DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
CPEN 211 Introduction to Microcomputers, Fall 2021

Lab 5: Datapath of the "Simple RISC Machine"

Week of Nov 15 to 19 (your code must be submitted via handin 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. Below we review some concepts covered in Flipped Lectures 3 and 4 then discuss how they relate to building the datapath of our computer.

1.1 From C to Assembly

The "Simple RISC Machine" executes programs written using a *very* small set of "instructions". Instructions are the smallest unit of computation that a programmer can specify. 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);
```

To execute this C code on the Simple RISC Machine architecture we will see it is necessary for a compiler—or CPEN 211 student—to first divide the computation into three separate "instructions". Each of the three instructions will specify a small portion of the above computation. These steps are analogous to steps in a cookbook recipe. For example, to bake bread you might follow the steps: "1. Mix flour, water and yeast; 2. Let dough rise; 3. Bake in oven." Similarly, we can implement the C code above using the following three Simple RISC Machine (SRM) instructions:

```
ADD t1, g, h;
ADD t2, i, j;
SUB f, t1, t2;
```

The first instruction, "`ADD t1, g, h`", adds the value of the variables "`g`" and "`h`" and puts them into the *temporary variable* "`t1`". You probably noticed that "`t1`" was not a part of our original C program. We used this temporary variable to help us split up our long line of C code into smaller steps. Similarly, the second line adds the variables "`i`" and "`j`". Finally, the third line subtracts the temporary variable `t2` from `t1` and puts the result in `f`.

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.

1.2 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.

1.2.1 Register File

To compute with the variables `g`, `h`, `i`, `j`, `t1`, and `t2`, our computer needs a way to "remember" their values. To remember their values, the Simple RISC Machine employs the hardware structure known as a *register file* 1, shown in Figure 1.

A register file is a small memory. A *memory* is a hardware block that remembers information. It does this by storing the information as numbers consisting of a fixed number of bits in a fixed number of *locations*. Each location has associated with it an *address*. The address of a memory location is analogous to the street address of a house. The address identifies the house, but is distinct from the contents of the house (who lives there). Another analogy is looking for a book in a library: Each book in a library has a "call number". The call number helps you find the book inside