Computer Architecture
Pipelined Processor
CPSC 321, Fall 2003, Project #2
Due Date: 12/08/2003 noon, via `turnin`.
Demonstrate your project the same day or earlier
Work in groups of up to four students.

1 Objective

The goal of this project is to pipeline your single-cycle MIPS-Lite processor.

2 Problem 1: Pipelining Your MIPS-Lite Design

2.1 Problem 1a: Pipelining

You have to implement the following five-stage pipeline:

IF Instruction Fetch: Access the instruction cache for the instruction to be executed.

ID Instruction Decode: Decode the instruction and read the operands from the register file. For branch instructions, calculate the branch target instruction address and compare the registers.

EX Execute: Bypass operands from other pipeline stages. One of the following activities can occur:

- For computational instructions, the ALU or the shifter performs the arithmetic or logical instructions.
- For load or store instructions, the data address is calculated.

MEM Data Memory Access: Access the data cache for load or store instructions.

WB Write Back: Write result to register file.

Add the appropriate pipeline registers to your single cycle design. Assume that memory accesses take only one cycle in your implementation.

Here are the instructions you must implement:

Type	Instructions
arithmetic (unsigned)	`addu, subu, addiu`
arithmetic (signed)	`add, sub, addi`
logical	`and, andi, or, ori, xor, xori`
shift	`sll, sra, srl`
compare	`slt, slti, sltu`
control	`beq, bne, bgez, bltz, j, jr, jal`
data transfer	`lw, sw, lui`

You should implement the components you need in Verilog. All Verilog models should correspond to realistic components (*e.g.*, register, comparator, *etc.*). No super-composite components, *e.g.*, a branch unit that takes in the opcode, operands, and PC, and outputs a new PC, or something like that.

All modules should be falling-edge triggered.

Make sure to verify the coding of instructions such as `bgez` in Appendix A of [PaHe]. Note that the "`rt`" field is actually used to distinguish instructions `bgez` and `bltz`.

2.2 Problem 1b: Initial Testing

To test your program, you should write diagnostic programs, similar to those that you wrote in Project 1. The general structure of the programs should be some calculations, data manipulations, *etc.*, followed by `sw`'s to the main memory. Then, at the end of the program, and at points during the execution, you can dump memory to show that the correct values were correctly written to memory.

Keep in mind that you haven't handled hazards yet (Problem 2), so you must be careful that your test programs don't try to use values too soon after they are generated.

3 Problem 2: Handling Hazards

Here is a list of the hazards you must handle:

Data Hazards. Data hazards occur when the data produced by an instruction is used as an operand by subsequent instructions.

- The ALU is one source of data hazards. Add the necessary forwarding logic and buses so that the result of an ALU operation can immediately be used by the following three instructions without waiting for the data to be written in the register file.
- Load instructions also present a data hazard because the data is not available until the end of the MEM phase. The MIPS instruction set specifies that load instructions have a one cycle delay. That means that the compiler cannot generate code sequences where the data of a load will be used during the following cycle. Implement your data-path so that it has one load delay slot.

Control Hazards. Branch instructions present a common case of control hazards. Comply with the MIPS instruction set definition and implement your processor so that it has one branch delay slot that is always executed regardless of the result of the branch.

Structural hazards. Are there any in this design? If so, explain what they are and how you are handling them. If not, why not?

Write or modify programs to test all the different hazard cases. Remember that hazards do not necessarily occur between two adjacent instructions. They can happen between two instructions that are separated by another instruction (or two?). Consider the following lines of code:

```
add    $1, $2, $3
sll    $5, $6
sub    $8, $1, $7
```

The `add` and `sub` instructions have a data hazard, yet there is an `sll` between them. Be sure to check these kinds of cases.

4 Deliverables

Turn in a copy of your Verilog code, processor schematic, your diagnostic program(s) and a report detailing your design decisions and any implementation methods, worthwhile describing. Also turn in simulation logs that show correct operation of the processor. These logs should show the operations that were performed, and then the contents of memory with the correct values in it. You do not need to turn in waveforms.

As part of your report, explain to us your testing philosophy.

- Why do you think that you have a working processor?
- How much time did your team spend on this project?