

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
CPEN 211 Computer Systems I, Fall 2023

Lab 5: Datapath of the “Simple RISC Machine”

Week of Nov 6 to 10 (your code must be submitted by 9:59 PM the evening before your lab session)

1 Introduction

Starting with this lab you will build a computer that implements a “Reduced Instruction Set Computer” (RISC). The ARMv7 that we learned about in lectures and used in Lab 4 is a RISC “instruction set architecture” (ISA). In Lab 5 to 7 you will build a RISC processor that, while much simpler than ARMv7, will give you a strong grasp of how computers actually work.

The “Simple RISC Machine” executes programs written using a *very* small set of instructions. Section 8 at the end of this handout provides a complete list of the instructions your computer will execute at end of Lab 7. This set of instructions is “Turing-complete” meaning your Simple RISC Machine computer could in principle run any program given enough time and enough memory.

To start, consider the following line of C code:

```
f = (g + h) - (i + j);
```

Assuming f, g, h, i, j are in registers R0 through R4, this C code can be implemented using the Simple RISC Machine (SRM) instructions:

```
ADD R5, R1, R2; // t1 = g + h
ADD R6, R3, R4; // t2 = i + j
SUB R0, R5, R6; // f = t1 - t2
```

Where we used R5 and R6 as temporary registers.

To implement a computer that can execute these instructions we will need a block of hardware that can add and subtract numbers. We also need some hardware that can store the numbers before and after the addition and subtraction operations. The following section describes one hardware design for a computer that can execute ADD and SUB instructions like those above.

1.1 Overview of the Simple RISC Machine Datapath

To execute the instructions above along with some others described later, in Lab 5, you implement the computer *datapath* shown in Figure 1. This section explains the different elements in Figure 1. While reading this section try to focus on understanding what each individual block does. In Section 2 we will see an example showing how the blocks work together.

The datapath consists of one register file 1 containing 8 registers, each holding 16-bits; three multiplexers 6, 7, 9; three 16-bit registers with load enable 3, 4, 5; a 1-bit register with load enable 10; a shifter unit 8; and an arithmetic logic unit (ALU) 2. Below we describe each component in detail.

1.1.1 Register File

The Simple RISC Machine employs the hardware structure known as a *register file* 1, shown in Figure 1. The register file is a small read-write memory containing eight locations each storing 16-bits. The number in the register name, e.g., “5” in R5, is used as the address to select one of the registers to read or write. To specify one of the eight locations requires only $\log_2(8) = 3$ -bits (even though each location can hold 16-bits) for the address.

An *individual* 16-bit register inside the register file is built using what is sometimes referred to as a *register with load enable*. Implement a register with load enable by writing (System)Verilog for the circuit in Figure 2. In Figure 2, when load is 0, out is passed through the top input of the multiplexer into the D input of the set of 16 flip-flops, which together form a 16-bit register. Thus, when load is 0 and there is a rising edge of the clock, the value in the 16-bit register does *not* change. Conversely, when load is 1 the

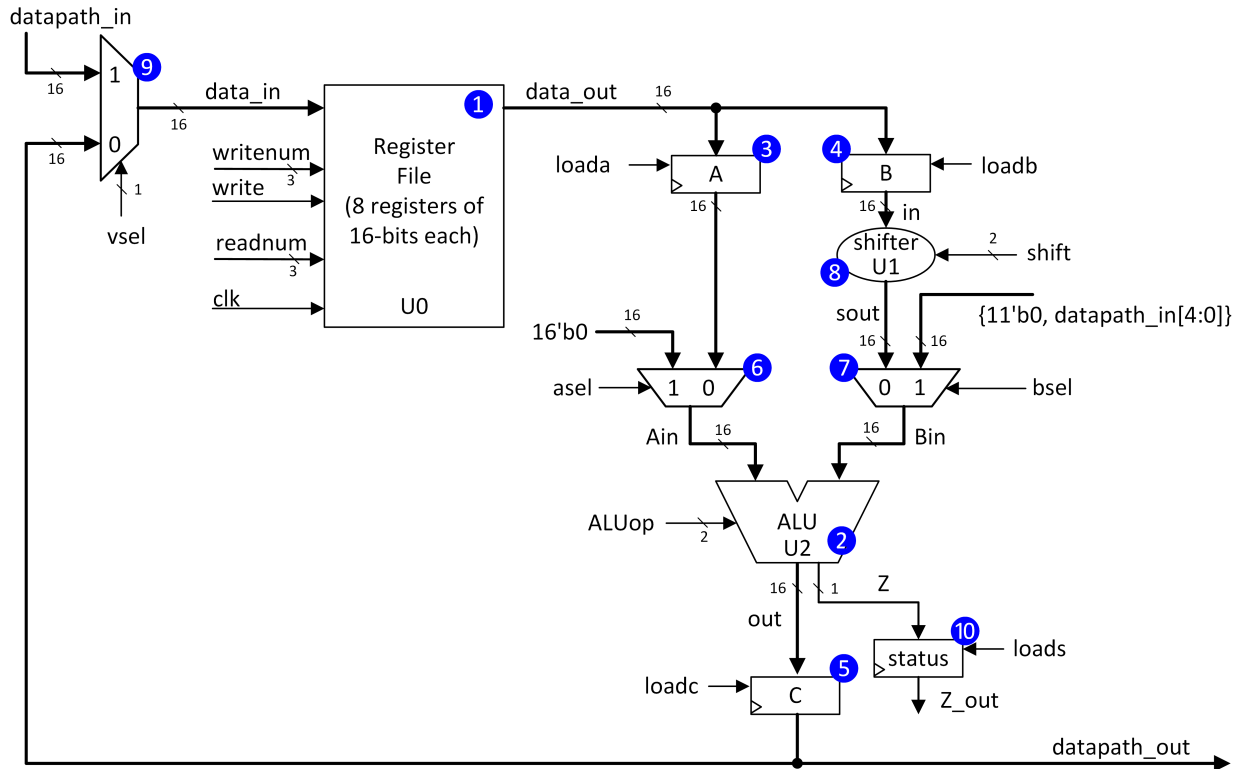


Figure 1: “Simple RISC Machine” Datapath

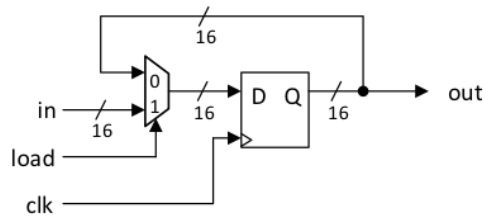


Figure 2: Register with Load Enable Circuit

value of out is updated to the value on in on the rising edge of the clock. In Figure 1 a 16-bit register with load enable is represented using the symbol shown in Figure 3.

Implement the register file out of eight instances of the 16-bit register with load enable circuit along with two binary to one-hot decoders, as shown in Figure 4. To be compatible with the autograder, your register file **must** contain signals named R0, R1, R2, R3, R4, R5, R6, R7 (Figure 4 omits R4, R5 and R6 for clarity).

Now, let’s consider how the value of variable “j” gets into and out of R3 inside the register file.

Suppose we want “j” to have the value 42 *before* we start executing our code. To put 42 in R3 you would place the 16-bit value 000000000101010 (binary for 42) on data_in, set the 3-bit input writenum to 011 (binary for 3), set write to 1 to indicate we wish to save, or *write*, the value 42 into location 3 in the register file, and input a rising edge on clk. This causes, the output of the upper 3:8 decoder to be driven to 00001000. Each output bit of the decoder is AND’ed with write. As write is 1, the load input to R3 is set to 1. On the rising edge of clk 000000000101010 will be copied to the 16-bit Q output of R3. At most one load enable input to R0 through R7 will be 1. If write is 0 all 8 load-enable signals are 0. Section 2 describes in more detail the “MOV” instruction that is used to write a constant into a register using the above steps.

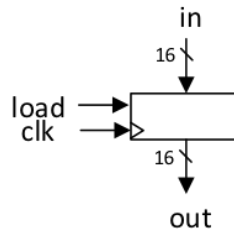


Figure 3: Register with Load Enable Symbol (Note: this symbol is used in both Figure 1 and 4)

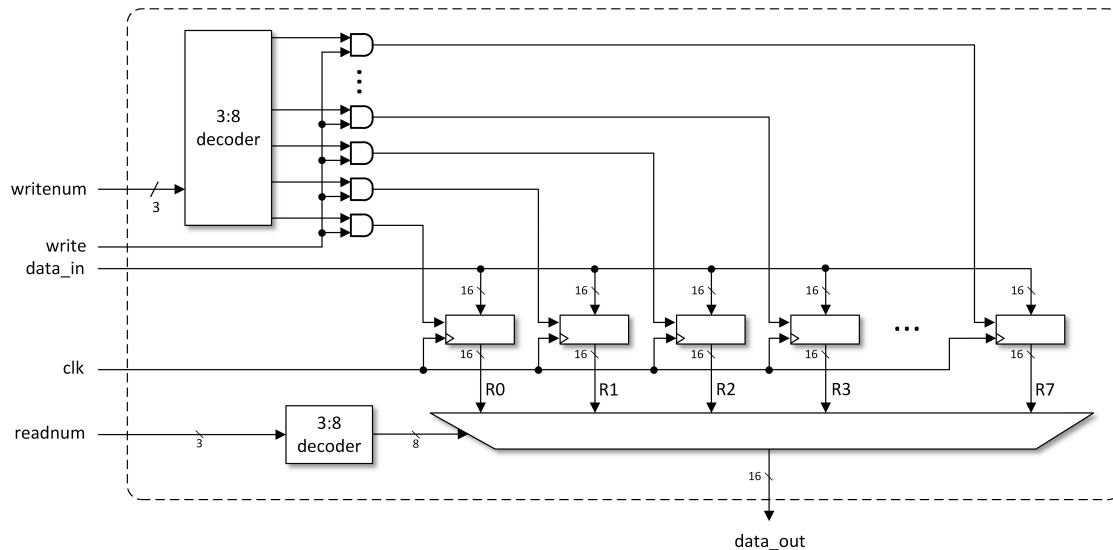



Figure 4: Register File Internal Structure (must contain signals R0, R1, ... R7 connected as shown)

To *read*, the value of “j”, which is in R3, we set the 3-bit bus *readnum* to 011. The 8-bit output of the lower 3:8 decoder will be 00001000 and this will cause the 8-input one-hot select mux to copy the value of R3 to *data_out*.

This register file is said to have two *ports*: one *write* port and one *read* port. Thus, up to one write and one read can be performed simultaneously. The read is combinational: whenever *readnum* changes, the value from the indicated register is driven out of the register file after some combinational delay. THE REGISTER READS ARE NOT COORDINATED TO THE CLOCK! The register write, however, is coordinated to the clock. At each rising clock edge, if *write* is 1, the value on the 16-bit register file input *data_in* is written into the register indicated by the value on *writenum*. This write only happens on the rising clock edge. If, at the clock edge, *write* is 0, no register is updated. Be sure you follow the Verilog style guidelines for your register file code.

1.1.2 Arithmetic Logic Unit

The  symbol, labeled 2, represents an *arithmetic logic unit* (ALU). The ALU can perform arithmetic or logical operations. This is the hardware that actual “computes” things inside a computer. The operation to be performed is indicated by the value of *ALUop* as shown in Table I

Your ALU should be purely combinational (there is no clock input). Whenever one of the inputs or *ALUop* lines change, the output changes appropriately (the Verilog “+” and “-” operations are combinational). If the main 16-bit result of the ALU (*out*) is zero the 1-bit Z output should be 1 and otherwise 0.

Value on ALUop input	Operation
00	Ain + Bin
01	Ain - Bin
10	Ain & Bin
11	~Bin

Table 1: ALU operations

shift	Operation
00	B
01	B shifted left 1-bit, least significant bit is zero
10	B shifted right 1-bit, most significant bit, MSB, is 0
11	B shifted right 1-bit, MSB is copy of B[15]

Table 2: Shift Operation Encoding

This output will be used in later labs for compare and branch instructions.

1.1.3 Pipeline Registers

The datapath contains three special 16-bit registers with load enable 345. These hold the datapath signals A, B, and C. We will use these registers while executing an individual instruction. We need at least one of the two registers A and B because the ALU is purely combinational and the register file can read out only one of R0 through R7 at a time. You may want to try to eliminate the other two registers in Lab 8 but for now you should keep them.

1.1.4 Source Operand Multiplexers

To enable more complex instructions besides addition and subtraction it is helpful if we can change the inputs to the ALU via the source operand multiplexers 67. For some instructions added in later labs these multiplexers are used to set the 16-bit Ain input to the ALU to zero. For other instructions we use them to input an “immediate operand”.

1.1.5 Shifter Unit

Inspired by the ARMv7 instruction set, the Simple RISC Machine allows one source operand of an instruction can be shifted before an operation for certain instructions. To enable this the shifter unit 8 is a purely combinational logic block that operates as follows. The shifter takes one 16-bit input from the Q output of register B 4 and outputs either the same value or the value shifted one bit to the left or right according to the value on “shift” as described in Table 2.

For example, if the input to the shifter was 1111000011001111, then the output of the shifter would be as shown in Table 3.

shift	Output of shifter
00	1111000011001111
01	1110000110011110
10	0111100001100111
11	1111100001100111

Table 3: Example shift operations starting with 1111000011001111

1.1.6 Writeback Multiplexer

Once the ALU has computed a value the main 16-bit result is captured in register C. If we want to use this value as the input to a subsequent instruction we need to write it into the register file. However, we will also want to input values into the register file from other sources. Thus, we add a writeback multiplexer 9.

1.1.7 Status Register

The status register 10 is used to remember if the output of a given ALU computation had a special outcome indicating which direction and if/else statement or loop should go. Specifically, the status register should record the Z output value from the ALU if loads is set to 1 and otherwise the status register should keep its current output the same.

2 Example Datapath Operation

Consider the addition of two registers, R2 and R3 with the result being stored in R5. The addition takes four clock cycles. During the first cycle, readnum is set to 2 to indicate we want to read the 16-bit contents of R2 from the register file. At the same time loada is set to 1 to indicate that register A should be updated on the next rising edge of the clock (note the little triangle in the bottom left of register A indicates a clock input). During the second cycle, we set loada back to 0, set readnum to 3 to indicate we now want to read the 16-bit contents of R3. At the same time loadb is set to 1. With these control input settings on the next rising edge of the clock the contents of R3 will be copied to register B. During the third cycle loadb is set back to 0, ALUop is set to “00” to indicate addition (see Table 1 above), asel is set to zero to ensure the value in register A appears at the Ain input to the ALU. Similarly, bsel is set to 0 to indicate the output of the shifter unit appears at the Bin input to the ALU. The shifter unit is combinational logic that takes the 16-bit contents of B as its input and outputs the value either unmodified, or shifted to the left or right by one bit position depending upon the control input “shift”. During this cycle the shift input is set to “00” to indicate the value in B should not be shifted. Also during this cycle, loadc is set to 1 to ensure the result of the addition is saved in register C on the next rising edge of the clock. We can optionally set loads to 1 if we want to record the “status” of the computation. In this lab the status will simply indicate if the 16-bit result of the ALU was zero. In later labs we will see how this status information can be used to help implement “if” statements and “for” loops in a language like C or Java. During the fourth cycle, loadc is set back to 0, vsel is set to 0, write is set to 1, and writenum is set to 5. Together these cause the value in register C to be fed back and written into register R5 within the register file.

Note that during the fourth cycle, the value fed back also appears on the output pins datapath_out. For this lab you can connect datapath_out to the 7-segment displays on the DE1-SoC using the logic provided in lab5_top.sv. This will be the primary way to tell if your datapath is working when it is on the DE1-SoC. However, this is not the fastest or easiest way to debug your circuit.

As alluded to earlier, the basic interface between hardware and software inside a computer is through “instructions”. Each instruction tells the computer how to move and operate upon some data. Consider an instruction of the form “MOV R2, #32”. This instruction would load the actual number 32 into register R2. This can be performed with the datapath as follows:

During the first cycle, assume the number 32 appears on the 8-bit signal datapath_in (in a later lab, we will consider more realistic memory read/write strategies). During this same cycle, vsel is set to 1, write is set to 1, and the number 2 (indicating register #2) is driven on writenum. Note that this instruction can be performed in only one cycle, unlike the ADD instruction, which takes 4 cycles. This will become important in the next assignment.

