CS 740, Fall 1996
Homework Assignment 2

Assigned: Sept. 26
Due: Thurs., Oct. 10, 1:00PM

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 Web page `assigns.html` in the class WWW directory.

For this assignment, you will want to retrieve the files in the directory

```
/afs/cs.cmu.edu/academic/class/15740-f96/public/asst/asst2
```

You will hand in a hard copy document for this assignment. You should also provide us with a version of your code file `ufix.c`. You can do this by either:

- Sending it as email to `Randy.Bryant@cs.cmu.edu`
- Putting it in an AFS directory to which we have read access. [You are on your honor not to go snooping around looking for versions of this file from other groups].

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 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.

	Instruction Fields				
	6	5	5	5	11
	Op	rs	rt	rd	Fun
				I16	
		I26			
Instruction	**Format**				
`LW Rt, I16(Rs)`	I_LW	rs	rt	I16	
`SW Rt, I16(Rs)`	I_SW	rs	rt	I16	
`Fun Rd, Rs, Rt`	I_OP	rs	rt	rd	Fun
`FunI Rt, Rs, I16`	I_Fun	rs	rt	I16	
`Bx Rs, Rt, DPC+I16L2`	I_Bx	rs	rt	I16	
`Jx DPC[31:28]#I26L2`	I_Jx	I26			
`JxR Rs`	I_OP	rs			Fun

Figure 1: Instruction Format for pMIPS

`sim.c`, `ufix.c` A partial implementation of the simulator. You will need to fix and extend the code in the file `ufix.c`.

In addition, there are some files containing utility routines for the simulator, user interface code, plus a subdirectory `tests` containing some test 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.

[**Note:** You may want modify the file `mips_tk` to include pathnames for the directory in which you install your code.]

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 Bxx. 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 Shown in the chart as JxR. These are JR and JALR. The particular jump type is encoded in the Fun field of the instruction.

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`.

Architectural Extension: Just to make things interesting, we will allow the instruction following a load to use the loaded result as an operand. This is not allowed in MIPS.

5. pMIPS Implementation

Figure 2 illustrates the pipeline structure of the pMIPS implementation. The design is based on one described in Chapter 6 of Hennessy & Patterson, *Computer Organization and Design: The Hardware Software Interface*, a different textbook than the one we use. The vertical rectangles in Figure 2 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. The rounded rectangles denote the logic of each of the pipeline stages, while the arcs denote the signal connections. Note that some of the stages have embedded memories within them, but these are not treated as pipe registers.

A pipe register has input signals and output signals. Each stage takes the output values of one or more pipe registers and generates the inputs to 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

