CS 347, Spring 1996
Laboratory Assignment 2

Assigned: Feb. 27
Due: Tue., Mar. 19, 3:00PM

1. Policy

You may work in a group of up to 3 people in solving the problems for this assignment. **You will hand in one writeup, which must identify all the members of the group.**

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/15347-s96/public/labs/la2`

You will hand in your assignment electronically. Further details on how to submit the assignments will be available soon.

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.

The files in the assignment directory include the following:

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

`mips_sim.c` A partial implementation of the simulator. You will need to fix and extend this code.

In addition, there are some files containing utility routines for the simulator, plus some test code as well as routines to help you generate your own tests.

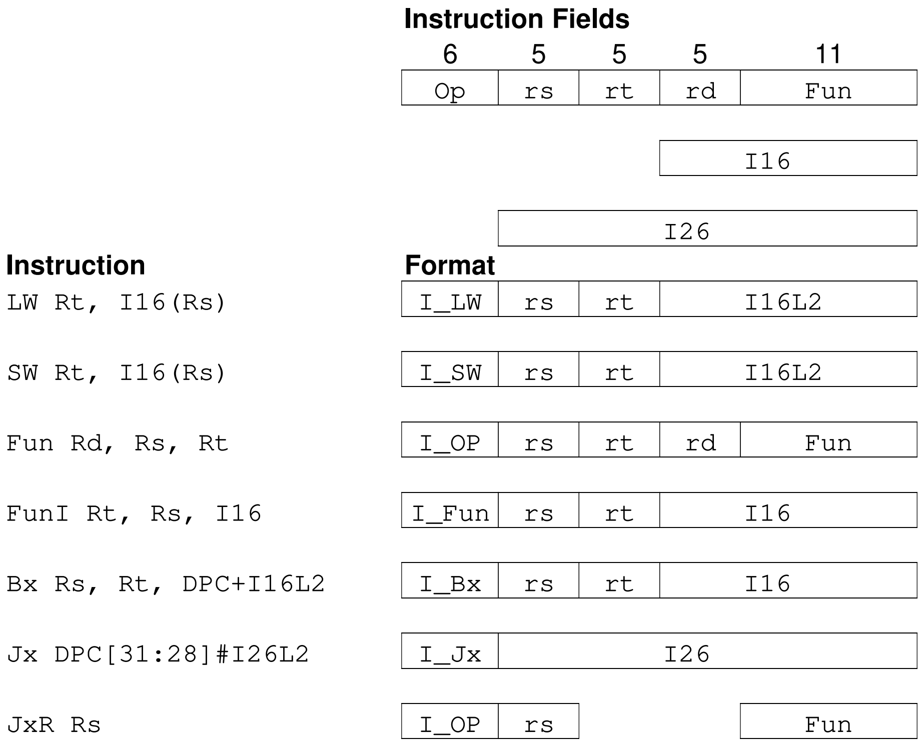

Figure 1: Instruction Format for pMIPS

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 6 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).

Memory These are `LW` and `SW`.

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.

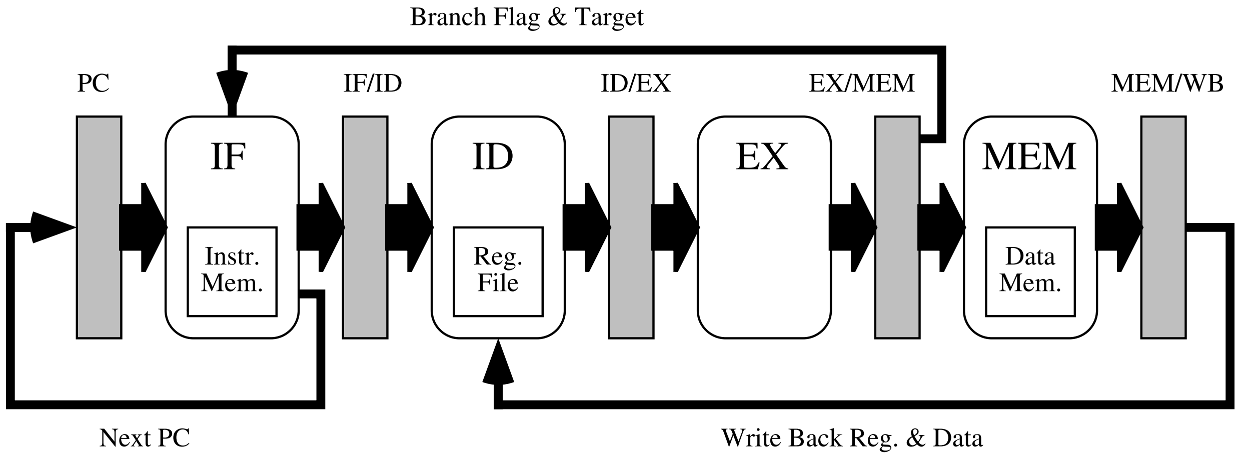

Figure 2: Simplified pMIPS Pipeline Organization

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 of the pipeline structure of the pMIPS implementation is based on the one described in Chapter 6 of Hennessy & Patterson, *Computer Organization and Design: The Hardware Software Interface*. 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 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. The Simulator

