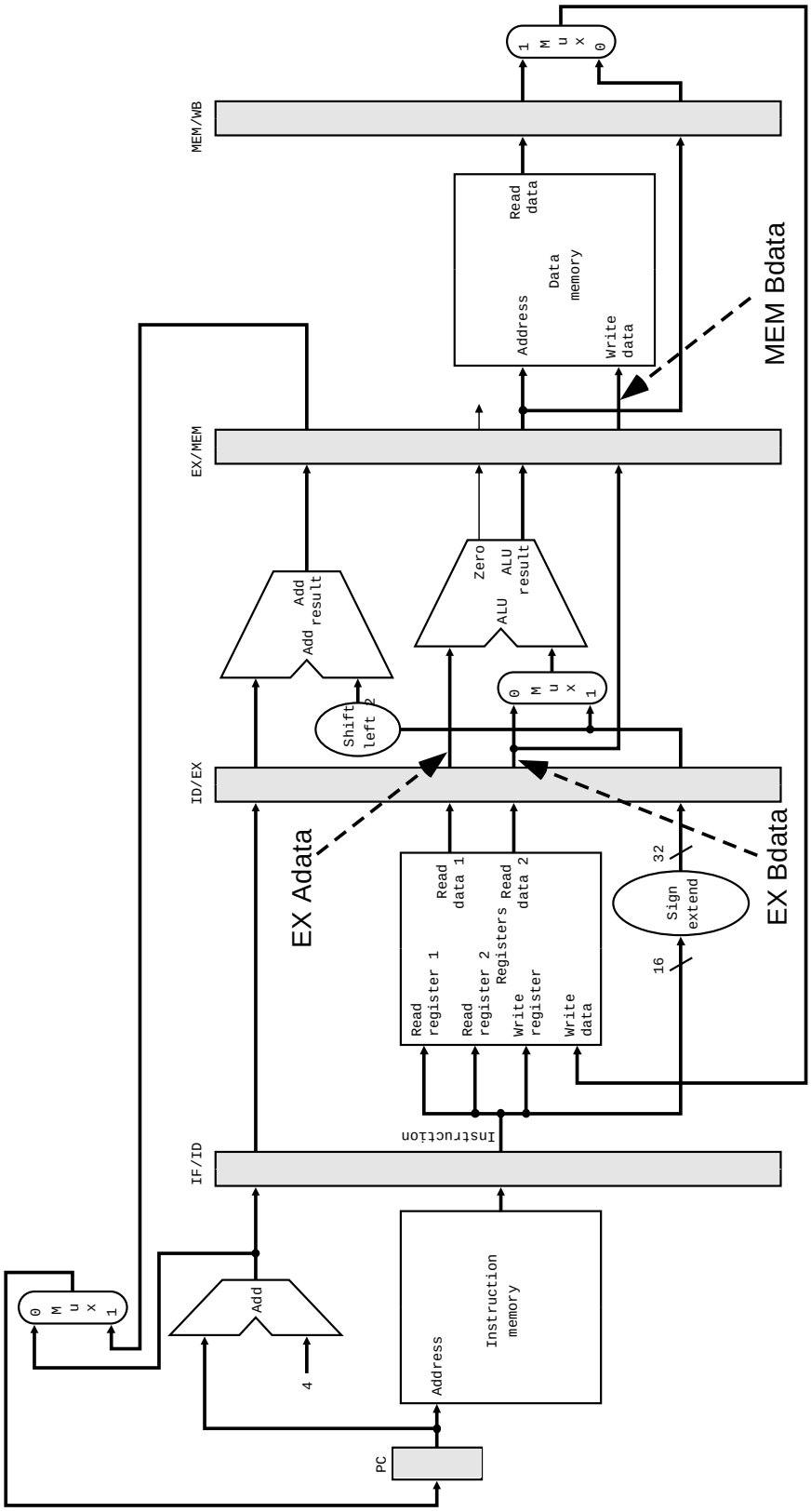

Figure 2: Detailed pMIPS Pipeline Organization (From Hennessy & Patterson)