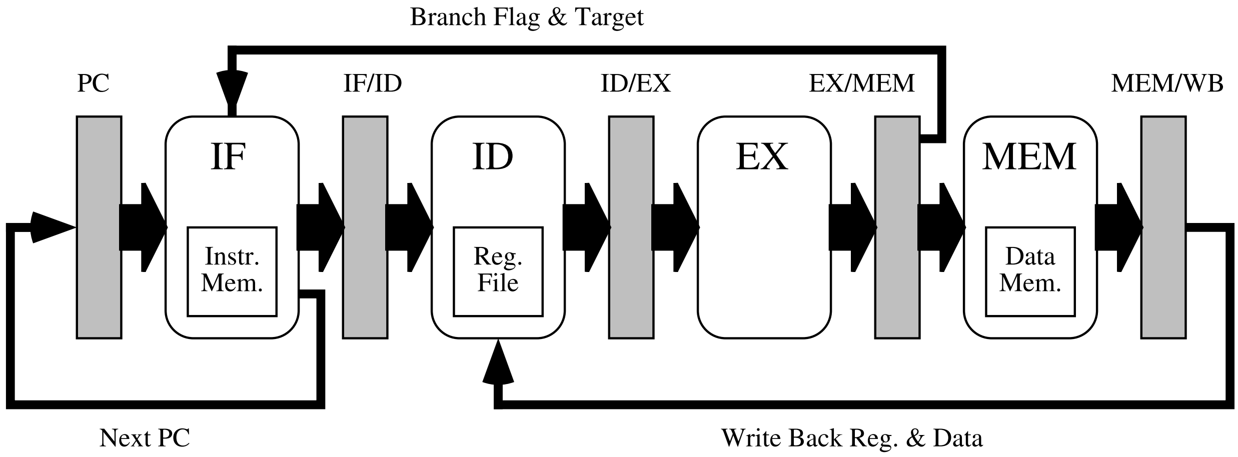

Figure 2: Simplified pMIPS Pipeline Organization

the register inputs, while during the *update* phase, the pipe registers store these inputs and produce them as outputs.

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.

To keep things simple, there are separate memories for instruction and data. In the actual processor, there are indeed separate instruction and data caches, but these both access a common main memory.

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`, representing the register outputs and inputs, respectively. Each pipe stage is implemented by a procedure that updates some of the next state fields of one or more pipe registers 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, 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.

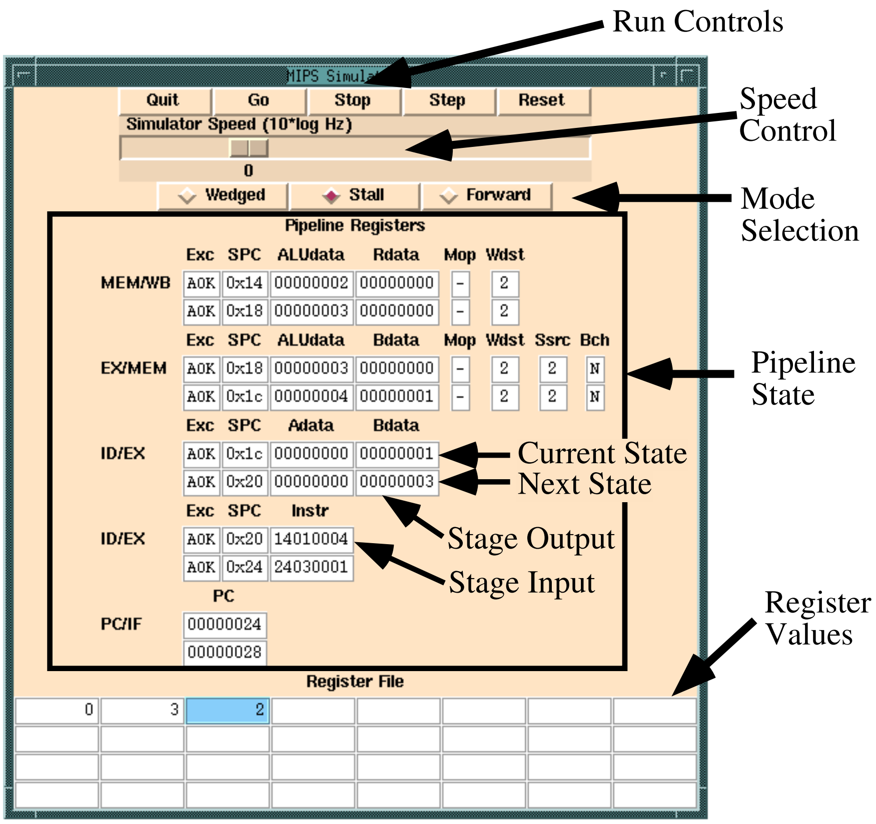

Figure 3: Main Control Panel for MIPS Simulator

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 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.
- **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.
- **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, while the lower row indicates the next state. With this orientation, a pipeline stage can be viewed as lying between two pipe registers. It takes its input, shown as the upper row of the pipe register below it, and computing its output, shown as the lower row of the pipe register above it.

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 Indentifies 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.