If you have trouble following the above discussion, please also review the slides “Lab 5 Introduction” on Piazza. Also, the following video shows your instructor demonstrating a working Lab 5 implementation using a DE1-SoC showing how executing instructions will look when you are done: https://cpen211.ece.ubc.ca/video/lab5_demo/lab5_demo.html.

3 Lab Procedure

You will get through the lab far more quickly if you break down the overall work into smaller parts and complete, compile and test (using a testbench) each one *before* moving on to the next. If you are working with a partner, remember you must use “pair programming”, which means you should not divide up the work but rather meet and work together on a single solution. You can work together remotely using [MS Visual Code Live Share](#). If you are worried you will not have time to do the entire lab then see the marking guideline in Section 4 to see how you can earn part marks.

3.1 Recommended Development Sequence

1. Create a new ModelSim project called lab5.
2. Add a file `regfile.sv` and write synthesizable code for your register file in this file. Make sure your register file uses exactly the following declaration which is required by the autograder (note Verilog is case sensitive):

```
module regfile(data_in,writenum,write,readnum,clk,data_out);
    input [15:0] data_in;
    input [2:0] writenum, readnum;
    input write, clk;
    output [15:0] data_out;
    // fill out the rest
endmodule
```

3. Add a file `regfile_tb.sv` to your project for your register file testbench module and inside it include your testbench module `regfile_tb` making sure to use lower case for the module name as this is required by the autograder. Your `regfile_tb` **must** contain a signal `err` that is initialized to 0 and set to 1 if an error is found and stays at 1 thereafter. The script part of your `regfile_tb` testbench should finish all tests before time 500. Section 8 provides some tips on how to write good unit level testbenches and introduces some Verilog syntax for automatically checking if the output results are correct to enable what is known as regression testing.
4. Compile `regfile.sv` in Quartus to check for synthesis errors (inferred latches, etc.). Check the resulting hardware that Quartus generates using “Tools” > “Netlist Viewers” > “RTL Viewer” to verify the hardware looks as you expect (e.g., combinational logic or flip-flops). Then, test and debug your design by compiling and simulating `regfile_tb.sv` along with `regfile.sv` in ModelSim. Use RTL level simulation so that your testbench can check the values of R0, R1, etc.
5. After you have debugged the register file should you go through the above steps for the ALU. Use the file names `alu.sv` and `alu_tb.sv`, ensure your testbench module is named `ALU_tb` and ensure your ALU module has the following declaration required by the autograder (note Verilog is case sensitive):

```
module ALU(Ain,Bin,ALUop,out,Z);
    input [15:0] Ain, Bin;
    input [1:0] ALUop;
    output [15:0] out;
    output Z;
    // fill out the rest
endmodule
```

Your `ALU_tb` must contain a signal `err` that is initialized to 0 and set to 1 if an error is found and stays at 1 thereafter. The script part of your `ALU_tb` testbench should finish all tests before time 500.

6. Implement your shifter module in `shifter.sv` and the testbench for your shifter in `shifter_tb.sv`. Ensure your testbench module is named `shifter_tb` and ensure your shifter module has the following declaration required by the autograder (note Verilog is case sensitive):

```
module shifter(in,shift,sout);
    input [15:0] in;
    input [1:0] shift;
    output [15:0] sout;
    // fill out the rest
endmodule
```

Your `shifter_tb` must contain a signal `err` that is initialized to 0 and set to 1 if an error is found and stays at 1 thereafter. The script part of your `shifter_tb` testbench should finish all tests before time 500.

7. Now that all three main datapath modules are trusted to work, instantiate them in your datapath and add the remaining building blocks. Instantiate each of the three units (Register file ①, ALU ② and Shifter ⑧) inside `datapath.sv`. Note that the autograder will assume your register file has the instance label `REGFILE` (in all caps) and that the input and outputs are consistent with the way they are referenced in `lab5_top.sv` and `lab5_autograder_check.sv` provided in the starter repo. Note the autograder and `lab5_top.sv` require that `asel`, `bsel` and `vsel` are binary select inputs. Next, add in the remaining logic blocks ③④⑤⑥⑦⑨⑩ to your datapath module using synthesizable Verilog that conforms to the style guidelines. Use no fewer than one always block or assign statement per hardware block in Figure 1. Register A, B, and C will each require an instantiated flip-flop module and an assign statement for the enable input in order to conform to the style guidelines.
8. Write a top level testbench module for your datapath called `datapath_tb` in `datapath_tb.sv`. To be compatible with the autograder your `datapath_tb` module must instantiate `datapath` using named port association (not implicit/positional) and it must define an internal signal called `err` that is initialized to 0 and set to 1 if an error is found and stays 1 thereafter. At a minimum your `datapath_tb` should test at least the sequence shown below:

```
MOV R0, #7           ; this means, take the absolute number 7 and store it in R0
MOV R1, #2           ; this means, take the absolute number 2 and store it in R1
ADD R2, R1, R0, LSL#1 ; this means R2 = R1 + (R0 shifted left by 1) = 2+14=16
```

However, to get full marks on the autograder your `datapath_tb` must be capable of identifying bugs in several alternative `datapath` modules we have crafted that containing design errors, and these may require more extensive test cases than required to get your own datapath bug free.

9. Test the overall datapath in ModelSim. If you see any suspicious outputs you should follow the debugging procedure in (b) to find relevant internal signals to add to the waveform viewer and restart and rerun the simulation.
10. Only after your overall design is working in ModelSim should you compile your top level and attempt to download to your DE1-SoC. Use `lab5_top.sv` ONLY to help with this step. If you encounter bugs here try step (d). If you still are not sure what is going on and why the results differ from ModelSim then modify `lab5_top.sv` to connect internal signals within your datapath to the LEDs on your DE1-SoC to help you follow the debugging rule “Quit Thinking and Look”.

4 Marking Scheme

A reminder that *both* partners must be in attendance during the demo. Any partner who is absent will automatically receive a mark of zero even if they did their fair share of the work.

The file `lab5_autograder_check.sv` is provided in the starter repo to enable you to verify your modules are defined consistently with what the autograder expects. **WARNING:** *The file `lab5_autograder_check.sv` is NOT the autograder that will be used mark you. Passing the checks in this file does NOT (in any way) guarantee you will not lose marks when your code is run through the actual autograder. You are responsible for designing your own test benches to verify you match the specification given earlier.* The file `lab5_autograder_check.sv` contains three test benches that you should run: The module `lab5_check_1` checks that your register file, shifter and ALU can be checked by the autograder. It should print out:

```
# CHECK #1 DONE: Your register file, shifter and ALU appear compatible with the
# autograder. ** NOTE ** You must also manually verify you had no simulation
# warnings (above) and that you have no inferred latches (e.g., using Quartus).'
```

The module `lab5_check_2` runs your unit level test benches. It should print out:

```
# CHECK #2 DONE: Your unit level testbenches appear compatible with the autograder.
# ** NOTE ** You must manually verify you had no simulation warnings
# by looking above this line in the transcript window in ModelSim.
```

The module `lab5_check_3` runs your datapath testbench. It should print out:

```
# CHECK #3 DONE: Your datapath testbench appears compatible with the autograder.
# ** NOTE ** You must manually verify you had no simulation warnings
# by looking above this line in the transcript window in ModelSim.
```

The module `lab5_check_4` checks that your datapath can be checked by the autograder. It should print out:

```
# CHECK #4 DONE: Your datapath appears to be compatible with the autograder.
# ** NOTE ** You must manually verify you had no simulation warnings
# (above) and that you have no inferred latches (e.g., using Quartus).
```

As noted in the messages above, it is important to verify you did not have any simulation warnings after simulating each of these test benches.

IMPORTANT: Check your submission repo carefully as you will lose marks if your submission does not contain a Quartus Project File (.qpf) and the associated Quartus Settings File (.qsf) that indicates which Verilog files are part of your project. This .qsf file is created by Quartus when you create a project. It is typically named `<top_level_module_name>.qsf` and contains (among others) lines indicating which Verilog files are to be synthesized. If you open up this .qsf file you should see lines that look like the following. The key part is that these line contain “VERILOG_FILE”:

```
set_global_assignment -name VERILOG_FILE shifter.sv
set_global_assignment -name VERILOG_FILE regfile.sv
set_global_assignment -name VERILOG_FILE lab5_top.sv
set_global_assignment -name VERILOG_FILE datapath.sv
set_global_assignment -name VERILOG_FILE alu.sv
```

The autograder will use your .qsf file to determine which Verilog files should be synthesized together. To be sure, note the above .qsf file is **not** the file `DE1_SoC.qsf` we provide in the starter repo for importing DE1-SoC pin assignments. Also remember to include a Modelsim Project File (.mpf), which **MUST** be called “`lab5_top.mpf`” and must include both your synthesizable verilog and your unit level testbench files (if you forget this file or it does not include your testbenches, you will at almost certainly get zero marks for Part 2 and/or Part 3). Finally include your programming (.sof) file and any waveform (.do) files. **You will lose one mark for each one of these files that is missing.**

Your mark will be computed partly by running it through an autograder and partly assigned by your TA as follows.

4.1 Autograder Marking [5 marks]

The autograder will assign a mark out of 5 as follows:

1.5 Marks 0.5 marks for each of `regfile.sv`, `alu.sv` and `shifter.sv`. For the autograder to not get confused, there must only be one file in your submitted .zip file with each of these names. Make sure your register file, ALU and shifter are free of inferred latches. Your register file, ALU and shifter must be defined in modules named `regfile`, `ALU` and `shifter` following the module declarations described in Section 3.1. You will get 0.5 marks for each unit that synthesizes, is free of inferred latches and/or other combinational-loops and passes the autograders tests.

1.5 marks The autograder will check your unit-level test benches which must be in `regfile_tb.sv`, `alu_tb.sv` and `shifter_tb.sv` and have module names `regfile_tb`, `ALU_tb` and `shifter_tb`. Each of `regfile_tb`, `ALU_tb` and `shifter_tb` must contain a signal `err` that is initialized to 0 then set to 1 if an error is found and stays at 1 thereafter. In addition, since the autograder will use designs for the ALU, register file and shifter that are different from your own your test benches should only check output signals. For each unit level test bench the autograder will try various incorrect designs to see how many your testbench flags as incorrect by setting `err` to 1. To receive full marks your testbench should catch “most” of our broken designs. As above, for the autograder to not get confused, there must only be one file in your submitted .zip file with each of these names. To be compatible with the autograder your unit level test benches should pass `lab5_check_1` and in addition **NOT** make any assumptions about the signal names inside `regfile`, `ALU` and `shifter` (i.e., do not use hierarchical signal names to refer to them) or you will get zero for this part. The sole exception is that you may access internal signals `R0`, `R1`, ... `R7` inside `regfile` since these names are required for the autograder for `datapath` as indicated in `lab5_check_4`.

1 mark For your top level testbench `datapath_tb` in `datapath_tb.sv`. Similar to the unit level test benches we will try several broken `datapath` modules to see if you catch our errors. As above, for the autograder to not get confused, there must only be one file in your submitted .zip file with this name. Your `datapath_tb` must contain a signal `err` that is initialized to 0 then set to 1 if an error is found and stays at 1 thereafter. The autograder will try various incorrect `datapath` designs to see how many your testbench flags as incorrect by setting `err` to 1. To receive full marks your testbench should catch “most” of our broken designs. To be compatible with the autograder your `datapath_tb` must pass `lab5_check_3` and in addition **NOT** make any assumptions about the signal names inside `datapath` or any modules it instantiates (i.e., do not use hierarchical signal names to refer to them) or you will get zero for this part. The sole exception is that you may access internal signals `R0`, `R1`, ... `R7` inside `regfile` since these names are required for the autograder for `datapath` as indicated in `lab5_check_4`.

1 mark For your `datapath` module which must be defined inside of `datapath.sv`. To get this mark your “.zip” file must contain the Quartus “.qsf” file containing `VERILOG_FILE` for each file required to synthesize your code as the autograder will use this file to find any files in your “.zip” required to synthesize your `datapath` module. Your `datapath` must also use the input output port names and widths implied by `lab5_top.sv` and `lab5_autograder_check.sv`. You will get this mark provided your complete design synthesizes without errors, is completely free of inferred latches and/or other combinational-loops (including in your ALU, shifter and register file) and passes our test benches.

4.2 Teaching Assistant assigned [5 marks]

Partners may get a different mark based upon their ability to answer the TA’s questions. If it becomes apparently one partner did more than two thirds of the work the partner who did less will receive a mark of zero. You **must** include at least one line of comments per always block, assign statement or module instantiation and in test benches you must include one line of comments per test case saying what the test is for and what the expected outcome is.

1.5 Marks For explaining to the TA your `regfile.sv`, `alu.sv`, `shifter.sv` and `datapath.sv` code Up to 1 mark may be deducted for violations of the style guidelines and/or for lack of comments.

1.5 Marks For explaining your test strategy in your test benches and showing your simulation waveforms. You may lose up to 1 marks if your test cases are not commented (one line per test case saying what is being tested and the expected outcome).

2 Marks For demonstrating your datapath works on your DE1-SoC using a test case of your own devising. Your TA will need to be convinced your design really works to get full marks. The following video should give you an idea how this part might look: https://courses.ece.ubc.ca/cpen211/2016/lab5_demo/lab5_demo.html. To ensure all students can be marked within the duration of your lab session your TAs will give you no more than 5 minutes to complete this part and may give you zero if you cannot complete the demo in this time. Given we will be marking you on Zoom one partner should explain what each step is while the other performs that step using their DE1-SoC (decide before your demo who will explain and who will perform the steps on their DE1-SoC).

5 Lab Submission

If you are working with a partner, your submission **MUST** include a file called “CONTRIBUTIONS.txt” that describes each student’s contributions to each file that was added or modified. If either partner contributed less than one third to the solution (e.g., in lines of code), you must state this in your CONTRIBUTIONS file and verbally inform the TA. Your TA will deduct 3 marks if CONTRIBUTIONS is missing and may deduct up to this amount if it lacks in meaningful detail. Note that submitted files may be stored on servers outside of Canada. Thus, you may omit personal information (e.g., your name, SN) from your files and refer to “Partner 1” and “Partner 2” in CONTRIBUTIONS.

In addition, if you used any artificial intelligence tools, such as ChatGPT or github copilot you **MUST** document all prompts or other inputs in a file “AI.txt”. There should be enough detail in AI.txt to generate either the same (or very similar) code in case of any allegation that your (System)Verilog was shared with other groups. A reminder you are not permitted to share AI prompts (sharing AI prompts will be treated as misconduct).

Submitted files may be stored on computer servers outside of Canada. Thus, you may omit personal information.

Submit your code using github classroom using the invite link for your section of Lab 3 available at https://cpen211.ece.ubc.ca/cwl/github_info.php. After you have submitted your changes you should see the changes on github.com in the repo created using the invite link.

5.1 Submission checklist

Ensure your repo has the following files:

6 Lab Demonstration Procedure

Lookup your assigned TA, their Zoom meeting link, your marking time and your place in the marking order at https://cpen211.ece.ubc.ca/cwl/ta_lookup.php. As in Lab 3 and 4, your TA will have the files you submitted via “handin”. However, ensure you have your DE1-SoC with you.