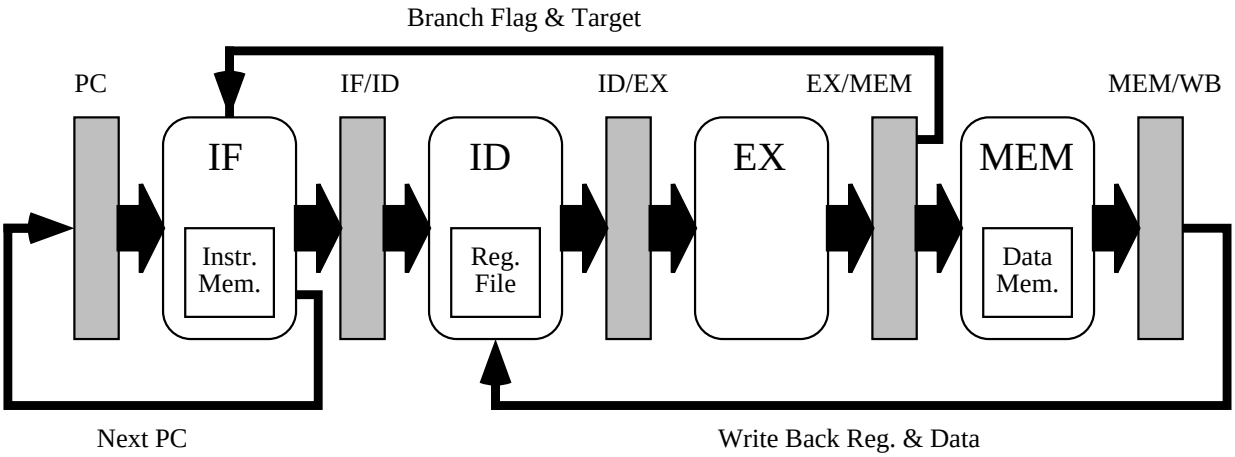

Figure 3: Simplified pMIPS Pipeline Organization

the current state will be set to all 0's the next time the pipe registers are updated. The various signals in the stages are encoded such that this will effectively create a NOP in the pipeline.

7. GUI Version of the Simulator

The GUI version of the simulator is generated from the source files with the command:

```
make mips_tk
```

When you invoke this program 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 4 and 5. 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 4 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.
- **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 only mode your initial simulator can actually perform. It lacks any mechanism for handling control or data hazards.

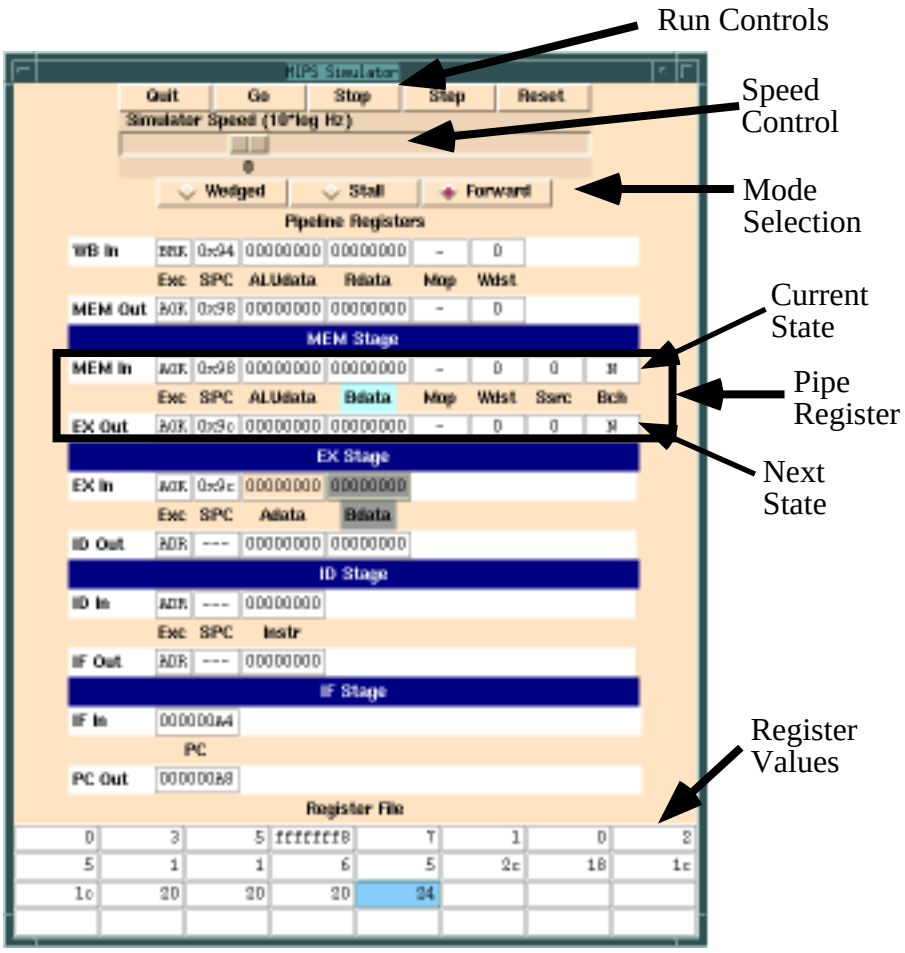

Figure 4: Main Control Panel for MIPS Simulator

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 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 4 rows of 8 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).

