E85: Digital Design and Computer Engineering
Lab 10: Multicycle Controller

Objective

In Labs 10 and 11, you will design a multicycle ARM processor in SystemVerilog and test it on a simple machine language program. This will tie together everything that you have learned in E85 about digital design, hardware description languages, assembly language, and microarchitecture, and give you the chance to design and debug a complex system. In Lab 10, you will build and test the controller. In Lab 11, you will build the datapath and test the whole system. You will need a working multicycle processor for your final exam.

1. Multicycle ARM Controller

Figure 7.31 of the textbook shows the controller for the multicycle processor implementing a subset of ARMv4. Figure 7.41 shows the Main FSM. Table 7.6 defines the Instruction Decoder. Table 7.3 and HDL Example 7.3 define the ALU Decoder. Page 400 and HDL Example 7.3 show the PC Logic. Table 6.3 and HDL Example 7.4 have the Condition Check logic.

Write a hierarchical Verilog description of the multicycle controller. The controller should have the following module declaration and should follow the hierarchy of Figure 7.31. Remember that Op, Funct, and Rd are bitfields of Instr.

```
module controller(input  logic         clk,
                  input  logic         reset,
                  input  logic [31:12] Instr,
                  input  logic [3:0]   ALUFlags,
                  output logic         PCWrite,
                  output logic         MemWrite,
                  output logic         RegWrite,
                  output logic         IRWrite,
                  output logic         AdrSrc,
                  output logic [1:0]   RegSrc,
                  output logic         ALUSrcA,
                  output logic [1:0]   ALUSrcB,
                  output logic [1:0]   ResultSrc,
                  output logic [1:0]   ImmSrc,
                  output logic [1:0]   ALUControl);
```

2. Test Bench

Develop a self-checking testbench for the multicycle controller. It should exercise all of the logic in the controller. Run your testbench and debug any errors you find.

A suggested approach is to adapt the test bench from HDL Example 4.39. Add more inputs and expected outputs. Each instruction should be applied for several cycles as the controller moves through various states to carry it out.

For greater confidence and to reduce the risk of discovering new bugs during Lab 11, optionally trade testbenches with a classmate. Test your controller with your colleague's testbench and debug any errors you find.

Debugging Hints

Unless you are extraordinarily unlucky, your controller won't work perfectly on the first try. If it did work, you would have missed out on the main learning objective of this lab and the next, which is how to systematically debug a complex system. Here are some tips to reduce the amount of time that debugging will take.

Minimize the number of bugs you have

Each bug takes a long time to locate, so a bit of extra time during the design phase can save you a lot of time during the debug phase.

- Remember that you are building hardware, so sketch the hardware you want and write the Verilog idioms that imply that hardware. Don't fall into the trap of writing Verilog code without thinking of the hardware it is implying.
- Carefully proofread your code. Make sure your signal names are spelled consistently and that module inputs/outputs are listed in the correct order.
- Synthesize your design in Quartus and look for warnings or errors. Make sure you understand which warnings are normal (e.g. no timing constraints set) and which need to be fixed. Take these warnings very seriously; they are the fastest way to detect subtle bugs in your design.
- Simulate your design with Modelsim and look for warnings when compiling. Modelsim has a different Verilog analyzer and will detect types of mistakes that don't produce warnings in Quartus. Take these warnings seriously too.

Minimize the time it takes to run a test

Once you are in the debugging phase, choose a workflow that is efficient so you can make a change to your code and rerun the test in a matter of seconds rather than minutes.

- All testing can be done in Modelsim. You do not need to use Quartus, and recompiling in Quartus is an unnecessary time-consuming step. However, if you have made major changes, you might wish to occasionally resynthesize the design in Quartus and look for warnings hinting that you've introduced new bugs.
- Add relevant waveforms in Modelsim. It's usually worthwhile to add all the signals in a module that you are debugging so that you don't have to go through the tedious process of adding more signals and resimulating. Change the radix to display 32-bit signals in hexadecimal.
- Remember that you don't need to restart Modelsim and re-add signals each time you change your code. Instead:
    - Compile -> Compile All
    - Make sure you have no warnings

- At the command line, rerun the simulation by typing
  - `restart –f`
  - `run 1000` (or however long you wish to run)

Systematically find your bugs

Inexperienced designers can waste enormous amounts of time debugging without a clear plan in mind. The following techniques can save you many hours.

- Understand what the expected inputs and outputs should be. Write down your expectations. This takes time, but will usually save far more time than it takes.
- Find the first place where a signal doesn't match your expectations. One bad signal will usually trigger others downstream, so focus your debugging on the first known error and don't worry yet about subsequent errors.
- Make sure the simulator displays all signals involved in computing the bad signal. If necessary, add them to the simulation and resimulate as given above. If one of these inputs is bad, repeat this process to continue tracing it back.
- Once all the inputs are good and the output is bad, you've localized your bug. Examine the relevant Verilog module and fix the mistake.
- Repeat this process until all bugs have been fixed.

What to Turn In

1. Please indicate how many hours you spent on this lab. This will be helpful for calibrating the workload for next time the course is taught.
2. Hierarchical SystemVerilog for your controller module matching the declaration given above.
3. Self-checking testbench. Explain how you chose your test cases to test all of the logic in the controller. Does your controller pass your testbench?

If you have suggestions for further improvements of this lab, you're welcome to include them at the end of your lab.