7 Suggestions and Debugging Tips

See the debugging video <https://youtu.be/2c3CZouKJKs> for details of how to trace a bug to its source. Some other tips and tricks that you may find helpful:

1. Code in an iterative fashion, meaning write a portion of the code and test it and correct bugs then write more code.

2. Debugging. In the very likely case that a signal (wire or reg) appears wrong in the waveform viewer, first find the Verilog corresponding to the hardware block that “drives” that signal. If there is an obvious error in the code for that block that can explain the exact wrong result you are seeing, then try fixing it. If there is no obvious error then you should not change the Verilog for that block! If you do make a change, remember to recompile your Verilog, restart the simulation and rerun the simulation. If your change did not fix the specific bug you were trying to fix, then undo it! This is important! If you make changes to your code that do not fix the bug they tend to make it harder to find the bug you were original interested in because the bug tends to “move around”. Instead of “undo” you can also comment out the “fix” code you added so you can get it back quickly in case you do end up needing it. Now, if/when you run out of things that could be wrong with the block defining the signal that looks wrong, do your best to guess which inputs to that block could lead to this incorrect output. Then add all the input signals to the block to the waveform viewer and restart the simulation and rerun it. If one of those inputs seems wrong, repeat Step 5 starting with the block that drives that signal.
3. If your code appears to work in simulation but not on the DE1-SoC, first check for and fix any inferred latch warnings found during synthesis. If this does not fix the problem, create a top-level RTL-level testbench that models the actions you are attempting using the switches on the board with the code in `lab5_top_tb.sv` and simulate it in ModelSim while adding internal signals. If the design appears to work pre-synthesis, then you may need to do post-synthesis simulations and temporarily add additional I/Os to your modules to check their values after synthesis (remember to remove those I/Os before submission).
4. SystemVerilog assertions. If you save your testbench file with the extension `.sv` and set the file properties to SystemVerilog you can make your testbench “self checking” by using the SystemVerilog assert statement. If we expect “s” to be `3'b100` at some point during the test script then we could write:

```
assert (datapath_tb.DUT.MUX1.s == 3'b100) $display("PASS");
      else $error("FAIL");
```

If you want simulation to stop on an error go to “Simulate > Runtime Options...” then select the “Message Severity” tab and change the setting for “Break Severity” to “Error”.

5. The following Verilog “force” and “release” syntax can be helpful for debugging after you put your datapath together if you later find a bug. In ModelSim from a test script and using the above external name syntax you can override the logic value generated by the circuit itself to “inject” your own test values using the Verilog keyword “force”. Continuing the example above, suppose “s” has the value “010” but you would like to see what the output “b” of instance “m” is if instead “s” was “100”. You could write the following line in your Verilog testbench script to find out:

```
force datapath_tb.DUT.MUX1.s = 3'b100;
```

Later in your test script you can go back to using the value generated by the circuit by using “release”:

```
release datapath_tb.DUT.MUX1.s;
```

6. Revision Control, Incremental Testing, Regression Tests.

To make the best use of git and github make a habit of making frequent “commits” after every major change or improvement to the code. Revision control (git) can go “hand in hand” with a practice known as “regression testing”. The idea of regression testing is that any time you change something you re-run all testbenches you have written so far to ensure no test case that *was* passing stopped working.

Regression testing works best if you make your testbenches “self checking” by printing out a “PASS/-FAIL” message at the end of the simulation. Every time you make a change to your design (e.g., the register file or ALU) or test bench, you rerun ALL of your tests and only commit changes if all existing tests “pass”. To speed this up, run ModelSim from the command line. For example, for your Lab 3 code could create a new file “regress.do” containing:

```
vlib work
vlog lab3_top.sv
vlog tb_lab3.sv
vsim -c work.tb_lab3
run -all
quit -f
```

Then, type “vsim -c -do regress.do” and hit enter. You should see outputs like you would in the transcript window in ModelSim.

8 The Simple RISC Machine Instruction Set Architecture

The information in Table 4 and 5 is only relevant to Lab 6 and 7. An assembler will be provided to you for Lab 7. Each row in these tables specifies a single instruction. The assembly syntax is in the leftmost column. Each instruction is encoded using 16-bits. The next 16 columns indicate the binary encoding for the instruction. The last column on the right summarizes the operation of the instruction. The most significant 3-bits of each instruction (bits 15 through 13) are the opcode which indicates which instruction or class of instruction is represented.

Terminology quick definitions. These will be explained in more detail in the Lab 6 to 8 handouts.

- Rn, Rd, Rm are 3-bit register number specifiers.
- im8 is an 8-bit immediate operand encoded as part of the instruction.
- im5 is a 5-bit immediate operand encoded as part of the instruction.
- sh is a 2-bit immediate encoded as part of the instruction used to control the shifter. Legal values for <sh_op> are “LSL#1”, “LSR#1”, or “ASR#1”.
- SX(f) sign extends the immediate value f to 16-bits.
- sh_Rm is the value of Rm after passing through the shifter connected to the Bin input to the ALU.
- Z, V, and N are the zero, overflow and negative flags of the status register (only Z is implemented in Lab 5).
- status refers to all three of Z, V and N.
- R[x] refers to the 16-bit value stored in register x.
- M[x] is the 16-bit value stored in main memory (added in Lab 6) at address x.
- PC refers to the program counter register (added in Lab 6).
- <label> refers to a textual marker in the assembly that indicates an instruction address

Assembly Syntax (see text)	“Simple RISC Machine” 16-bit encoding																Operation (see text)	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
Move Instructions	<i>opcode</i>			<i>op</i>		<i>3b</i>			<i>8b</i>									
MOV Rn, #<im8>	1	1	0	1	0	Rn			im8								R[Rn] = sx(im8)	
MOV Rd, Rm{, <sh_op>}	1	1	0	0	0	0	0	0	Rd	sh	Rm					R[Rd] = sh_Rm		
ALU Instructions	<i>opcode</i>			<i>ALUop</i>		<i>3b</i>			<i>3b</i>	<i>2b</i>	<i>3b</i>							
ADD Rd, Rn, Rm{, <sh_op>}	1	0	1	0	0	Rn			Rd	sh	Rm					R[Rd]=R[Rn]+sh_Rm		
CMP Rn, Rm{, <sh_op>}	1	0	1	0	1	Rn			0	0	0	sh	Rm					status=f(R[Rn]-sh_Rm)
AND Rd, Rn, Rm{, <sh_op>}	1	0	1	1	0	Rn			Rd	sh	Rm					R[Rd]=R[Rn]&sh_Rm		
MVN Rd, Rm{, <sh_op>}	1	0	1	1	1	0	0	0	Rd	sh	Rm					R[Rd]=~sh_Rm		
Memory Instructions	<i>opcode</i>			<i>ALUop</i>		<i>3b</i>			<i>3b</i>	<i>5b</i>								
LDR Rd, [Rn{, #<im5>}]	0	1	1	0	0	Rn			Rd	im5							R[Rd]=M[R[Rn]+sx(im5)]	
STR Rd, [Rn{, #<im5>}]	1	0	0	0	0	Rn			Rd	im5							M[R[Rn]+sx(im5)]=R[Rd]	

Table 4: Data Processing and Data Movement Instructions

Assembly Syntax (see text)	“Simple RISC Machine” 16-bit encoding																Operation (see text)
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Branch	<i>opcode</i>			<i>op</i>		<i>cond</i>			<i>8b</i>								
B <label>	0	0	1	0	0	0	0	0	im8								PC = PC+1+sx(im8)
BEQ <label>	0	0	1	0	0	0	0	1	im8								if Z = 1 then PC = PC+1+sx(im8) else PC = PC+1
BNE <label>	0	0	1	0	0	0	1	0	im8								if Z = 0 then PC = PC+1+sx(im8) else PC = PC+1
BLT <label>	0	0	1	0	0	0	1	1	im8								if N != V then PC = PC+1+sx(im8) else PC = PC+1
BLE <label>	0	0	1	0	0	1	0	0	im8								if N!=V or Z=1 then PC = PC+1+sx(im8) else PC = PC+1
Direct Call	<i>opcode</i>			<i>op</i>		<i>Rn</i>			<i>8b</i>								
BL <label>	0	1	0	1	1	1	1	1	im8								R7=PC+1; PC=PC+1+sx(im8)
Return	<i>opcode</i>			<i>op</i>		<i>unused</i>			<i>Rd</i>	<i>unused</i>							
BX Rd	0	1	0	0	0	0	0	0	Rd	0 0 0 0 0							PC=Rd
Indirect Call	<i>opcode</i>			<i>op</i>		<i>Rn</i>			<i>Rd</i>	<i>unused</i>							
BLX Rd	0	1	0	1	0	1	1	1	Rd	0 0 0 0 0							R7=PC+1; PC=Rd
Special Instructions	<i>opcode</i>			<i>not used</i>													
HALT	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	go to halt state

Table 5: Control Instructions

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
CPEN 211 Computer Systems I, Fall 2023

Lab 6: Finite State Machine Controller for the “Simple RISC Machine”

Week of Nov 20 to 24 (your code must be submitted by 9:59 PM the evening before your lab session)

1 Introduction

In this lab you extend your datapath from Lab 5 to automate the process of executing instructions on your datapath. If you did not complete Lab 5 you can use someone else’s Lab 5 solution as a starting point for this lab, provided **both** you and the person sharing their code have received a mark for Lab 5 and both of you register the borrowing on the http://cpen211.ece.ubc.ca/cwl/student_register_peer_help.php website as outlined in the CPEN 211 Academic Integrity Policy and you **MUST** also explicitly mention this fact in CONTRIBUTIONS.txt. If you do use someone else’s Lab 5 it is recommended you use it to help you fix errors or missing functionality in your own Lab 5.

In this lab you (1) add a finite state machine controller to automate the process of setting control inputs to the datapath; and, (2) adding an instruction register to provide some inputs to your finite state machine. In the lab procedure outlined in Section 4 you implement these two additions together. To help you understand the required changes Section 2 provides an explanation for why the changes are needed. It does this by considering *preliminary* and incomplete designs. Reading the material in Section 2 will help you “understand the system” (the first rule of debugging). Next, Section 3 introduces the six instructions you need to implement for this lab. Section 4 specifies the changes you need to make in this lab. Sections 5, 6 and 7 describe the lab marking scheme, submission and demo procedures.

2 Tutorial: How to control a datapath with a finite state machine

In this section we consider a sequence of incomplete designs to see how each part we are adding in this lab helps. Do not write (System)Verilog for the designs in this section.

2.1 Controller for a single, fixed instruction (ADD R2, R5, R3)

To execute “ADD R2, R5, R3” on the datapath in Lab 5 requires four clock cycles and manually setting the control inputs. A faster way to set the control inputs is using a finite state machine. Figure 1 illustrates a **preliminary** state machine design for implementing “ADD R2, R5, R3”. The inputs to the state machine are *reset*, *s* and the clock (not shown). The outputs of the FSM are inputs to the Lab 5 datapath. In Figure 1 any datapath input not shown as an output of a given state is 0 in that state. For example, when in state *GetA* the datapath input *loadc* is assumed to be 0 even though it is not explicitly shown inside the circle representing state *GetA*. Later, in Section 4.2, you will design your own state machine. When you do, remember you need to set all outputs in every state (including zero and don’t care outputs).

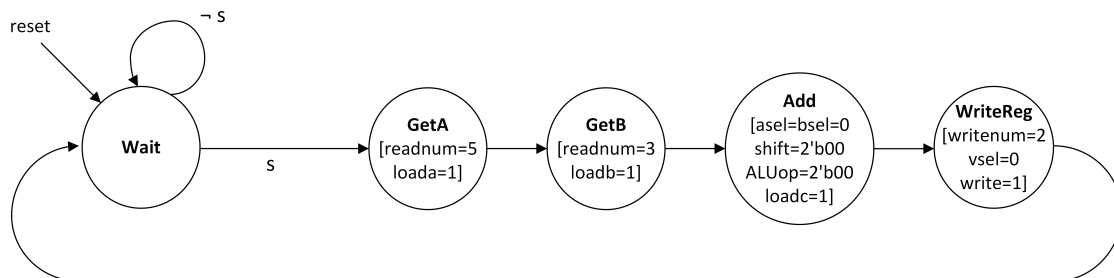


Figure 1: Finite state machine for “ADD R2, R5, R3” (do not build).

In Figure 1, after reset the state machine waits for a start signal *s* in state *Wait*. Here a value of 1 on

s indicates we should execute the sequence of four steps that implements “ADD R2, R5, R3”, where each step takes one clock cycle (e.g., one rising edge of clk).

Figure 2 illustrates the timing of a circuit combining the controller in Figure 1 with the data path from Lab 5 would behave assuming R5 initially contains 16'h13. ModelSim performs **functional simulation** in which the output of flip-flops and combinational logic occur after an infinitesimally small (but non-zero) delay so that values will appear to change right at the rising edge of clk . To understand Figure 2 it is important to remember the output of flip-flops and combinational logic in a real circuit do not actually change instantly after the rising edge of the clock in a real circuit.

In detail the behavior in Figure 2 is as follows: When in *state* *Wait* and s is 1, the rising edge of clk at the start of *Cycle n* causes *state* to change to *GetA*. The change in *state* causes *readnum* to be set to 5 (shown as “3'd5”) and *loada* is set to 1. At the next rising edge of the clock, between *Cycle n* and *Cycle n+1*, the contents of register R5 are copied to Register A and simultaneously *state* changes to *GetB*. The *state* change to *GetB* causes *readnum* to change from 5 to 3. If you were to look at this waveform in ModelSim the slight delays in the change of *state*, *readnum*, *loada*, *loadb* and A after the rising edge of clk would not be shown. The lack of such a delay in ModelSim waveforms leads some students to worry that *readnum* changing at this rising edge could cause the contents of R3 to get copied into Register A (instead of R5). Provided you have written Verilog that follows the synthesis rules taught in class, this incorrect behavior will not happen in the real circuit or your ModelSim simulations for two reasons: First, the flip-flops inside of register A start to copy their D input to their Q output at the rising edge of clk between *Cycle n* and *Cycle n+1*. Due to circuit delays, we do not see the output of register A change until some time after the rising edge but the copying started at the rising edge. Second, again reflecting how the real circuit operates the flip-flop outputs connected to *state* change slightly after the rising edge of clk and the state machine outputs *readnum*, *loada* and *loadb* change slightly after the change in *state* due to the delay of combinational logic. Thus, *loada* and *readnum* are still equal to 1 and 5 when the update of register A starts so the contents of R5 are (correctly) copied to A **before** the change in *readnum* during *Cycle n+1* can cause a problem.

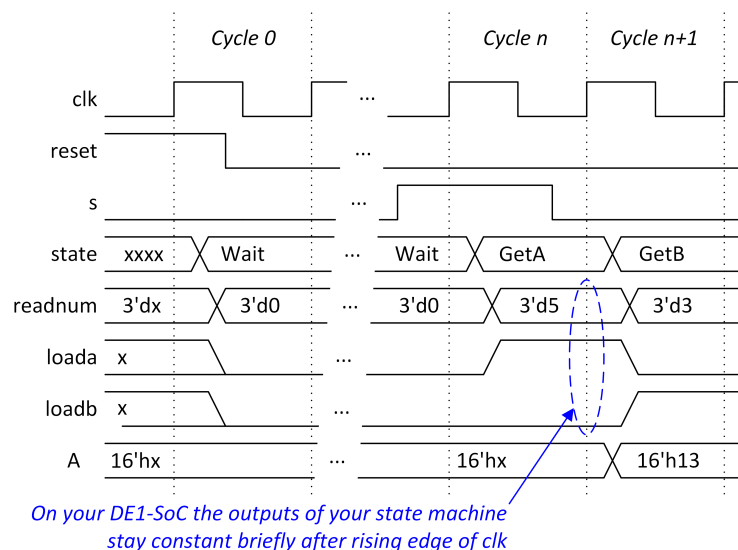


Figure 2: Register A updated on same rising edge of clk that state changes from *GetA* to *GetB*.

2.2 Controller for a more general “ADD” instruction

What if we want to be able to use different registers besides R2, R5, and R3 while performing addition? We can introduce some *programmability* by adding an *instruction register*. Figure 3 illustrates the datapath from Lab 5 combined with a state machine and instruction register. In this figure, the block labeled “Instruction

Register” contains three 3-bit fields: Rd, Rn and Rm. Each 3-bit field is used to specify the name of one of the eight registers inside the register file inside the datapath. The block labeled “FSM Controller” implements the (revised) state machine illustrated at the top of Figure 3. This state machine uses the datapath and instruction register to implement “ADD Rd, Rn, Rm”. For example, to execute “ADD R2, R5, R3” we would set Rd=3'b010, Rn=3'b101, and Rm=3'b011.