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

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 `Bdata` (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 2. In the event of bypassing, however, these operands may be supplied from other locations. The color moves to indicate the label of the source field.

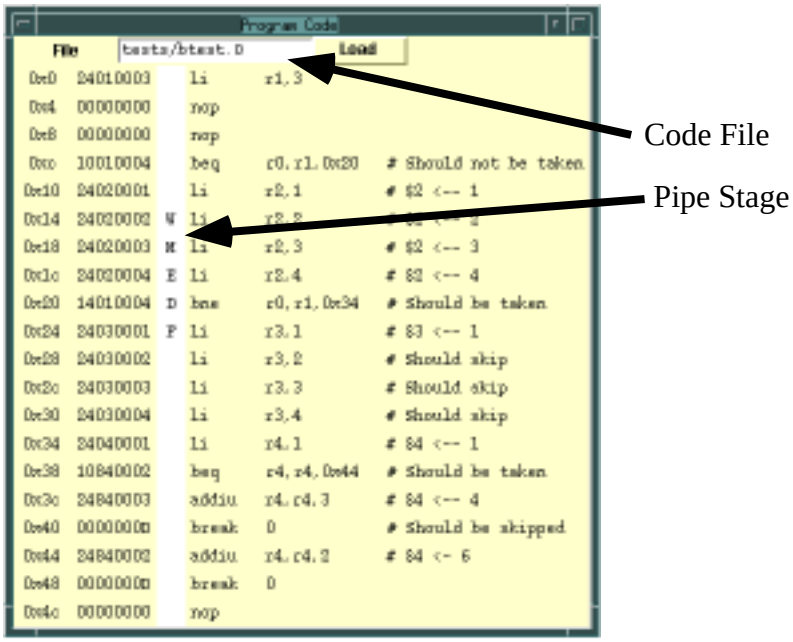

Figure 5: Code Window for MIPS Simulator

8. 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 a MIPS 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
```

We have noticed that some SGI machines create disassembled code that is incompatible with our simulator. It is safer to use a DECSTATION to generate the `.O` files. The machines reached by telnetting to `unix.andrew.cmu.edu` will suffice. Although you must generate .O files on a MIPS machine, the simulator can run on any machine. You only have to use the MIPS machine to generate the test code.

The code is displayed in a window such as that shown in Figure 5. 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.

Subdirectories `tests`, `demos`, and `exc` contain some sample code files. Three files in the `tests` subdirectory: `btest.O`, `dtest.O`, and `jtest.O` contain test cases to stress control hazards, data hazards, and jump instructions respectively. The fourth file, `recurse.O`, is provided to test the proper working of recursion. Note, however, that the tests are not exhaustive. You will need to generate additional tests to make sure your implementation is correct. The `demos` and `exc` subdirectories contain the demonstration examples used in the class lectures.

9. Your Task

For this assignment you have four tasks:

1. Implement the four jump instructions.
2. Implement the proper handling of hazards in *stall mode*.
3. Implement the proper handling of hazards in *forward mode*.
4. Generate test code that will test all instruction types and all hazard possibilities.

All of the "hardware" modifications involve changing only the code in the file `stages.c`. You may change your own versions of the other files, e.g., to print out stuff while debugging, but your simulation should run properly with the original versions, since `stages.c` is the only file you will be allowed to hand in.

9.1. Jump Instructions

These instructions should be implemented in a manner consistent with the rest of the design. Think of the different procedures provided as blocks of hardware that you want to keep as simple as possible. One approach is to view jumps as branches that are always taken. The only difference is that the jump targets must be computed according to the MIPS convention, and that instructions `JALR` and `JAL` must save the value of `DPC`+4 in a register (`Rd` for `JALR` and $31 for `JAL`).

You can implement these instructions by making appropriate changes to the procedure `do_ex_stage`.

Note that these instructions introduce new forms of control and data hazards, e.g., on their register arguments as well as (possibly) register 31. Still, you can test your implementation by executing code with enough nop's inserted to guard against any hazards.

9.2. Hazard Handling in Stall Mode

This mode should be followed when the global variable `sim_mode` is set to `S_STALL`. It should rely only on stalling. This includes the handling of branch hazards—only instructions that will definitely be executed should be fetched.

You can implement this version by inserting appropriate code into the procedure `do_stall_check`. At the end this procedure there is some code that will do the actual stalling for you: you only need to tell it *when* to stall. You can stall at the IF, ID or EX by setting the appropriate variable to 1. The three variables involved are `stall_if`, `stall_id` and `stall_ex`.

All the information you need to do a succesful job is provided to you: you can get the current pipe-register state by means of the global variables `id_in`, `ex_in` etc., and the computed next state is available via arguments to the procedure. Both current and next state will be valid, since all the stages have already computed their outputs by the time `do_stall_check` is called.

To guide your implementation, we have indicated where your code should be located by provided comment dividers, named `START STALL MODE` and `END STALL MODE`. **All your stall-mode code should be located between those two dividers. In addition, you are neither allowed to create any static variables nor to change/create any global variables.** There is no need to create or modify new global state to come up with a correct solution.

9.3. Hazard Handling in Forwarding Mode

For forwarding mode, you will need to use the following mechanisms:

- Use forwarding (i.e., bypassing) whenever possible to handle data hazards without incurring any delays.
- Add stalls to handle any data hazards for which forwarding alone does not suffice.
