Carnegie Mellon University
18-347 Introduction to Computer Architecture

Lab 1: Multiported Register Files and State Machines

Due: The week of September 15th, 2003 prior to start of Lab

(100 points)

Fall 2003

Objective

The purpose of this lab is to begin implementing components of your superscalar MIPS processor core. In particular, this lab involves designing and implementing a multiported register file which can support concurrent reads and writes from several sequential instructions. This lab also involves building a simple, but powerful, state machine which will be used in your processor.

Introduction

This lab starts with the implementation of one critical component in your superscalar MIPS processor core: the register file. The register file is a unit in the processor which supplies the functional units (ALU, cache write ports) with input operands and stores the results of calculations and loads for subsequent use. Modern instruction sets typically supply the programmer with 8 - 32 architecturally-visible registers for integer computations.

This lab requires more time than the warmup project from Lab 0, so start early!

Part 1: 3-ported Register File

We went over the internals of a 3-ported register file in 18-240. In this part of Lab 1, you will implement a small register file for the MIPS processor that is suitable for executing one instruction at a time (similar to the classical MIPS R2000 processor). To support a single instruction, the register file must allow two concurrent reads and one write. You are to design, implement and test a register file in structural Verilog (the storage registers may be described by `` `include ``'ing the behavioral Verilog file /afs/ece/class/ece347/LABS/components/dreg.v). The register file should have the following characteristics:

- Inputs: Two 5-bit source register numbers (one for each read port), one 5-bit destination register number (for the write port), one 32-bit wide data port for writes, one write enable signal, clock, and reset
- Outputs: Two 32-bit register values, one for each of the read ports
- When a source register number equals the destination register number, the register file should always output the current value of the addressed D-flip-flop (which may change when the new value is clocked in)
- Register zero ($0) should always return a value of zero when read. No state should change if

an instruction specifies $0 as a destination register

For handin, you must include a block diagram of your register file implementation (including major units such as muxes, register arrays, data path and control path logic, etc....). Hand in the Verilog code for your register file, your Verilog test cases, and annotated input/output waveforms from your tests (annotations should explain what test case accomplishes).

Part 2: A 6-ported Register File

To support superscalar execution (multiple instructions executing in parallel), the register file must be able to supply enough operands for all of the executing instructions. Your superscalar MIPS core will eventually execute up to two instructions every cycle, requiring four register read ports and two independent register write reports.

Build upon your register file from the previous part by adding two more read ports and one write port (don't forget to save a copy of the previous part separately!). You should be careful to assign each port to a specific instruction (first or second); order does matter and the ports are not created equal! Because you are now supporting two instructions which should execute sequentially (this means the programmer expects to see one instruction fully completed before starting the next instruction), some troublesome corner cases — known officially as *hazards* — must be detected.

1) Consider the following code sequence:

```
R3 <- R1 + R2

R5 <- R3 + R4
```

If these two instructions were executed simultaneously, it is possible for the second instruction to receive an old value for R3 if it reads the register before R3 has been written by the first instruction (that's a Bad Thing!). We will give a more formal discussion of this later in the course. For now, it is sufficient to flag a data hazard where the later instruction uses the earlier instruction's result by asserting a special single bit error output line from your register file. The state of the register file is undefined after this point.

2) Another situation can occur when two sequential instructions write to the same register:

```
R3 <- R1 + R2

R3 <- R4 + R5
```

If both instructions executed without causing an exception (e.g., an illegal divide by zero), this situation is permissible. To make the processor simpler in later projects, we will also forbid this situation. An error should also be flagged and the state of the register file is undefined. This error signal will be helpful later when debugging your processor pipeline.

For hand-in, you must include a block diagram of your register file implementation, the structural Verilog code for your register file, your Verilog test cases,. and the annotated input/output waveforms from your tests.

Part 3: Finite State Machines

As you will learn later in 18-347, two-bit saturating counters are used in structures throughout computer architecture. For instance, the repetitive behavior of branches can be predicted quite accurately with an array of two-bit saturating counters.

Design and implement a Moore state machine for a two-bit saturating up-down counter in structural Verilog, using the behavioral D-flip-flop from Part 1. The counter should increment or decrement, depending upon the current state and an input signal. Since it is a saturating counter, when the highest and lowest states are reached, the counter must stay in that state until the increment input is inverted. The counter should have the following inputs:

- Clock
- Reset
- Enable: only change counter values when asserted
- Increment: increment when asserted, otherwise decrement

Output the current value of the counter.

For handin, include your state diagrams, Verilog code, test cases, and input and output waveforms demonstrating that your counter operates correctly.

Questions

Explain the difference in design complexity between your 3-ported and 6-ported register files. In Verilog, was the 6-ported register file harder to for you design and test? Why?

Modern register files may have several times as many ports and registers as the one in this project. Is it more difficult to scale by adding more ports or increasing the register count? Why?

Handin

Verilog code and test cases should be handed in electronically on AFS before your assigned lab period on the week of **September 15**. Please copy your Verilog files to:

`/afs/ece/class/ece347/handin/`*`ece_username`*`/lab1`

Hand in your waveform output, diagrams, and answers to the questions at the beginning of your assigned lab period in HH1107 on the week of **September 15.** Since this lab may be done in pairs, only one copy needs to be turned in. Please use one handin directory and place a file named **`usernames.txt`** in the directory, listing both people who worked on the lab.

Grading

- **Part 1**
  - Register file block diagram 10 points
  - Verilog code 15 points
  - Verilog test cases 10 points
- **Part 2**
  - Register file block diagram 10 points
  - Verilog code 20 points
  - Verilog test cases 10 points
- **Part 3**
  - State diagram 5 points
  - Verilog code 5 points
  - Test cases 5 points
- **Questions** **10 points**
- **Total** **100 points**