Compared with Figure 1 the output of the states for the state machine in Figure 3 have been modified. Instead of directly specifying constant values for *readnum* and *writenum* the state machine now has an output called “*nselect*”. The signal *nselect* is used as the select input of a three input multiplexer. This multiplexer selects which of the 3-bit values Rd, Rn or Rm inside “Instruction Register” should be driven to *readnum* and *writenum* (we used the same multiplexer for *readnum* and *writenum* as at any given time we are either reading or writing).

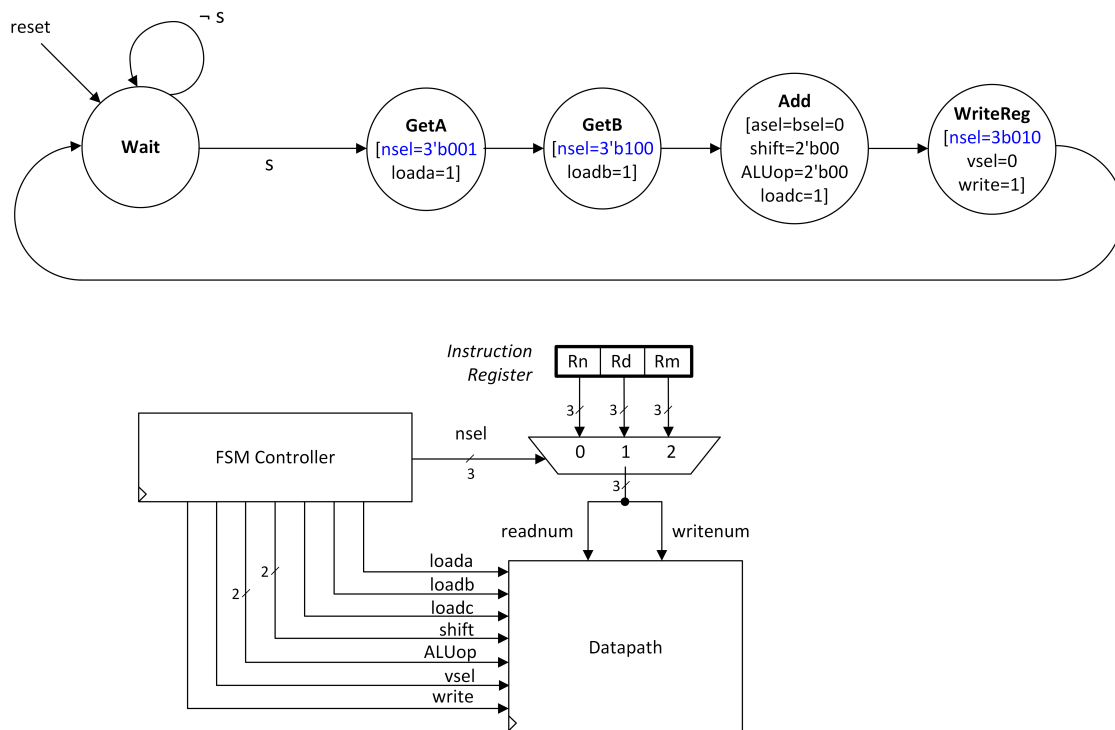


Figure 3: A partial instruction register (do NOT build).

2.3 Controller for more than one instruction

Next, we consider how to extend the design above to execute multiple types of instructions. For example, suppose we want to be able to execute “MOV R3, #42” as well as “ADD R2, R5, R3”? We can do this by extending the instruction register to include an *opcode* field indicating the type of instruction. Figure 4 illustrates a modified design where the instruction register now includes a 1-bit *opcode* field specifying whether to perform addition (*opcode*=0) or move immediate (*opcode*=1). To support the MOV instruction, the instruction register is also extended to include some bits, labeled *Immediate*, used to specify the value to copy into the register named by Rd during the MOV instruction. For example, for “MOV R3, #42” Immediate would contain the value 42 represented as a binary number. The top part of Figure 4 shows we extended our state machine to include a state *Decode* whose role is choose between performing the sequence of four steps required for an ADD instruction versus the single step required for this MOV. To enable the correct transition out of *Decode* note that we made the value of *opcode* an input to the state machine.

We also connected the 16-bit *Immediate* field of the new instruction register to the *datapath_in* portion of the datapath.

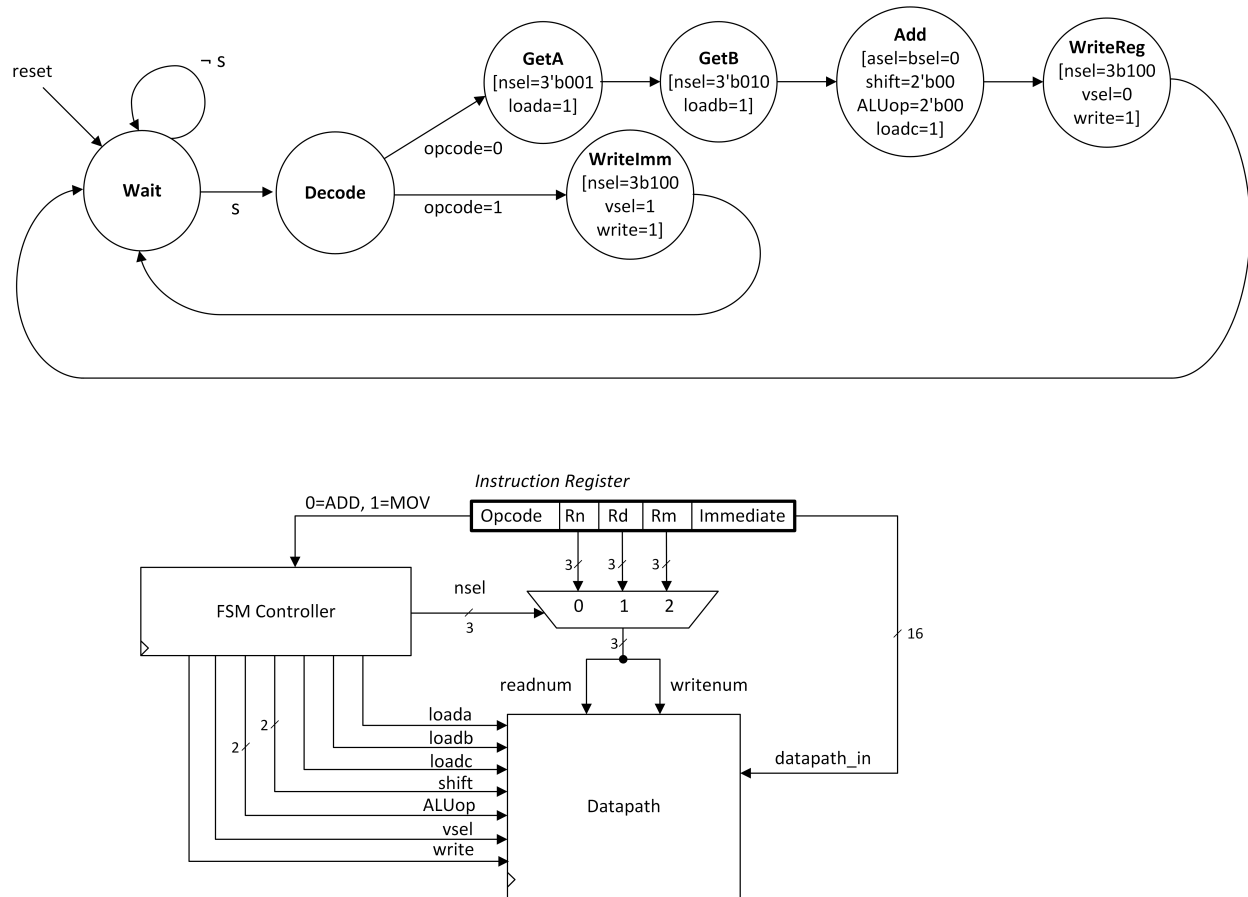


Figure 4: Supporting both MOV and ADD instructions (do NOT build).

Our design can now support execution of multiple types of instruction and the registers used by an instruction can be varied after the hardware is built. At this point the instruction to execute is *encoded* with 26-bits: One bit for the opcode, 9-bits total for the three 3-bit *register specifiers* and 16-bits for the constant value. For example, the operation “ADD R2, R5, R3” would be encoded as:

0 010 101 011 0000000000000000

Different computer *instruction set architectures* (ISAs), such as x86 and ARM, represent a given operation, such as addition, using a different encoding (pattern of 1’s and 0’s). For ARM processors, each instruction is encoded in 32-bits. For x86, different instructions may be encoded using a different number of bits (between 8 and 120 bits). For Lab 6 through 7 we provide an encoding for the Simple RISC Machine ISA in which each instruction is encoded in just 16-bits. The portion of the Simple RISC Machine ISA that you will implement in Lab 6 is shown in Table I, which is explained below. Using this encoding is required by the autograder for Lab 6.

3 The Simple RISC Machine Instruction Set Architecture

This section introduces the six instructions you will implement in Lab 6.

3.1 Assembly Syntax and Encoding

Each row of Table 1 defines an instruction of the Simple RISC Machine ISA that you will implement in Lab 6. Note the encodings in this table differ from ARM.

Table 1 is divided into three main sections. The first column, labeled “Assembly Syntax”, is a human-readable textual representation of each instruction. In assembly format each Simple RISC Machine instruction starts with an opcode mnemonic. For example, in “MOV Rn, #<im8>” the opcode mnemonic is MOV. We use this representation because it is easier to remember that MOV means “move a value from somewhere to a register” than it is to remember what “110” means. Here, “110” is the value under the heading “opcode” in the next column of Table 1. Returning to the Assembly Syntax column we see that the registers used by each instruction are also given a mnemonic representation. For example, for “MOV Rn, #<im8>” the instruction will move a value into the register Rn, where Rn can be R0, R1, ... R7. Recall these are the names of registers inside the register file that you built in Lab 5. Some instructions also include other information when written in assembly format using the Assembly Syntax. For example, in “MOV Rn, #<im8>” the part “#<im8>” represents an 8-bit binary number encoding a value between -128 and 127 (in decimal). The 8-bit value is stored inside the instruction itself. Values used by a program that are encoded inside instructions like this are typically referred to as an “immediate operand” in most instruction set architectures. In “MOV Rd, Rm{, <sh_op>}” the portion “{, <sh_op>}” is an optional shift operation (use “LSL#1”, “LSR#1”, or “ASR#1” for <sh_op>). This part of the instruction indicates what you want the shifter you built in Lab 5 to do when executing the instruction.

When programming in assembly syntax in Lab 7 you will write instructions using the notation in the first column of Table 1, but you replace Rn, Rm, and Rd with the names of specific registers between R0 and R7. Similarly, you will replace <im8> with a signed number between -128 and 127 and replace <sh_op> with one of LSL#1, LSR#1, or ASR#1.

The last column, labeled “Operation”, specifies precisely what the instruction should do using the datapath from Lab 5. For example, for “MOV Rn, #<im8>” this column contains “R[Rn] = sx(im8)”. The “sx()” part tells us that the 8-bit immediate value #<im8> should be sign-extended to 16-bits. That means we interpret the 8-bit value as a 2’s complement number and if the most significant bit of the 8-bit value, (bit 7) has the value 1, then we consider the 8-bit number to be negative. When converting the number to 16-bits, we fill in the upper 8-bits with all 1’s. Similarly, if bit 7 was 0, we would fill in the upper 8-bits with all 0’s. The portion “R[Rn] =” indicates that this sign extended value should be placed into the 16-bit register identified by Rn. Note that the steps specified under the “Operation” column can take more than one clock cycle. The notation used in this part of the table is summarized as follows:

- Rn, Rd, Rm are 3-bit register number specifiers.
- im8 is an 8-bit immediate operand encoded as part of the instruction.
- <sh_op> and sh are 2-bit immediate operands encoded as part of the instruction. The value of sh is used to control the shifter you built in Lab 5.
- sx(f) sign extends the immediate value f to 16-bits.
- sh_Rm is the value of Rm after passing through the shifter connected to the Bin input to the ALU.
- status is a 3-bit status register. The three bits are called Z, V and N which stand for zero, overflow and negative, respectively.
- f(x) is a 3-bit value, corresponding to the three bits called Z, V and N in the status register, indicating whether x is zero, whether the calculation caused an overflow, and/or whether x is a negative 2’s complement value.
- R[x] refers to the 16-bit value stored in register x.

Assembly Syntax (see text)	“Simple RISC Machine” 16-bit encoding																Operation (see text)		
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
Move Instructions	<i>opcode</i>			<i>op</i>		<i>3b</i>			<i>8b</i>										
MOV Rn, #<im8>	1	1	0	1	0	Rn			im8								R[Rn] = sx(im8)		
MOV Rd, Rm{ ,<sh_op>}	1	1	0	0	0	0	0	0	Rd	sh	Rm						R[Rd] = sh_Rm		
ALU Instructions	<i>opcode</i>			<i>ALUop</i>		<i>3b</i>			<i>3b</i>	<i>2b</i>	<i>3b</i>								
ADD Rd, Rn, Rm{ ,<sh_op>}	1	0	1	0	0	Rn			Rd	sh	Rm						R[Rd]=R[Rn]+sh_Rm		
CMP Rn, Rm{ ,<sh_op>}	1	0	1	0	1	Rn			0	0	0	sh	Rm						status=f(R[Rn]-sh_Rm)
AND Rd, Rn, Rm{ ,<sh_op>}	1	0	1	1	0	Rn			Rd	sh	Rm						R[Rd]=R[Rn]&sh_Rm		
MVN Rd, Rm{ ,<sh_op>}	1	0	1	1	1	0			0	0	Rd	sh	Rm						R[Rd]= ~sh_Rm

Table 1: Assembly instructions introduced in Lab 6 (this table is explained in Section 3)

Finally, the set of columns under the heading “Simple RISC Machine” 16-bit encoding in Table 1 specify exactly how an instruction in assembly format should be converted into the 1’s and 0’s that will be placed into the instruction register. For example, the instruction “ADD R4, R0, R1” is encoded as the 16-bit value:

101 00 000 100 00 001

Bits 15, 14, and 13 of each instruction is a special “operation code” or “opcode” that identifies the basic operation the instruction performs. Thus, the first three bits above, 101, represents the “opcode” used for the instruction “ADD R4, R0, R1”. The opcode specifies the type of operation performed by the instruction. Instructions with opcode 101 are defined in the Simple RISC Machine ISA to be “ALU instructions”. ALU instructions read two registers perform an operation on the values in these registers using the ALU and then write the result back to a register in the register file. For such instructions the next two bits, Bit 12 and Bit 11, specify the ALUop input to the ALU. In this example these bits have the value 00, which corresponds to the addition operation for the ALU you designed in Lab 5. The next three bits indicate the middle register (called Rn). In our example, this is R0, so the next three bits are 000. The next three bits encode the destination register that will be written by the instruction. In our example, this is R4. For ALU instructions the next two bits are the “shift” input to the shifter in your Lab 5 datapath. The final three bits specify the other register that is read by the instruction. In this example, that is R1, which is encoded as 001.

Rn, Rd, Rm are 3-bit numbers that refer to one of the eight 16-bit registers inside of the register file. in the first column “{, <sh_op>}” is an optional shift operation (use “LSL#1”, “LSR#1”, or “ASR#1” for <sh_op>); sx() means sign extend (described below); sh_Rm is the 16-bit value resulting from shifting Rm using the code “sh” (bits 3:4) as input to the shifter from Lab 5.

3.2 Instruction Descriptions

Next, we briefly summarize the operation of each instruction. The first instruction in Table 1, “MOV Rn, #<im8>”, takes bits 0 to 7 of the instruction (labeled “im8”) and “sign extends” these bits to a 16-bit value. Recall that in 2’s complement the most significant bit is a 1 if the number is negative and it is 0 if the number is positive. We can take an 8-bit positive number (with bit 7 equal to zero) and make a 16-bit positive number with the same value by simply concatenating 8 bits that are all zero. Similarly, we can take an 8-bit negative number (with bit 7 equal to 1) and make a 16-bit negative number with the same value by concatenating 8-bits with all 1’s. E.g., consider sign extending the number 3 from 8-bits to 16-bits:

8-bit representation of 3 16-bit representation of 3
00000011 0000000000000011

Similarly, consider sign extending -3 from 8-bits to 16-bits:

8-bit representation of -3 16-bit representation of -3

11111101

1111111111111101

After performing this sign extension, the MOV instruction writes the resulting 16-bit sign value to one of the eight 16-bit registers inside the register file. It identifies which of the 8 16-bit registers inside the register file to write using the 3-bit 8 to 10 of the instruction (labeled Rn). Recall with 3-bits we can uniquely identify 8 things since $2^3 = 8$. We return to discuss the second version of MOV further below.

