EE 108B Lab Assignment #2
Processor Datapath Design
Due: Tuesday, February 12, 2013

1. Introduction

Now that you have seen some of the benefits of the software approach to problems, we will spend the next three labs building a processor that is capable of executing MIPS instructions. In this lab, you will complete an implementation of a single cycle processor.

2. Requirements

You will be completing the MIPS processor implementation using the starter files provided to you. This will involve editing parts of the instruction fetch unit, instruction decoder, and ALU. You will be required to:

1. Write assembly programs that thoroughly demonstrate the correct execution of each instruction type.
2. Synthesize your design on a Xilinx Virtex 5 FPGA and demonstrate a working program with display instructions.

3. Implementation Details

Processor Model:
Before you start coding, look at each of the modules we have provided and try to understand what each one of them does. Ask one of the TAs if you have any questions about the model.

- mips_top.v - top level module
- mips_cpu.v – module containing all components of the MIPS processor
- mips_testbench(_text/_long).v – very basic testbenches that run instructions in sequence
- irom.v – a ROM containing MIPS instructions, you will put your test program(s) here
- instruction_fetch.v - fetches the next instruction from memory
- regfile.v – 32-entry, two-read, one-write 32-bit register file
- decode.v – decodes instructions and sets control signals for the processor
- alu.v – datapath for arithmetic/logic operations
- memory_top.v – interface with data memory and memory-mapped devices
- Several other files that you should not need to worry about

Getting Started:
All software can be run on the EE 108B cluster computers. To log in to an EE 108B cluster machine use:

```
ssh {yoursunetid}@ee108-{x}b.stanford.edu -Y
```

Where {yoursunetid} is your SUNet ID, and {x} is a number from 1 to 8. (These are the 8 lab stations in Packard 129.)

You will primarily be using Icarus Verilog, GTKWave, and the Xilinx synthesis tools for this project. To access these tools, you need to execute the following command from your home directory:

```
echo source /usr/class/ee108b/ee108.cshrc >> .cshrc
```

This has to be done only once for all labs and is used to setup your environment to run the tools. The first time you do it you will need to reload your environment for it to take effect.

Verilog Code:
Most of your work should be in the instruction_fetch.v, decode.v, alu.v, and irom.v modules. Indeed it is possible to implement the lab while modifying only the files above and without adding any new registers or wires. However, you are free to modify any files that you want in any way that you like, but you must document every file that you modify and the changes you make.

The processor we have provided will increment the program counter by 4 after each instruction, but does not allow for branches or jumps. In instruction_fetch.v, calculate the program counter in the event of jumps or branches, and determine how to assign the next PC.

In decode.v, you will have to determine which ALU operation should be performed for each instruction listed below. In many cases, the answer will be obvious (for the ADD instruction, select the add operation), but in some cases it will not be. You will also be calculating the 32-bit extended immediate value, which might be calculated differently depending on the type of instruction being executed. The instructions you must implement are:

ALU Operations: ADD, ADDU, ADDI, ADDIU, SUB, SUBU, SLT, SLTU, SLTI, SLTIU, AND, ANDI, OR, ORI*, XOR, XORI, NOR, SRL, SRA, SLL*, SRLV, SRAV, SLLV, LUI

Memory Instructions: LW, SW*

Jump/Branch Instructions: BEQ, BNE, BLTZ, BLEZ, BGTZ, BGEZ, J, JR, JAL, JALR

(* = already implemented and used in the starter code)

In alu.v you will have to calculate the result for the different operations you have specified in decode.v. Once again, some results will be straightforward to implement, while others will require more thinking. Multiple MIPS instructions may map to the same ALU operation. For example, ADD and ADDI are indistinguishable to the ALU.

Memory-mapped I/O for display:
To write to the display, store a word to the address 0xffff000c with the contents {8'b0, color, x, y}, where color is a byte containing a 3-bit RGB value (just like in lab 1), x is an unsigned byte ranging from 0 to 39, and y is an unsigned byte ranging from 0 to 31. If you are ambitious you can modify your pong.s from lab 1 to run on your MIPS processor once it is complete.

4. Simulation Instructions

You should use the provided Makefile to compile and simulate your design:

```
make // this compiles the design into an executable
make test // runs the executable (also runs make)
make wave // opens the waveform file
make text // runs a testbench that displays instruction info as text
make display // opens the display simulator (also runs clean and test)
make clean // removes test output files
```

Notes:
Any of the make targets can be run on their own and will do the correct thing. For simplicity, however, when using `make wave` simply run `make test` again and refresh from within GTKWave to avoid creating a new GTKWave instance after making changes to your code.

When you are ready to synthesize your design (i.e., you have written a thorough irom that covers many test cases and verified correct functionality), you can open the Xilinx synthesis with:

```
ise &
```

From ISE, open the project file `synth/mips.xise`, then generate the programming file and use iMPACT to program the FPGA with your design.

5. Submission Requirements

You will demonstrate your lab simulations and synthesis to the TA during office hours on the due date (or at an alternate time that you must arrange in advance).

You must submit your implementation and lab report electronically by 11:59PM on the due date. Please refer to the lab report guidelines handout for what must be in your report.

In your submission, you must include the following:

1. All your lab code in a directory (`make display` should succeed and show a frame of display output).
2. All testbenches that you wrote (if any, just your irom.v is fine).

For synthesis, write a simple irom.v that displays a pattern on the screen. You should be able to demonstrate that your processor works by explaining what is displayed. The controls for the synthesized module are similar to EE 108A modules. Press the left button to start and stop executing instructions, and press the top button to reset (though this does not clear the screen). Press the center button to see a test pattern on the display.

6. Extensions

If you have completed the above requirements, you may choose to implement one or more of the following extensions (no credit will be awarded for extensions if any of the basic functionality is missing or incorrectly implemented):

- Overflow detection: Set the alu_overflow signal in the ALU high whenever an operation results in overflow (be careful to distinguish between signed and unsigned operations). If you do this, you must provide a screenshot for ADD that demonstrates this, and your screenshot for ADDU must demonstrate that the output stays low when overflow occurs in an ADDU operation.
- Implement BGEZAL and BLTZAL.

Extensions must be shown in the demonstration and pass automated tests to earn credit.

7. Grading Rubric

Lab Report:
5 points – following report guidelines (no missing sections, title page, etc.)
20 points – thorough design discussion

Code:
5 points – turning in your code
45 points – passing automated tests

Demonstration:
25 points – synthesizing a design and showing a simple display program
+4 points – overflow detection
+6 points – BGEZAL and BLTZAL

Total: 100 points (110 possible with extensions)