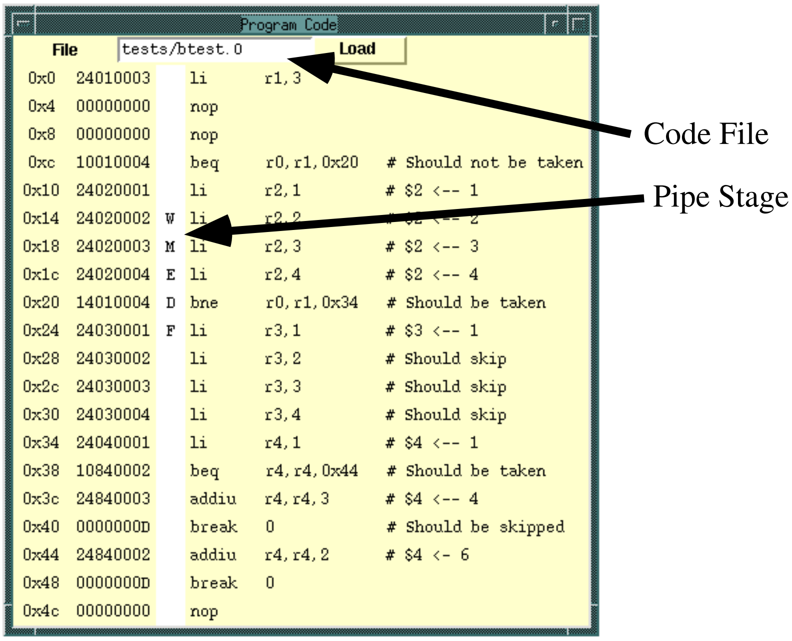

Figure 4: Code Window for MIPS Simulator

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. 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
```

Remember that the simulator executes using the hexadecimal coded instruction. You can't change the code by simply editting the assembly code comments in the .O file.

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.

The three given files `btest.O` and `dtest.O`, and `jtest.O` contain test cases to stress control hazards, data hazards, and jump instructions respectively. Note, however, that the tests are not exhaustive. You will need to generate additional tests to make sure your implementation is correct.

9. Your Task

For this assignment you have three tasks:

1. Implement the four jump instructions.

2. Implement two different fixes to the design to handle hazards properly.

3. 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 `ufix.c`. You may change your own versions of the other files, e.g., to embellish the user interface, but your simulation should run properly with the original versions.

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 register 31.

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

Remember that you must handle not only the usual hazards for the MIPS pipeline, but also those induced by a load instruction followed by an instruction requiring the loaded result.

You should implement two different forms of hazard handling.

Stalling: The first should rely only on stalling. This mode should be followed when the global variable `sim_mode` is set to `P_STALL`. You can implement this version by inserting appropriate code into the procedure `do_stall_check`.

Forwarding: The second should 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. 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 `P_FORWARD`. You can implement this version by inserting appropriate code into the procedures `do_ex_stage`, `do_mem_stage`, and `do_stall_check`.

9.3. Testing

The test cases in `btest.O`, `dtest.O`, and `jtest.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.

You should document your testing methodology by creating a table of all the different hazard types, as well as an indication of how you test that this hazard is handled properly.

To put everything together, write code that implements a recursive routine using a runtime stack, following the MIPS conventions for argument passing, stack structure, register usage, etc. For example, it might sum all the numbers from 1 to n. One way to do this would be to generate the assembly code on a MIPS machine and then translate it into our format. You may reduce the size of the stack frame to save on data memory.

9.4. Additional Guidelines

- 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 outputs and computing pipe register inputs. With forwarding, you will find that the EX and MEM stages access the outputs 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 utilize values at the inputs or the outputs of the pipe registers, but its only effect should be to make some of the registers stall or bubble during the update phase.
- Your implementation will require writing around 150 lines of code.

10. Hand In

1. A description of how you implement the jump instructions.
2. A description of your hazard handling mechanisms. Document the conditions under which you stall or cancel stages and forward data. Don't just show the code!
3. Documentation of your testing methodology.