The second MOV instruction, “MOV Rd, Rm{, <sh_op>}” reads Rm into datapath register B and then sets asel=1 to select the 16-bit zero input for the Ain input to the ALU. Since ALUop is “00” the ALU adds the zero on Ain to the shifted value of Rm on Bin and places the result in register C. The result is then written to Rd.

The next four instructions in Table 1 are called ALU instructions because their main purpose is to use the ALU you built in Lab 5. Such instructions are the main “workhorses” of any general-purpose computer design. The ADD Rd, Rn, Rm{, <sh_op>} instruction reads the contents of register Rm, optionally shifts the value one bit to the left (for example, “ADD Rd, Rn, Rm, LSL#1”), one bit to the right without sign extension (“ADD Rd, Rn, Rm, LSR#1”) or with sign extension (“ADD Rd, Rn, Rm, ASR#1”). Then adds the result to Rn and places the sum in Rd. For example, if R0 contains 25 and R1 contains 50, then after executing the instruction “ADD R2, R1, R0” the contents of R2 would be 75.

The ADD instruction reads register Rn into register A and reads register Rm into register B. Bits 11 to 12 of the instruction register are directly fed to the ALUop input to the ALU. Since these bits are “00” for ADD instructions ALUop will be “00” which corresponds to addition. So, the Ain and Bin inputs to the ALU will be added together by the ALU. The operand in register B is shifted as specified by bits 3 and 4 that are fed directly from the instruction register into the “shift” input of your datapath from Lab 5.

The AND instruction is very similar to ADD. However, both the CMP and MVN instructions, while using the ALU, are different. CMP is the only instruction that should update the three status bits. For CMP we use the ALUop for subtraction however, we are only interested in the value of the status outputs of the ALU. E.g., we can use CMP to check if the value in R1 and R2 are equal by subtracting R2 from R1 and checking if the result is zero using the Z status flag. As with ADD and AND we can shift the contents of the B register. In Lab 7 we will add branch instructions that read the status register after it is set by a CMP instruction. For MVN we perform a bitwise NOT on the contents of Rm. As with the other ALU operations we can shift the value in the B register.

4 Lab Procedure

The changes for this lab are broken into two stages.

4.1 Stage 1: Datapath Modifications

Extend the mux on the data input to your register file to have the four inputs illustrated on the right in Figure 5. The sximm8 input (which stands for sign extended 8-bit immediate) will eventually be driven by the Instruction Decoder you add in Stage 2 below. You will use this input to the mux when implementing control logic for the “MOV Rn, #<im8>” instruction in Table 1. Here mdata is the 16-bit output of a memory block you will be adding in Lab 7. Next, sximm8 is a 16-bit sign extended version of the 8-bit value in the lower 8-bits of the instruction register. Next, PC is an 8-bit “program counter” input that will be explained and used in the Lab 7 Bonus. However, to avoid introducing bugs later, it is recommended you add a 16-bit mdata and 8-bit PC inputs to your datapath module in Lab 6 and connect them to the 4-input 16-bit multiplexer as shown in Lab 6. For Lab 6 you can “assign” zero to mdata and PC.

Next, modify the mux input to Bin as shown in Figure 6. Here sximm5 is a 16-bit variable you should declare in datapath. Here sximm5 stands for “sign extended 5-bit immediate”. We will connect sximm5 to another block in Stage 2.

Next, extend the status register to three bits. One bit should represent a “zero flag”, which was what “status” represented in Lab 5. Another bit should represent a “negative flag” and be set to 1'b1 if the most

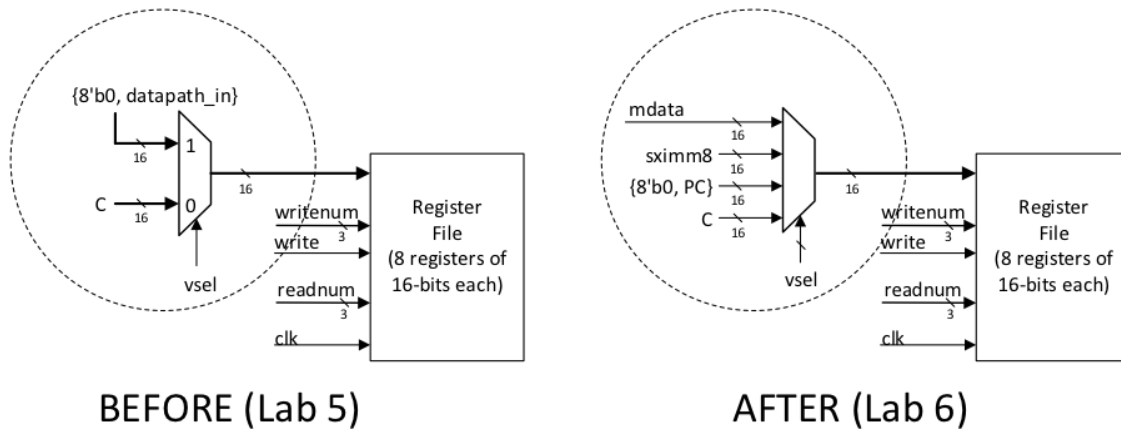


Figure 5: Modification to Lab 5 Datapath: Input to Register File

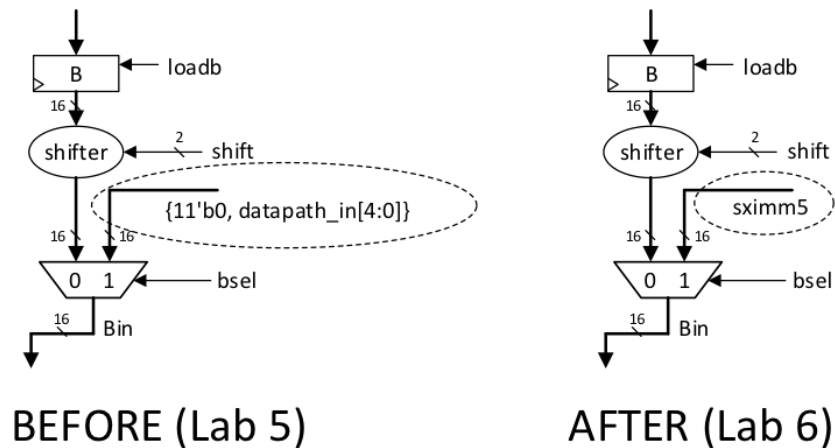


Figure 6: Modification to Lab 5 Datapath: “Bin” Multiplexer

significant bit of the main 16-bit ALU result is 1. The final bit represents an overflow flag. You should compute signed overflow as described in Section 10.3 of Dally. In Lab 7 you will use the status flags to support “if” statements and “loops” in C.

4.2 Stage 2: Datapath Controller

The next step is to add an instruction register, an instruction decoder block, and finally design a state machine to control your datapath. Inside a file `cpu.sv` create a module `cpu` to instantiate and connect together these three components along with your datapath. A significant portion of your Lab 6 mark will be determined using an auto grader (see Section 5). To avoid losing marks your top level module must be called `cpu`, follow the specification below, and be in a file named `cpu.sv`:

```

module cpu(clk,reset,s,load,in,out,N,V,Z,w);
  input clk, reset, s, load;
  input [15:0] in;
  output [15:0] out;
  output N, V, Z, w;

```

Figure 7 illustrates how the various components of your design should be connected within your `cpu` module. Your state machine is connected to the rest of the circuit. The instruction currently being executed

is stored in the 16-bit Instruction Register. The instruction register and *nsel* output of the state machine are inputs to the Instruction Decoder block, which is described below.

The value on *in* should be copied into your instruction register on the rising edge of *clk* if *load* is set to 1. If *load* is 0, the contents of your instruction register should NOT change. On the rising edge of *clk* if *reset* is 1 your state machine should go to a reset state. After being reset, your state machine should not perform any computations until *s* is set to 1 and there is a rising edge of *clk*, much like in the example state machine in Figure 4. The value on *out* should be the contents of register C inside your datapath, which is useful for the demo on your DE1-SoC (the autograder will not check the value of “out” since it can directly inspect the contents of the register file). The outputs *N*, *V* and *Z* should provide the value of the negative, overflow and zero status register bits. As shown in Table 1, these three flags are set only by the CMP instruction. The *N* flag should be set to 1 if the 16-bit result of the subtraction performed by the CMP instruction is negative, regardless of whether an overflow occurred. The output *w* should be set to 1 if your state machine is in the reset state and is *waiting* for *s* to be set to 1 and set to 0 otherwise (i.e., when in any other state). After executing an instruction, your state machine should return to this state. See the example state machine in Figure 4 (NOTE: do *not* use the instruction register design from Figure 4 as it does not correspond to the required instruction encodings in Table 1).

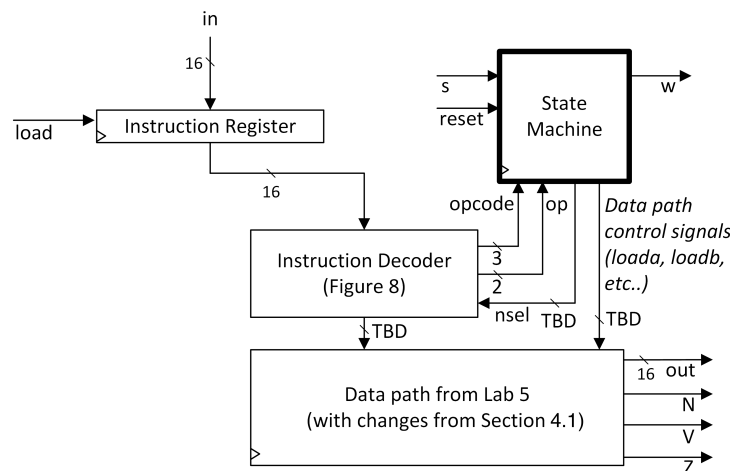


Figure 7: Your *cpu* module should contain your finite state machine controller (Controller), an instruction register, an instruction decoder and a datapath. In this figure, TBD means “to be determined” (by you!).

The purpose of the Instruction Decoder block is to extract information from the instruction register that can be used to help control the datapath. The Instruction Decoder block in Figure 7 should implement the logic shown in Figure 8. Your state machine should drive any datapath inputs not set by the decoder block (e.g., labeled “TBD” in Figure 7).

The output of the state machine should be the settings of all the inputs to the datapath, the signal *nsel* used to select which register to connect to *readnum* and *writenum*, and the *w* output used to indicate to the autograder that your state machine is (or is not) in the wait state. The inputs to the state machine are *clk*, *reset*, the start signal *s* the *opcode* and *op* fields of the current instruction in the instruction register.

How should you design your state machine? The examples shown in Figure 3 and 4 used a Moore type finite state machine where the output depends only on the current state. You can do the same or, if you want, use a Mealy state machine.

Regardless of which type of state machine or the coding style you use, the best way to *design* your state machine is in stages. In the first step, get your state machine to work for a single instruction from Table 1. Pick an instruction from Table 1 and think through what steps you need to perform with your datapath from

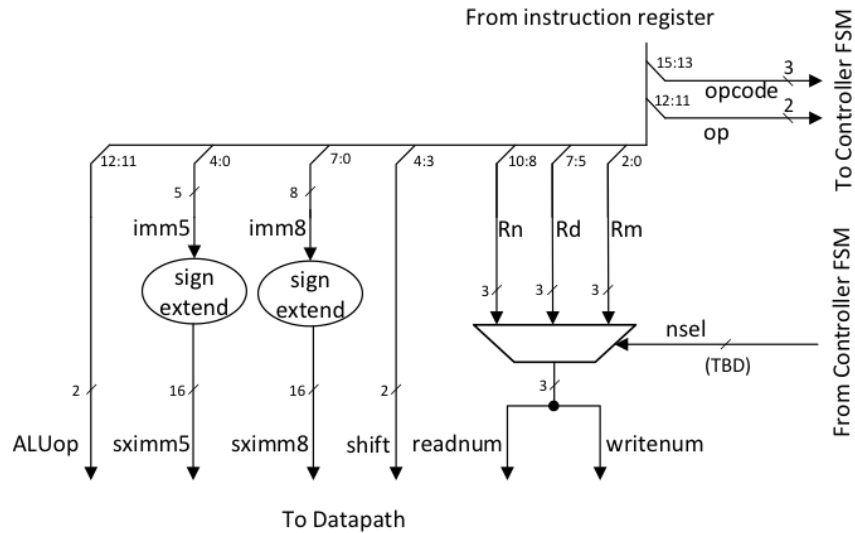


Figure 8: Instruction Decoder

Lab 5 to perform the steps listed in the “Operation” column. One way to do this is by referring to Figure 1 in the Lab 5 handout. Work out the number of clock cycles it takes to perform the data operations required for the instruction. Each cycle will require an additional state beyond the “Wait” state shown in the example in Figure 3. After the last state required to execute the instruction on your datapath, your state machine should return to the “Wait” state as shown in the example in Figure 3. In the “Wait” state make sure your **w** output is 1 so the autograder (or your own test bench) knows the computer has finished executing the instruction and is ready to execute a new instruction.

Once you have very carefully tested your first instruction is working, both in ModelSim and on your DE1-SoC, you should check that version into your revision control system (e.g., git) in case you make a change that breaks that first instruction while modifying your state machine to support a second instruction. When adding the second instruction, you should add a “Decode” state like that shown in Figure 4. You should reuse this Decode state when adding any subsequent instructions. After adding the Decode state, figure out the states corresponding to the steps required to execute the new instruction on your datapath. You will add additional states (e.g., like WriteImm in Figure 4) for this new instruction. Now the “Decode” state has two potential next states. To decide which next state your state machine should go to from “Decode” use the opcode and op values used for encoding the instruction you are adding (find these in Table 1). See Figure 4 for a simplified example showing how to determine which state to go to after “Decode”.

After adding each instruction test that *both* the new instruction and the prior instructions work (both in ModelSim and on your DE1-SoC). Unless you are VERY confident in your Verilog coding abilities, you should NOT attempt to code up a state machine for all instructions before doing any testing. If you do, you will spend much more time trying to figure out the source of even a single bug than you would have by testing each additional instruction as you add it. Since you need to show a testbench as part of the marking scheme, why not create it as you go and use it to help you save time by catching bugs early?

To reduce the complexity of your state machine you may want to see if you can find ways to reuse states added for earlier instructions when adding a new instruction. However, this is not required.

The input to your top level module is the encoded instruction. Thus, to test your overall design you will first want to create some simple programs that you can input to your instruction register one at a time. To do that, first write a textual assembly code representation and only then encode each instruction into 1’s and 0’s using Table 1. As an example, the following test case.

```
MOV R0, #7 // this means, take the absolute number 7 and store it in R0
MOV R1, #2 // this means, take the absolute number 2 and store it in R1
ADD R2, R1, R0, LSL#1 // this means R2 = R1 + (R0 shifted left by 1) = 2+14=16
```

can be encoded as:

```
1101000000000111
1101000100000010
1010000101001000
```

For full marks on the lab you need to encode additional instructions. You can use the `lab6_top.v` file we provide to test your design on the DE1-SoC. This is a modified version of `lab5_top.v` and works in a similar way.

5 Marking Scheme

If you have a partner *both* of you must be in attendance during the demo. You **must** include at least one line of comments per always block, assign statement or module instantiation and in test benches you must include one line of comments per test case saying what the test is for and what the expected outcome is. For your state machine include one comment per state summarizing the datapath operations and one comment per state transition explaining when the transition occurs.

You will lose marks if your github repo does not contain a quartus project file, modelsim project file, or programming (.sof) file. You will also lose marks if your repo is missing any source code (whether synthesizable or testbench code) or waveform format files.