- Handle branches by assuming they will not be taken, and canceling any instructions that should not have been fetched in the event the branch is taken. The way to do this is to call the procedure `sim_cancel_stage` from within `do_stall_check`. Remember that the instruction in the delay slot should be executed in either case.
- You may handle jumps either in the same way as branches (always mispredicting!) or by stalling.

This mode should be followed when the global variable `sim_mode` is set to `S_FORWARD`. You can implement this version by inserting appropriate code into the procedures `do_ex_stage`, `do_mem_stage`, and `do_stall_check`. In the latter, the code will be similar to that of the stalling mode, although you'll need to stall in fewer cases. For the code within the EX and MEM pipeline stages you will implement three "multiplexors" at locations marked in Figure 2. The A and B muxes in the EX stage can override the `Adata` and `Bdata` operands coming from the ID/EX pipeline register. The S mux in the MEM stage can override the `Bdata` operand coming from the EX/MEM pipeline register. Your code should implement the control for these multiplexors by setting the variables `amux`, `bmux` and `smux` appropriately.

To guide your implementation, we have indicated where your code should be located by provided comment dividers, named `START FORWARD MODE` and `END FORWARD MODE`. **All your forward-mode code should be located between those dividers. In addition, you are neither allowed to create any static variables nor to change/create any global variables, except for amux, bmux and smux.** Again, there is no need to create new global state to come up with a correct solution.

9.4. Testing

The test cases in `btest.O`, `dtest.O`, and `jtest.O`, and `recurse.O` are a good start, but they are not comprehensive. You should carefully analyze all the different instruction types, all their possible pipeline interactions, and generate test code that will exercise these interactions. Note that you need not consider every possible instruction for all possible data values. Instead you can group instructions into classes and try them for representative data values.

Writing lots of test code by hand and then running them on the GUI interface is not a good idea—it is very time consuming and hard to get reliable results. Instead, you should set up a systematic "test bench," consisting of a semi-automatic way to generate, simulate, and evaluate test cases. Using a scripting language such as Perl or Tcl can be very useful for this task. Refer to the lecture discussion on microtests as one possible approach.

For this problem you should hand in the following:

- Show us how you tested your 4 new jump instructions.
- A description of your testing methodology for data hazards. You should document your testing methodology by creating table(s) of all the different hazard types, as well as an indication of how you test that this hazard is handled properly. Look at the lecture slides for the sort of table we expect to see here.

- The results of running your code against these tests.
- A description of your testing methodology for control hazards. You should try to devise a systematic way to tabulate possible control hazards, e.g., instructions that should or should not be executed within some distance of a branch or jump.
- The results of running your code against these tests.

Remember that you must also handle hazards induced by a load instruction followed by an instruction requiring the loaded result. Be especially careful to test hazards involving register 0.

9.5. Additional Guidelines

- It is important that you do not change any global state other than the mux signals, nor create any new global state.
- For the hazard handling, please make sure *all* of your code is located between the appropriate comment dividers.
- Although it may be tempting, do not rearrange what gets computed in what stage, or add any additional pipeline state.
- Remember that each stage must obey the protocol of taking pipe register current state and computing pipe register next state that will feed into the next stage. With forwarding, you will find that the EX and MEM stages access the current state of multiple pipe registers.
- The stall logic is implemented by a separate procedure `do_stall_check` that is called between the operate phase of the stages and the update phase of the pipe registers. This procedure can therefore make use of both the current and next statet of all the pipe registers, but its only effect should be to cancel or stall some of the pipeline stages.
- Your implementation will require writing around 100 lines of code.

10. Hand In

1. Your version `stages.c` (electronic handin only).
2. A description of how you implement the jump instructions.
3. A description of your stall-mode hazard handling mechanism. Document the conditions under which you stall stages. Don't just show the code!
4. A description of your forward-mode hazard handling mechanisms. Document the conditions under which you stall or cancel stages and forward data.
5. Documentation of your testing methodology (see above).