The code in the file `mips_sim.c` is a fully functional, although not very fancy, behavioral simulator for pMIPS. It should be possible to compile and execute this on any Unix computer. When given an "object code" (described below) file, it will begin execution at instruction location 0 and continue until either a `BREAK` instruction is encountered, or a preset cycle limit is exceeded. If the `verbose` flag is true, it prints lots of information about what's going on, although interpreting it requires keeping track of how instructions move through the pipeline.

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. At present, the only use of stalls is to avoid executing the instructions following a `BREAK`.

6.1. Non-pipelined simulation

If the flag `non_pipelined` is true, the simulator does not pipeline instructions, but executes only one instruction at a time, executing each pipeline stage in turn. This mode is free of hazards.

7. 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 hexadecimal) optionally followed by text, containing, for example, an assembly code version of the instruction.

To help you generate this object code, a facility is provided in which you write the code as MIPS assembly. See the examples in the files `btest.s` and `dtest.s`. The makefile contains rules for compiling these first into `.o` files and finally generating the `.O` file.

Note: this translation can only be done on a MIPS computer in the Andrew environment.

The two given files `btest.O` and `dtest.O` contain test cases to stress the control and data hazards, respectively. The files `btest.answer` and `dtest.answer` give the register states that these tests should generate. Note, however, that the tests are not exhaustive. You will need to generate additional tests to make sure your implementation is correct.

8. Your Task

For this assignment you have three tasks:

1. Implement the four jump instructions in the non-pipelined version of the simulator.
2. Enumerate all the possible hazards that can occur among instructions in the pipelined version of the simulator. For each hazard, generate a short fragment of test code that demonstrates this hazard by giving a different result for the pipelined version of the simulator and the non-pipelined version.
3. Fix the design to handle hazards properly.

8.1. 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.

Remember that `JAL` and `JALR` should set register 31 to the address of the instruction following the delay slot.

When you extend the design to handle hazards, be sure to handle data hazards on any register argument as well as on register 31.

8.2. Hazard Enumeration and Testing

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

For each hazard, generate a short fragment of test code that demonstrates this hazard by giving a different result for the pipelined version of the simulator and the non-pipelined version.

For example, the following piece of code displays a read-after-write (RAW) hazard with arithmetic instructions:

```
addi $1, $0, 1
addi $1, $1, 1
```

On the non-pipelined version, this correctly results in $1 having the value 2. On the pipelined version, this incorrectly results in $1 having the value 2.

8.3. 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. Your hazard handling should utilize the following mechanisms:

1. Use forwarding (i.e., bypassing) whenever possible to handle data hazards without incurring any delays.
2. Add stalls to handle any data hazards for which forwarding alone does not suffice.
3. 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.

8.4. Further Testing

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. 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. You may also want to increase the size of the data memory.

8.5. Additional Guidelines

- Although it may be tempting, do not rearrange what gets computed in what stage.
- Remember that each stage must obey the protocol of taking pipe register outputs and computing pipe register inputs.
- The stall logic is implemented by a separate procedure 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.
- You may add more state to the pipe registers, although not much is required.
- Your implementation will require writing around 150 lines of code, depending on how much status reporting you want to provide.
- You will not be graded on the speed of your implementation. You will be graded on the clarity of your code. Do not sacrifice clarity for speed.

9. Hand In

1. A list of possible hazards, with a description of each, with the snippets of code that demonstrate those hazards.
2. A description of how you handle each possible hazard. You should describe how the hazard can arise, what method you use for handling it, and how the method is implemented. Don't just show the code!
3. A description of how you implement the jump instructions and how you handle any resulting hazards.
4. A description of your testing procedure.
5. Your modified version of the code.

You are welcome to embellish the simulator, to implement additional instructions, to implement exception handling, to improve the mechanism for generating object code, etc. Some amount of extra credit will be given for efforts above and beyond the requirements.