If you used someone else's Lab 5 code you are still responsible for being able to explain how it works. If your submission includes AI generated code you must include an AI.txt file with enough detail to reproduce code that looks like your submission in case of later concern about the true provenance of your code (i.e., a suspicion of cheating).

Your mark will be computed as the sum of the following rubric:

Stage 1 changes [1 Mark] For your Stage 1 code in `datapath.v` and being able to explain the associated (System)Verilog to the TA. You may also lose marks here for lack of commenting in your code or if you have no testbench in `datapath_tb.v` for your Stage 1 changes.

Stage 2 changes [3 Marks] Your state machine must include sufficient comments. During your marking session you must be able to explain your code in detail when asked, and demonstrate that your state machine works using your submitted testbench with ModelSim and your submitted `lab6_wave.do` file. Your mark for this part will be:

3/3 If your state machine in `cpu.v` implements all instructions in Table 1 and you demonstrate a detailed knowledge of how your state machine works when asked by your TA, your submission contains a set of very convincing test cases **of your own devising** in `cpu_tb.v` including at least three tests for each instruction in Table 1. Each test should be designed to test a different part of your design that might have an error and you should be able to explain to your TA what potential error or mistake in coding that design is meant to catch. It is your responsibility to think of what might go wrong when coding your state machine and combining it with your datapath and how the tests might catch those errors. For full marks here you should be able to demonstrate some test cases using gate-level simulation (possibly using a second testbench in `cpu_tb.v`).

2/3 If your state machine implements all the instructions from Table 1, but you have less than three tests for each instruction in your `cpu_tb.v` or you cannot provide examples of the types of bugs that the tests might catch.

1/3 If your state machine does not implement all of the instructions in Table 1 or your state machine includes very few comments or comments that are not very meaningful, or you do not include a testbench.

0/3 If you cannot explain your code or there is no code submitted for this part.

DE1-SoC Demo [2 Marks] For demonstrating your CPU works on your DE1-SoC using a test case of your own devising involving some of the LEDs on the DE1-SoC. To get 2/2 marks here this test case **MUST** work **AND** use **ALL** of the instructions in Table 1. You will get 1/2 here if this test case works and it uses at least three different types of instructions from Table 1 (but not all of them). If you cannot get a test case involving at least three different types of instructions from Table 1 your mark will be 0/2.

Autograder [4 Marks] Finally, four marks will be assigned objectively by an auto-grader that will test your cpu module using a variety of inputs. To ensure your code is compatible with our autograder you should both ensure you can download your working design to your DE1-SoC **AND** be sure you can simulate the lab6_check testbench module we provide in lab6_autograder_check.v provided on Piazza. You should be sure you get the message “INTERFACE OK” in the ModelSim transcript window when you do this. Please note that the message “INTERFACE OK” does **NOT** ensure your Lab 6 submission will pass any of our autograder tests, but if you do not get this “INTERFACE OK” message you will get 0/4 for this part. **Your autograder mark may be manually reduced to 0/4 for this portion if you cannot explain to your TA how your code works to their satisfaction.**

Your cpu module must follow the specification in Section 4.2 and moreover, the output of the registers inside your register file must be accessible via the hierarchical names `cpu.DP.REGFILE.R0` through names `cpu.DP.REGFILE.R7`. Thus, your datapath must be instantiated with the instance name DP inside your cpu module and inside of your datapath module your register file must have the instance name REGFILE, inside of your register file, the 16-bit registers R0 through R7 must be accessible on signals (wire or reg) called R0 through R7. To ensure this you may need to make minor changes to your datapath from Lab 5. Your mark for this part will depend upon how many instructions pass our test cases and will be:

4/4 If every single *type* of instruction in Table 1 passes *all* of the auto-graders test cases.

3/4 If all but one *type* of instruction in Table 1 passes *all* of the auto-graders test cases. This means for example, if you did not have time to get **one** of the instructions in Table 1 working, but you got all the other instructions working (as judged by our autograder), then you would get this mark.

2/4 If all but two *types* of instruction in Table 1 passes *all* of the auto-graders test cases. This means for example, if you did not have time to get **two** of the instructions in Table 1 working, but you got all the other instructions working (as judged by our autograder), then you would get this mark.

1/4 If all but three *types* of instruction in Table 1 passes *all* of the auto-graders test cases. This means for example, if you did not have time to get **three** of the instructions in Table 1 working, but you got all the other instructions working (as judged by our autograder), then you would get this mark.

0/4 If four or more *types* of instruction in Table 1 each fail at least *one* of the auto-graders test cases. This means for example, if you did not have time to get **four** of the instructions in Table 1 working, but you got all the other instructions working (as judged by our autograder), then you would get this mark.

IMPORTANT: Check your submission repo on github.com carefully as you will lose marks if your github submission does not contain a Quartus Project File (.qpf) and the associated Quartus Settings File

(.qsf) that indicates which Verilog files are part of your project. This .qsf file is created by Quartus when you create a project. It is typically named `<top_level_module_name>.qsf` and contains (among others) lines indicating which (System)Verilog files are to be synthesized. If you open up this .qsf file you should see lines that look like the following. The key part is that these line contain “VERILOG_FILE”. The autograder will use your .qsf file to determine which Verilog files should be synthesized together. To be sure, note the above .qsf file is **not** the file `DE1_SoC.qsf` we provided in Lab 3 for importing DE1-SoC pin assignments. Also remember to include your Modelsim Project File (.mpf) and your programming (.sof) file, (.vo) file for gate-level simulation testing, and any waveform (.do) files.

6 Lab Submission

If you work with a partner or you borrowed someone’s Lab 5 code (even if only to debug your Lab 5 code) your submission **MUST** include a file called “CONTRIBUTIONS.txt” that describes each student’s contributions to each file that was added or modified. If either partner contributed less than one third to the solution (e.g., in lines of code), you must state this in your CONTRIBUTIONS file and inform the instructor by sending email to <mailto:aamodt@ece.ubc.ca>. Note that submitted files may be stored on servers outside of Canada. Thus, you may omit personal information (e.g., your name, SN) from your files and refer to “Partner 1” and “Partner 2” in CONTRIBUTIONS. Submit your code using github classroom.

If you used any AI tools to help write code you must include a file AI.txt describing their use in sufficient detail that we can reproduce your code (or something very similar) in case of any concern about the provenance of your submitted code (e.g., if your code looks similar to other student submissions). A reminder that sharing of AI prompts is not permitted.

7 Lab Demonstration Procedure

As with Lab 3 to 5, your TA will have your submitted code with them and have setup a “TA marking station” where you will go when it is your turn to be marked. Be sure to bring your DE1-SoC in case the TA does not have theirs and/or they need to mark multiple groups in parallel.

8 Hints and Tips

You may find it helpful to use the Verilog include directive to define constants used in multiple files.

After getting your Verilog to compile in ModelSim, but before running any simulations in ModelSim, it is worth try to compile your synthesizable modules in Quartus just to look at the warnings. Quartus provides useful warnings for many “silly mistakes” that ModelSim happily ignores. If you see no suspicious warnings in Quartus, then move on to simulating your testbench in ModelSim.

When using “\$stop;” in a testbench and running with “run -all” in ModelSim a source window will pop-up when “\$stop” is reached. If you use a text editor other than ModelSim to edit your files (e.g., vi or emacs), make sure to close this window before restarting simulation. If you for any reason modify your testbench outside of ModelSim (perhaps to add a test case) and you then restart simulation you will get a long set of about 50 pop-ups saying the file was modified outside of ModelSim. If you forget and this happens, note you can likely close these roughly 50 dialogs faster then restarting ModelSim by clicking on “Skip Messages” then selecting “Reload” repeatedly.

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
CPEN 211 Computer Systems I, Fall 2023

Lab 7: Adding Memory and I/O to the “Simple RISC Machine”

Week of Nov 27 to Dec 1 (your code must be submitted by 9:59 PM the evening before your lab session)

1 Introduction

In this lab you extend the datapath and finite-state machine controller from Lab 6 to include a memory to hold instructions. Then, we will add two instructions so that we can use this same memory to hold data. Finally, we extend the interface to memory to enable communication with the outside world using memory mapped I/O. If you did not complete Lab 6 you can use someone else’s Lab 6 solution as a starting point for this lab, provided **both** you and the person sharing their code have received a mark for Lab 6 already and you register the borrowing on the http://cpen211.ece.ubc.ca/cwl/student_register_peer_help.php website as outlined in the CPEN 211 Academic Integrity Policy. You should receive an email confirmation after you submit this form and you should keep that email for your records. If you use someone else’s Lab 6 code as a starting point for Lab 7 make sure you also note this in your CONTRIBUTIONS file. Details of the CPU design competition will be posted in a separate “Lab 7 Bonus and Competition” handout.

2 Tutorial

In Lab 5, you had to manually control each element of your datapath using slider switches on your DE1-SoC. In Lab 6, you added a finite-state machine that did this for you, but you still had to use the slider switches to enter encoded instructions as input to your state machine. In this lab, you add a read-write memory to hold instructions so you no longer need to enter each instruction using the slider switches. With some minor changes we can also use the same memory to hold data. This section outlines the changes at a conceptual level and Section 3 describes the detailed changes required.

2.1 Instruction Memory

Figure 1 illustrates a *simplified* view of how one might interface memory containing instructions to the components from Lab 6. Each memory location holds 16-bits versus 8-bits for ARMv7. Each Simple RISC Machine instruction is also 16-bits long and thus occupies one memory location. A program counter (PC) is connected to the address input of the memory. The PC contains the address of the next instruction to execute and is implemented with a register with load enable. The memory contains 256 memory locations thus the program counter is $\log_2(256) = 8$ -bits wide. Note that, unlike in ARMv7, the PC in the Simple RISC Machine architecture is not a register inside the register file. The PC is updated when the load enable

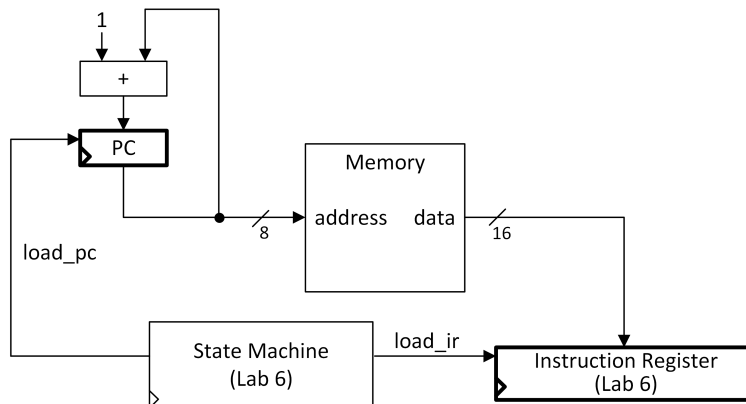


Figure 1: Adding an instruction memory and program counter (simplified).

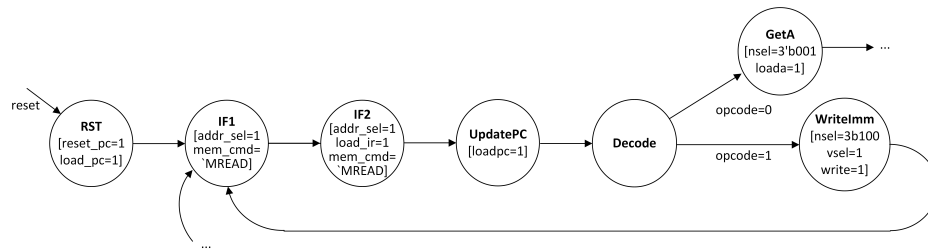


Figure 2: Modified State Machine to Interface with Instruction Memory

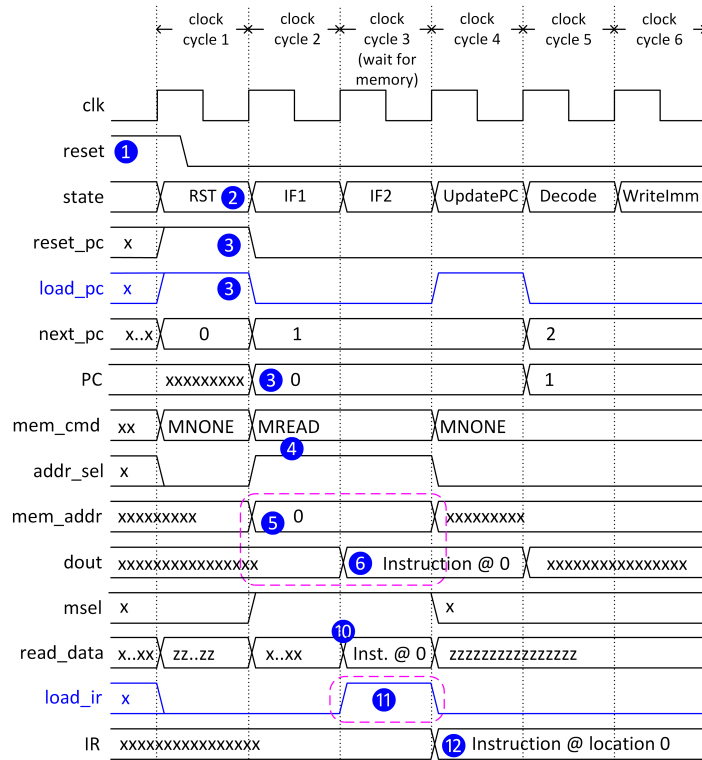


Figure 3: Timing of Instruction Read and PC Update (note one cycle delay on dout).

input, `load_pc`, is set to 1 by the state machine. In Figure 1 the PC is updated by adding 1 to compute the address of the next instruction. The bonus adds branches enabling the PC to change by other values.

The 16-bit value in the memory location at the address identified by the PC is the next instruction to execute, encoded in binary. However, to execute this instruction it must first be copied into the instruction register. This is accomplished by setting `load_ir` to 1.

Figure 2 illustrates how one might change the finite state machine from Figure 4 in the Lab 6 handout to use `load_pc` and `load_ir`. The four states *Reset*, *IF1*, *IF2* and *UpdatePC* replace the *Wait* state from Lab 6. Figure 3 illustrates the use of this state machine to read an instruction from memory into the instruction register. The other signals in these figures (e.g., `reset_pc`, `addr_sel`, `m_cmd`, ...) are described later.

Briefly, in Figure 3 on Cycle 1 the *Reset* state resets to PC to zero (the datapath circuitry for setting PC to zero is shown later). In state *IF1* (Cycle 2) the address stored in the program counter is sent to the address input of the memory (`mem_addr`). The memory is implemented using a RAM block in the Cyclone V. This RAM block has synchronous reads meaning the contents of memory do not appear on the memory's data output (`dout`) until after the *next* rising edge of the clock (during Cycle 3). Because of this “delay”, we

cannot load the instruction register while in State *IF1*. The 16-bit encoded instruction becomes available on *dout* during Cycle 3 while in State *IF2*. In State *IF2* the state machine sets *load_ir* to 1 so that on the *next* rising edge of the clock the instruction is copied from *dout* into the instruction register. In state *LoadPC* we set *load_pc* to update the program counter to the address of the next instruction to execute.

2.2 Using Memory for Data

Besides storing instructions in memory, it is also helpful to use memory for holding data. For example, consider the following line of C code:

```
g = h + A[0];
```

If “g” is in R1, “h” is in R2 and the starting address of array A is in R3, we can implement the above C code using the two instructions:

```
LDR R5, [R3]      // temporary register R5 gets A[0]
ADD R1, R2, R5     // g = h + A[0]
```

Here the instruction “LDR R5, [R3]” is called a *load* instruction. This load instruction first reads the value in R3 then reads the location in memory at this address. For example, if R3 happened to have the value 4 the load instruction would read the 16-bit value in memory at address 4 and copy it into register R5.

What if we want to change the value in memory? Consider the following line of C code:

```
A[0] = g;
```

This can be implemented using the following *store* instruction:

```
STR R1, [R3]
```

This instruction first reads the registers R3 and R1, then it writes a copy of the contents of R1 into the memory location with the address in R3. For example, if R1 had the value 55 and R3 had the value 4, then after the store instruction the contents of memory at address 4 would be equal to 55.

3 Lab Procedure

The changes for this lab are broken into three stages. In Stage 1 you add support for fetching instructions from the memory and executing them using the state machine and datapath from Lab 6. In Stage 2 you add support for using the memory to store data as well as instructions. This requires adding support for two new instructions (1) a load instruction (LDR) and (2) a store instruction (STR). Finally, in Stage 3 you add support for allowing the load and store instructions to access the switches and red LEDs on your DE1-SoC.

3.1 Stage 1: Executing Instructions from Memory

In Stage 1 you will implement the hardware shown in Figure 4. As indicated in the figure, some additions should go into your `cpu` module from Lab 6 and others in a new `lab7_top` module for Lab 7 that should follow the declaration:

```
module lab7_top(KEY,SW,LEDR,HEX0,HEX1,HEX2,HEX3,HEX4,HEX5);
  input [3:0] KEY;
  input [9:0] SW;
  output [9:0] LEDR;
  output [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
```

Below we first describe how to add and initialize the memory then describe how the rest of the design interfaces with that memory.

3.1.1 Implementing and Initializing Memory

To implement the memory include the Verilog module “RAM” from Slide Set 11 and instantiate it in `lab7_top` with instance name MEM. The memory is initialized with instructions contained in a file `data.txt`. The

format of each line in `data.txt` is “@<addr> <contents>”. Here <addr> is a memory address in hexadecimal and <contents> is the data to load into the corresponding memory location, written in binary. For example, the instruction sequence:

```
MOV R0, #7 // this means, take the absolute number 7 and store it in R0
MOV R1, #2 // this means, take the absolute number 2 and store it in R1
ADD R2, R1, R0, LSL#1 // this means R2 = R1 + (R0 shifted left by 1) = 2+14=16
```

can be loaded into `lab7_top.MEM` by putting the following text in “`data.txt`” in your project folder:

```
@00 1101000000000111
@01 1101000100000010
@02 1010000101001000
```

Here the instruction “`MOV R0, #7`” was encoded as “1101000000000111” using the encoding in Table 1 of the Lab 6 handout and is placed in memory at address 0. Since the memory actually contains 256 locations and we are only specifying the contents of three ModelSim will leave the contents of the remaining 253 locations undefined (i.e., x’s). Since leaving values undefined is not an option for synthesis Quartus will warn us that the “number of words (3) in memory file does not match the number of elements in the address range [0:255]” and will initialize the other 253 memory locations to contain all zeros.

You can manually create `data.txt` for shorter test cases. For longer programs an assembler, called `sas`, is provided. You can either build `sas` using the source code provided in the starter repo under the “assembler” folder and run it on your computer or you can use the version installed on `ssh.ece.ubc.ca`. To use `sas` you write a text file <file>.s (e.g., `data.s`) containing assembly (e.g., lines such as “`MOV R0, #7`”) then on a computer where `sas` is installed run `sas` using <file>.s as an input. For example, to use the version of `sas` installed on the ECE computers: (1) transfer <file>.s to your ECE account; (2) log into `ssh.ece.ubc.ca`; (3) at the command prompt on `ssh.ece.ubc.ca` type:

```
~cpen211/bin/sas <file>.s
```

where <file>.s is the assembly file you want to compile.

The assembler generates two output files. The file <file>.lst shows both the encoded instructions and the human readable assembly. The file <file>.txt is a file you can rename to `data.txt` in your project directory to initialize your memory.

You *must* place “`data.txt`” (the memory initialization file containing your test program) in the working directory used by ModelSim and Quartus. In ModelSim this is the project directory you specified when creating the project and in Quartus this is the directory you specified in answer to the prompt “What is the working directory for this project?” in the New Project Wizard. If you setup your ModelSim and Quartus projects properly, these should be the same directory so that there is no need to copy `data.txt` from one directory to another. In ModelSim, type `pwd` in the transcript window to verify which directory ModelSim will search to find your `data.txt`. If you forgot to set the ModelSim project directory then in the transcript window you can type `cd <path>`, replacing <path> with the directory you placed your `data.txt` file. If you encounter compilation errors in Quartus related to not finding “`data.txt`” create a new project and remember to set the working directory. Do **not** specify a relative or full path to “`data.txt`” in your Verilog files as this will cause the autograder to fail.

3.1.2 Executing Instructions Stored in Memory

After adding MEM in your `lab7_top` add a PC and the multiplexers and tri-state drivers shown in Figure 4. A 16-bit tri-state driver can be implemented using the following Verilog:

```
assign out = enable ? in : {16{1'bz}};
```

where “in” is a 16-bit reg or wire that is connected to 16-bit wire “out” using 16 tri-state drivers all controlled by 1-bit reg or wire “enable”. When “enable” is 1 all 16 tri-state drivers are enabled—i.e.,

bits which corresponds to address from 0 to 511 (decimal) or 0x000 to 0x1FF (hexadecimal). In Stage 3 we will use addresses 0x100 through 0x1FF for accessing input/output devices. Thus, if the Simple RISC Machine tries to read or write to an address from 0x100 to 0x1FF we do not want the memory to respond. To achieve this objective we control the enable input of the tri-state driver with two comparators and an AND gate as described below.

To control access by the memory to *read_mem* a first equality comparator (8 in Figure 4) determines whether the high-order bits correspond to the address range the memory contains. Specifically, the address corresponds to the memory if bit-8 is 0. The second equality comparator (9 in Figure 4) determines whether the operation is a read command. The outputs of the comparators are AND-ed together to determine whether the data read from memory should be driven to the data bus via a tri-state driver (7 in Figure 4). The enable input to the tri-state drive is true during Cycle 2 and 3 in Figure 3, thus causing whatever value is on *dout* to be driven to *read_data* (10 in Figure 3 and 4).

Finally, as the instruction bit are available on Cycle 3 the *load_ir* signal becomes high that cycle (11 in Figure 3) which causes them to be loaded into the instruction register on Cycle 4 (12 in Figure 3).

Once you understand the above, update your state machine and test that you can execute a program with a few instructions. If you find yourself having trouble getting instructions into the instruction register via the tri-state driver you may want to first directly connect *dout* to the input of the instruction register to get that working and only then connect the tri-state driver logic.

3.2 Stage 2: Putting Data in Memory

In this stage we add support for load and store instructions. To do this first add the Data Address register, *write_data* bus, and associated control logic for the memory write input shown in Figure 5.

Next, modify your state machine to add support for the load and store instructions in Table 1. In the column “Operation” the notation “M[x]” refers to either reading or writing the 16-bit memory location with address x. The Lab 5 handout has a summary of the other notation used in this table. The finite state machine for your LDR instruction should cause the data path to read the contents of the register Rn inside your register file, add the sign extended 5-bit immediate value encoded in the lower 5-bits of the instruction, then store the lower 9-bits of “R[Rn]+sx(im5)” in the Data Address register in Figure 5. Next, your state machine will need to set *addr_sel* to 0 so that *mem_addr* is given by this address. Your state machine will also need to set *mem_cmd* to indicate a memory read operation. The value read from memory should be saved in R[Rd].

The STR instruction should also start by reading the contents of the register Rn inside your register file, add the sign extended 5-bit immediate value encoded in the lower 5-bits of the instruction, then store the lower 9-bits of “R[Rn]+sx(im5)” in the Data Address register in Figure 5. Then, the contents of R[Rd] should be read from the register file and output to *datapath_out* which should be connected to *write_data*. Finally, the finite state machine for your store instruction should set *addr_sel* to 0 so that *mem_addr* is set to this value and set *mem_cmd* to indicate a memory write operation.

The HALT instruction in Table 1 has *opcode* = 111. The HALT instruction should cause the program counter to no longer be updated. You can implement the HALT instruction by adding a state reached from *Decode* when the opcode is 111 which loops back to itself. From this state you will need to reset the program counter using *reset*. This instruction is useful because we do not have an operating system to return to when our program ends.

Figure 6 shows an example of a test program you can compile with *sas* to test your load, store and halt instructions. You can also make up your own test program. The X first line, “MOV R0, X” is interpreted by *sas* as a label associated with the line “X:”. The *.lst* file output by *sas* contains:

```
00          1101000000000101          MOV R0,X
```

The X has been converted to 00000101 (i.e., 5) as HALT is placed in memory at address 4 and the next memory location after it is 5. The line *.word 0xABCD* places 0xABCD in memory at address 5. Thus, the


```

MOV R0,X
LDR R1,[R0]
MOV R2,Y
STR R1,[R2]
HALT
X:
.word 0xABCD
Y:
.word 0x0000

```

Figure 6: Example test program for Stage 2.

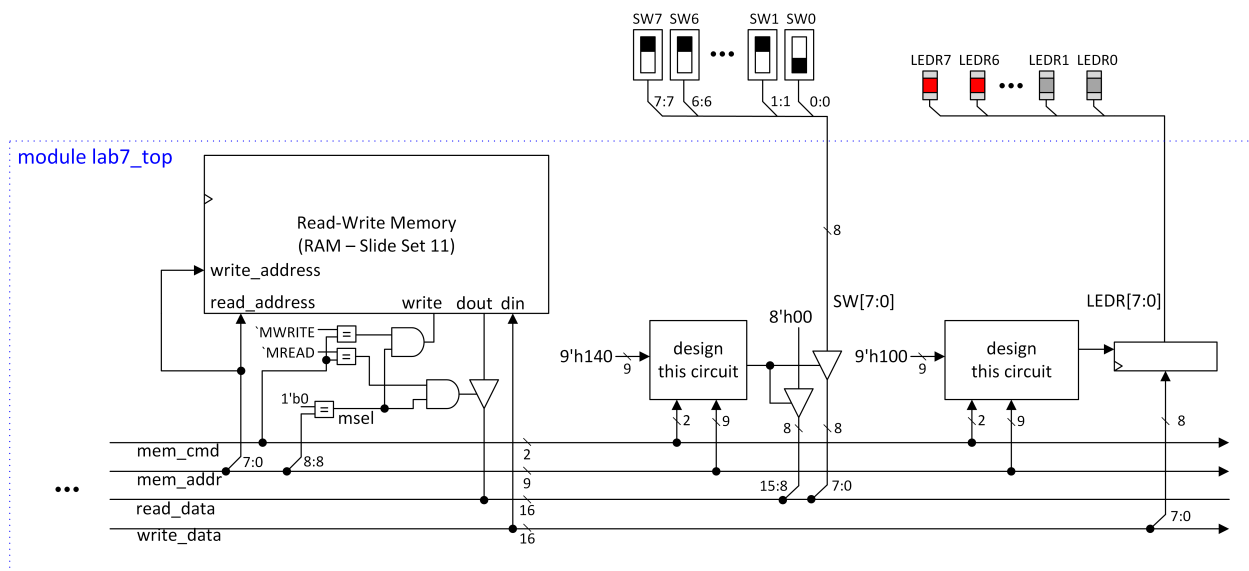


Figure 7: Changes for adding memory mapped input and output devices.

explaining when the transition occurs.

Please remember to check your submission folder carefully as you will lose marks if your github repo does not contain a quartus project file, modelsim project file, or programming (.sof) file. You will also lose marks if your repo is missing any source code (whether synthesizable or testbench code) or waveform format files.

Your mark will be computed in part using an auto grader to evaluate your synthesizable code and in part by your TA.

Use lab7_autograder_check.sv provided on Piazza and run the lab7_check_tb testbench it contains to ensure your submitted code will be compatible with our autograder by ensuring you get the message “INTERFACE OK” in the ModelSim transcript window. Also make sure your code is synthesizable by Quartus; the autograder will run testbenches on your Verilog after first synthesizing it using Quartus. Failure to do either of the above checks may result in a score of 0/5 from the autograder.

Stage 1 (instruction memory) [2 marks autograded; 2 marks assigned by TA] Two marks will be determined by the autograder based upon the number of passing test cases. All instructions from Lab 6 must be implemented correctly so if you lost marks on Lab 6 you should fix those errors. Your TA will assign you up to two marks for this portion if: (a) you can explain how fetching works using a testbench of your own devising that is included in your handin submission along with a wave.do file; (b) you can explain how your state machine code controls fetching instructions; and (c) your Verilog is working on your DE1-SoC.

```

MOV R0, SW_BASE
LDR R0, [R0]      // R0 = 0x140
LDR R2, [R0]      // R2 = value on SW0 through SW7 on DE1-SoC
MOV R3, R2, LSL #1 // R3 = R2 << 1 (which is 2*R2)
MOV R1, LEDR_BASE
LDR R1, [R1]      // R1 = 0x100
STR R3, [R1]      // display contents of R3 on red LEDs
HALT
SW_BASE:
.word 0x0140
LEDR_BASE:
.word 0x0100

```

Figure 8: Example test program for Stage 3.

Stage 2 (data memory) [2 Marks autograded; 2 Marks assigned by TA] Marks will be determined by the autograder based upon the number of passing test cases. Your TA will assign you up to one mark for this portion if you can explain how your Verilog implements load and store instructions.

Stage 3 (memory mapped I/O) [2 Marks assigned by TA] Your TA will assign you up to two marks for this portion based upon demonstrating a test case using the I/O capabilities and/or for explaining your code if it is not working. You can use the test program in Figure 8 or one of your own design.

5 Lab Submission

Your submission **MUST** include a file called “CONTRIBUTIONS.txt” that describes each student’s contributions to each file that was added or modified. If either partner contributed less than one third to the solution (e.g., in lines of code), you must state this in your CONTRIBUTIONS file and inform the instructor by sending email to <mailto:aamodt@ece.ubc.ca>. Note that submitted files may be stored on servers outside of Canada. Thus, you may omit personal information (e.g., your name, SN) from your files and refer to “Partner 1” and “Partner 2” in CONTRIBUTIONS. Submit your code using github classroom.

6 Lab Demonstration Procedure

If you work with a partner both must attend the lab marking session. Autograder marks will be set to zero for students who do not show up for their lab demonstration.

As with Lab 3 to 6, during your demo your TA will have your submitted code with them and have setup a “TA marking station” where you will go when it is your turn to be marked. Be sure to bring your DE1-SoC in case the TA does not have theirs and/or they need to mark multiple groups in parallel.

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
CPEN 211 Computer Systems I, Fall 2023

Lab 7 Bonus: Supporting Branches in the “Simple RISC Machine”

This bonus is autograded only, the submission deadline is 11:59 PM Dec 7 (all lab sections).

1 Introduction

This bonus lab completes your Simple RISC Machine by adding two types of branch instructions. When compiling high-level programming languages such as C, C++ or Java, or interpreting a dynamic language such as Python, *conditional* branch instructions are used to implement “for” and “while” loops and “if” and “switch” statements. Function calls are also implemented using a form of branch instruction. If you completed all of Lab 5 to 7, the changes in this lab will make your Simple RISC Machine “Turing complete”. This means a program can be written to implement *any* algorithm that can run within the 256 word memory you added in Lab 7. How fast your computer runs depends on your detailed implementation such as how many states you used in your FSM from Lab 7 or whether you pipelined your design. To make this lab a bit more fun, a part of your mark will be based upon how fast your final Simple RISC Machine is compared to your classmates.

When you submit your code it will be ranked versus other submissions that have already been submitted. Your code will not be ranked if our correctness checks fail (and moreover will not give you any feedback on what fails). This is to encourage you to test your design; you’ll have to wait until after the submission deadline to get detailed feedback on any failing tests (you can still earn part marks even if some checks fail).

1.1 Branch Instructions

In Lab 7 the program counter (PC) was incremented after each instruction. To support loop and “if” constructs at the assembly level we introduce conditional branch instructions. In the Simple RISC Machine conditional branch instructions either update the PC to $PC+1$ or $PC+1+sx(im8)$, where $sx(im8)$ is the lower 8-bits of the instruction sign extended to 9-bits. When executing the conditional branch instruction the choice between updating the PC to $PC+1$ or $PC+1+sx(im8)$ is made by considering the values of Z, N, and V status flags. The CMP (compare) instruction from Lab 6 is used to set these flags. Thus, a conditional branch instruction determines which instruction to execute next based on the result of the computation so far. The ability of conditional branches to select which instruction to execute based upon the results of a past computation transforms your Simple RISC Machine into a general-purpose computer.

An example of a simple C program that includes a loop is shown in Figure 1. The corresponding Simple RISC Machine program is shown in Figure 2. The right side of Figure 2 shows the assembly code that defines the program. Rather than typing this out you can simply download it from Piazza (see lab7bonus_fig2.s). The left side of the figure shows the corresponding memory addresses where each instruction and data element is placed in memory. Finally, the middle portion shows the contents of memory, in binary. Notice that any given memory location can contain either instructions or data and there is nothing about the memory that distinguishes data from instructions. The program in Figure 2 adds up the values in array “amount” identified by the label amount: on line 24 and stores the total in the memory location with address 0x14 identified by the label “result” on line 29. The initial value of “result” (memory address 0x14) is zero, but after the program reaches the HALT instruction the contents of memory location 0x14 should contain 850 (0000001101010010 in binary).

The only instruction in Figure 2 that is new versus Lab 7 is the “BLT” instruction on line 16. As in ARM (see Flipped Lecture #4), BLT stands for “branch if less than”. The BLT instruction determines the next program counter (PC) value by comparing the negative (N) and overflow (V) flags. Recall from Lab 5 and 6 that the status flags are determined by the ALU while performing a subtraction of *Bin* input from the *Ain* input during a CMP instruction. Thus, the BLT instruction on line 16 uses the values of N and V set by


```

int N = 4;
int amount[] = {50,200,100,500};
int result = 0;
int main(void) {
    int i = 0;
    int sum = 0;
    for(i=0; i<N; i++) {
        sum = sum + amount[i];
    }
    result = sum;
}

```

Figure 1: C code corresponding to assembly code in Figure 2

“CMP R1,R0” on line 15.

Ignoring the possibility of overflow, if the result of R1-R0 is negative, then it must be true that R1 is less than R0. However, if R1 is negative and R0 is positive, then it is possible the subtraction operation performed by CMP R1,R0 overflows and results in a positive value at the 16-bit output of the ALU that feeds into register C in Figure 1 in the Lab 5 handout. To take account of the possibility of overflow BLT checks whether the negative flag (N) is not equal to the overflow flag (V). If they are not equal, then R1 must have

1	Address	Content of memory	Assembly code (lab7bonus_fig2.s on Piazza)
2			
3	0x00	1101000000001111	MOV R0,N // R0 = address of variable N
4	0x01	0110000000000000	LDR R0,[R0] // R0 = 4
5	0x02	1101000100000000	MOV R1,#0 // R1 = 0; R1 is "i"
6	0x03	1101001000000000	MOV R2,#0 // R2 = 0; R2 is "sum"
7	0x04	1101001100010000	MOV R3,amount // R3 = base address of "amount"
8	0x05	1101010000000001	MOV R4,#1 // R4 = 1
9			
10			LOOP:
11	0x06	1010001110100001	ADD R5,R3,R1 // R5 = address of amount[i]
12	0x07	0110010110100000	LDR R5,[R5] // R5 = amount[i]
13	0x08	1010001001000101	ADD R2,R2,R5 // sum = sum + amount[i]
14	0x09	1010000100100100	ADD R1,R1,R4 // i++
15	0x0A	1010100100000000	CMP R1,R0
16	0x0B	0010001111111010	BLT LOOP // if i < N goto LOOP
17			
18	0x0C	1101001100010100	MOV R3,result
19	0x0D	1000001101000000	STR R2,[R3] // result = sum
20	0x0E	1110000000000000	HALT
21			
22			N:
23	0x0F	0000000000000100	.word 4
24			amount:
25	0x10	0000000000110010	.word 50
26	0x11	0000000011001000	.word 200
27	0x12	0000000001100100	.word 100
28	0x13	0000000111110100	.word 500
29			result:
30	0x14	1011101011011101	.word 0xBADD

Figure 2: Example (lab7bonus_fig2.s) with branch instruction (use with lab7bonus_autograder_check.v)

Assembly Syntax (see text)	“Simple RISC Machine” 16-bit encoding																Operation (see text)
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Branch	<i>opcode</i>			<i>op</i>		<i>cond</i>			<i>8b</i>								
B <label>	0	0	1	0	0	0	0	0	im8								PC = PC+1+sx(im8)
BEQ <label>	0	0	1	0	0	0	0	1	im8								if Z = 1 then PC = PC+1+sx(im8) else PC = PC+1
BNE <label>	0	0	1	0	0	0	1	0	im8								if Z = 0 then PC = PC+1+sx(im8) else PC = PC+1
BLT <label>	0	0	1	0	0	0	1	1	im8								if N != V then PC = PC+1+sx(im8) else PC = PC+1
BLE <label>	0	0	1	0	0	1	0	0	im8								if N!=V or Z=1 then PC = PC+1+sx(im8) else PC = PC+1

Table 1: Assembly instructions introduced in Stage 1 (sx & im8 defined in Lab 6 handout)

been less than R0 when “CMP R1,R0” was executed and BLT updates PC to $PC + 1 + sx(im8)$. Here im8 is the lower 8-bits of the BLT instruction and PC is initially the address of the BLT instruction in memory, which is 0x0B for the BLT instruction on line 16 in Figure 2.

The encoding for the BLT instruction is shown in Table 1. From the middle portion of line 16 in Figure 2 we can see that sas encoded “BLT LOOP” as “001000111111010”. The lower 8-bits corresponding to im8 are “11111010” which is -6 in decimal. Thus, if R1 is less than R0 we update the PC to be equal to $0x0B + 1 + (-6) = 11 + 1 - 6 = 0x06$ which is the address of the first instruction after the label “LOOP:” (line 10 in Figure 2).

The rest of the instructions in Table 1 operate as follows: The “B” instruction branches unconditionally to the label provided. This instruction is useful for implementing “if-then-else” constructs and “while” loops. The “BEQ” instruction updates the PC to point to the instruction after the label if the status flags indicate source operands of the last CMP instruction were equal. Similarly, the “BNE” instruction branches to the instruction at the label if the source operands of the last “CMP” instruction were not equal. Finally, the “BLE” instruction updates the PC to point to the instruction after the label if the first operand was less than or equal to the second operand.

1.2 Supporting function calls

The final addition to the Simple RISC Machine is adding support for function calls and returns.

To start, consider the C code in Figure 3. Recall that in C you must declare a function before calling it. The first line helps accomplish this by declaring that function leaf_example takes four arguments of type int and returns a value of type int. The function main calls leaf_example on line 5. The function leaf_example computes the value of “(g + h) - (i + j)” and returns it to main. How do we implement the function call to “leaf_example” at line 5 in Figure 3? You might think you could use the unconditional branch “B” from Table 1 to jump from “main” to the start of the function leaf_example and

```

1  extern int leaf_example(int,int,int,int);
2
3  int result = 0xCCCC;
4
5  void main() {
6      result = leaf_example(1,5,9,20);
7  }
8
9  int leaf_example(int g, int h, int i, int j)
10 {
11     int f;
12     f = (g + h) - (i + j);
13     return f;
14 }

```

Figure 3

Assembly Syntax (see text)	“Simple RISC Machine” 16-bit encoding																Operation (see text)
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Direct Call	<i>opcode</i>				<i>op</i>		<i>Rn</i>			<i>8b</i>							
BL <label>	0	1	0		1	1	1	1	1		<i>im8</i>						R7=PC+1; PC=PC+1+sx(<i>im8</i>)
Return	<i>opcode</i>				<i>op</i>		<i>unused</i>			<i>Rd</i>	<i>unused</i>						
BX <i>Rd</i>	0	1	0		0	0	0	0	0	<i>Rd</i>	0	0	0	0	0	0	PC= <i>Rd</i>
Indirect Call	<i>opcode</i>				<i>op</i>		<i>Rn</i>			<i>Rd</i>	<i>unused</i>						
BLX <i>Rd</i>	0	1	0		1	0	1	1	1	<i>Rd</i>	0	0	0	0	0	0	R7=PC+1; PC= <i>Rd</i>

Table 2: Instructions for function calls and returns.

another unconditional branch “B” at the end of `leaf_example` to jump back to `main`. However, that would permit us to call `leaf_example` from only one place. Instead, you will implement the “branch and link”, BL instruction in Table 2. The use of the BL instruction is shown in Figure 4 (described below), which is the Simple RISC Machine assemble equivalent to the C code shown in Figure 3.

The “branch and link” instruction (BL) performs two operations. First, it saves PC+1, which corresponds to the instruction stored at the next address after the BL instruction, to the *link register* R7. The choice of R7 is an architectur2 design decision that must be agreed upon between hardware and software designers. The Simple RISC Machine use of R7 for branch and link is similar to the use of R14 in ARM. For the BL instruction on line 12 in Figure 4 PC+1 is 0x0a, which corresponds to the address of the next instruction *after* the function call. By saving this address in a known location, namely R7, we enable software to return after the function call is finished.

Second, the BL instruction updates the PC to the address associated with the start of the function being called. For the BL instruction on line 12 in Figure 4 this address is 0x0C, which is the address of the memory location containing the first instruction in the function `leaf_example`.

The use of STR and LDR instructions in `leaf_example` on lines 16, 17, 25 and 26 is for saving and restoring register values to the stack.

To return from `leaf_example` to `main` we use the instruction BX R7 on line 27 in Figure 4. This BX instruction simply copies the named register into the PC.

The last instruction in Table 2 is BLX. This function is useful for supporting object-oriented programming because it essentially enables calling a function with a “pointer”.

2 Lab Procedure

Starting from your code from Lab 7, modify your design in two stages. In the first stage add support for the conditional branch instructions in Table 1 and in the second stage add support for the call and return instructions in Table 2. To do this think through all necessary changes to your existing design based on the instruction definitions in these tables. Modify the provided testbench lab7bonus_stage2_tb.v to work with your state machine as indicated by the comments in the file. Your lab7bonus_top should use the input signal CLOCK_50. If you look in DE1_SoC.qsf you will see CLOCK_50 is connected to pin PIN_AF14 on the Cyclone V on your DE1-SoC. On the printed circuit board of your DE1-SoC this pin is connected to another chip that generates a 50 MHz clock signal. In other words, CLOCK_50 is a 50 MHz clock that is input to your lab7bonus_top. By the end this lab you will have built a real computer so it is finally time to hook it up to an external clock instead of pressing a key to advance each clock cycle. This will allow you to run longer programs quickly.

All student submissions that pass the Lab 8 auto grader will be ranked on performance. Performance on a given workload is given by one over execution time, where execution time can be computed using the following formula:

$$\text{Execution Time} = \text{Total Cycles} \times \text{Cycle Time} \quad (1)$$

Here Total Cycles is the number of clock cycles to execute a given program as measured using ModelSim using a counter reset to zero by the reset (KEY1) and which otherwise increments by one each cycle until

1	Address	Content of memory	Assembly code (lab7bonus_fig4.s on Piazza)
2			
3	0x00	1101011000011000	MOV R6,stack_begin
4	0x01	0110011011000000	LDR R6,[R6] // initialize stack pointer
5	0x02	1101010000011001	MOV R4, result // R4 contains address of result
6	0x03	1101001100000000	MOV R3,#0
7	0x04	1000010001100000	STR R3,[R4] // result = 0;
8	0x05	1101000000000001	MOV R0,#1 // R0 contains first parameter
9	0x06	1101000100000101	MOV R1,#5 // R1 contains second parameter
10	0x07	1101001000001001	MOV R2,#9 // R2 contains third parameter
11	0x08	1101001100010100	MOV R3,#20 // R3 contains fourth parameter
12	0x09	0101111100000010	BL leaf_example // call leaf_example(1,5,9,20)
13	0x0a	1000010000000000	STR R0,[R4] // result=leaf_example(1,5,9,20)
14	0x0b	1110000000000000	HALT
15			leaf_example:
16	0x0c	1000011010000000	STR R4,[R6] // save R4 for use afterwards
17	0x0d	1000011010111111	STR R5,[R6,#-1] // save R5 for use afterwards
18	0x0e	1010000010000001	ADD R4,R0,R1 // R4 = g + h
19	0x0f	1010001010100011	ADD R5,R2,R3 // R5 = i + j
20	0x10	1011100010100101	MVN R5,R5 // R5 = ~(i + j)
21	0x11	1010010010000101	ADD R4,R4,R5 // R4 = (g + h) + ~(i + j)
22	0x12	1101010100000001	MOV R5,#1
23	0x13	1010010010000101	ADD R4,R4,R5 // R4 = (g + h) - (i + j)
24	0x14	1100000000000100	MOV R0,R4 // R0 = return value (g+h)-(i+j)
25	0x15	0110011010111111	LDR R5,[R6,#-1] // restore saved contents of R5
26	0x16	0110011010000000	LDR R4,[R6] // restore saved contents of R4
27	0x17	0100000011100000	BX R7 // return control to caller
28			stack_begin:
29	0x18	0000000011111111	.word 0xFF
30			result:
31	0x19	1100110011001100	.word 0xCCCC

Figure 4: Example (lab7bonus_fig4.s) with branch and link (use with lab7bonus_stage2_tb.v).

the HALT instruction is executed and thus LEDR[8] is set to 1. For this competition, Cycle Time will be measured using TimeQuest Timing Analyzer in the Linux version of Quartus 15.0 after compiling your Verilog for the Cyclone V FPGA in the DE1-SoC. In Quartus look under “TimeQuest Timing Analyzer” -> “Slow 1100mV 85C Model” -> “Fmax Summary” -> “Fmax”. The value of Cycle Time measured in seconds is 1 divided by Fmax. After submitting with handin (or HandinUI) the autograder will be run automatically and if you pass all checks your ranking will appear at a [ranking webpage](#) which is updated automatically each time you submit your code using handin or HandinUI. Note it may take some time for your ranking to appear as each submission takes about 10 minutes to run and other students may have submitted before you.

The autograder requires your design set LEDR[8] to one when executing HALT and zero otherwise. Thus, the autograder requires that after HALT executes, when reset is asserted and there is a rising edge of clk, LEDR[8] returns back to 0 and remains 0 until HALT is executed again. Also, ensure your program counter register output is called PC inside a module with instance name CPU. See lab7bonus_autograder_check.v for more details. You will note that the checker checks that PC is equal to 0x0F while the HALT instruction is stored in memory at address 0x0E. The reason for this can be seen by referring back to Figure 3 in the Lab 7 handout. In that figure you will see there is an “Update PC” state that sets $PC = PC + 1$ and that this state is encountered before the “Decode” state where you would detect the instruction in the instruction register contains a HALT instruction. You don’t have to use these same states in your controller, but to be compatible with the autograder your HALT instruction should set PC to the address of the next instruction. Note that your top-level module should be renamed to lab7bonus_top.

3 Marking Scheme and Competition Instructions

Ensure that when you simulate the module lab7bonus_check_tb in lab7bonus_autograder_check.v it prints out “INTERFACE OK” and that your code is synthesizable by Quartus (e.g. works when downloaded to your DE1-SoC). **Inferred latches and/or synthesis errors in Quartus will result in both disqualification from the performance competition and an additional 3.0 marks deducted.** Ensure you include Quartus and ModelSim Project files.

Table 1 instructions [6 marks autograder] Your autograder mark will be five marks for passing correctness checks for all required instructions.

Table 2 instructions [4 marks autograder] Your autograder mark will be 3 marks for passing all correctness checks for all three instructions in Table 2.

Competition [up to 10 marks] In addition to the above, after the submission deadline (and all legitimate autograder concerns have been addressed) the top 10 submissions that pass all checks will earn a 5 mark bonus (i.e., 15/10). The top ten is based upon your ranking in the [ranking webpage](#) (updated automatically each time you submit your code using handin or HandinUI). Moreover, the #1 top ranked submission will earn 155 marks extra on top of the above (i.e., 30/10, equivalent to an extra 9% to their final grade), the #2 ranked will earn 10 marks extra (i.e., 25/10), and the #3 will earn 5 marks extra (i.e., 20/10). Submissions are ranked by geometric mean speedup across a set of “benchmarks”, versus a low performance reference design.

Submitting early has the virtue of letting you know if your code passes our autograder checks. However, to encourage you to design your own testbenches you will not get any feedback on what has failed if anything until after the submission deadline.

4 Lab Submission

Submit your code using the github invite link for Lab-7 Bonus (only) from https://cpen211.ece.ubc.ca/cwl/github_info.php.