the library. In this analogy, the call number acts like a memory address and the text inside of the book acts like the data (i.e., numbers) that we store in a memory location. For example, suppose we save the number 42 in memory location with address 3. Later we can ask the memory for the contents at address 3 and the memory will return the value 42. In most computers each memory location holds an 8-bit quantity known as a byte. Some memories have a large number of locations so they can store lots of information. For example, if your computer or phone has 16 GB of RAM, this means it has 16 billion locations (the "G" means giga, which stands for one billion), each of which stores an 8-bit value, also known as a *byte* of information (byte is abbreviated to "B"). To simplify the overall design, in the Simple RISC Machine, each location will hold 16-bits.

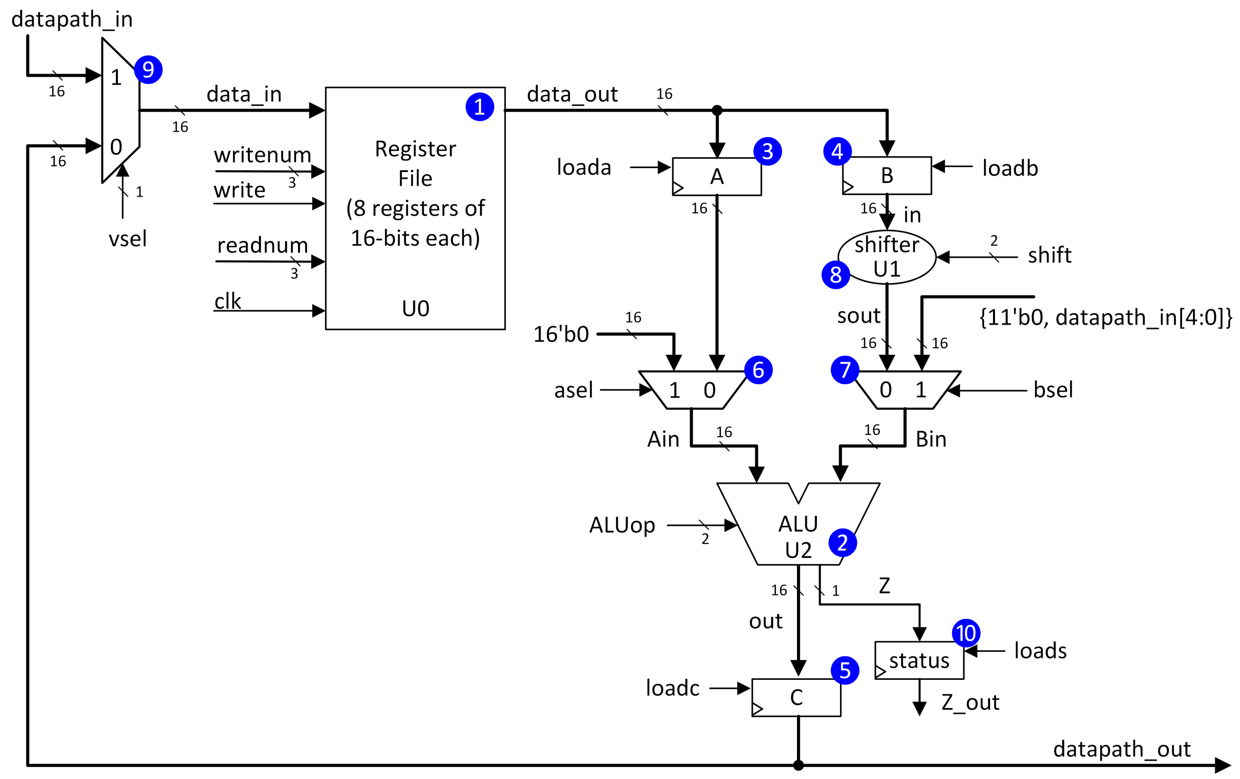

Figure 1: "Simple RISC Machine" Datapath

The register file for the Simple RISC Machine is a memory that has eight locations numbered 0 through 7, each of which can hold a 16-bit number. Notice that to specify one of the eight locations requires only $log_2(8) = 3$-bits, even though each location can hold 16-bits. The eight different locations are more commonly referred to as R0, R1, ..., R7, where the R stands for "register". Thus, the entire register file can store $8 \times 16 = 128$ bits of information. An *individual* 16-bit register inside the register file is built using what is sometimes referred to as a *register with load enable*. A register with load enable can be implemented with the circuit shown in Figure 2.

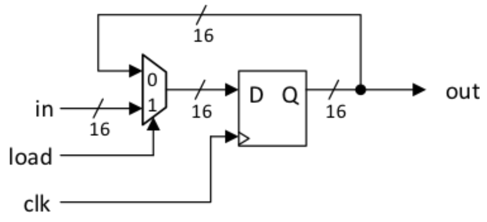

Figure 2: Register with Load Enable Circuit

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 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.

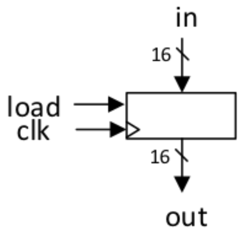

Figure 3: Register with Load Enable Symbol (Note: this symbol is used in both Figure 1 and 4)

To implement the register file, you will use eight separate instances of the 16-bit register with load enable circuit. The overall logic diagram for the complete register file is shown in Figure 4. You will notice this diagram includes two binary-to-one hot decoders. To keep it readable, Figure 4 does not show all locations (e.g., R4, R5 and R6 are omitted).

To store the value of a variable into the register file, we need to pick one of the eight 16-bit registers with load enable. In APSC 160 this choice was made for you by your C compiler. In CPEN 211 you will need to make this choice yourself. For example, we could decide that for our example program we will put "`g`" in `R0`, "`h`" in `R1`, "`i`" in `R2`, and "`j`" in `R3`. Thus, our program now looks like:

```
ADD t1, R0, R1;
ADD t2, R2, R3;
SUB f, t1, t2;
```

Now, for this example you still need to allocate registers in the register file for `t1`, `t2` and `f`. Let's use `R4` and `R5` to hold `t1` and `t2`. What about "`f`"? Well, you could use `R6`, but you are starting to run very low on "free" registers. If the program contains only the one line of C code it does not need to hold onto "`g`" after the first instruction so you can reuse `R0` for "`f`". After making these register "allocation decisions" the program looks like the following:

```
ADD R4, R0, R1;
ADD R5, R2, R3;
SUB R0, R4, R5;
```

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

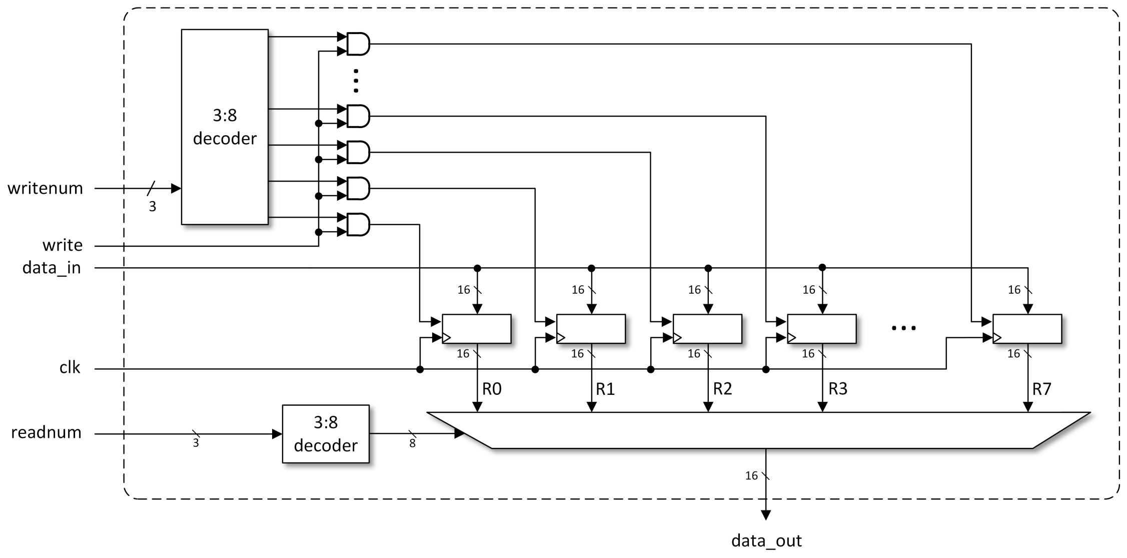

Figure 4: Register File Internal Structure

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 `0000000000101010` (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 `0000000000101010` 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.

To recall, or *read*, the value of "`j`", which is now stored 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.

1.2.2 Arithmetic Logic Unit

This funny symbol labeled 2, which is a trapezoid with a little triangular notch, represents an *arithmetic logic unit* (ALU). The ALU can perform arithmetic or logical operations. This is the main piece of hardware that actual "computes" things inside a computer. Which operation should be performed by your ALU is indicated by the value on the ALUop input shown in Table 1.

Note that it is important you use the values in the above table or in Lab 7 the assembler program we give you will generate code that does not work with your Simple RISC Machine. 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).

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

In Lab 7 we will add support for instructions used to implement features of C such as "if" statements. These instructions will need to know some information about the values being computed up to that point in the program. One important piece of information will be if the main 16-bit result of the ALU (`out`) was exactly zero. If so, the 1-bit `Z` output should be 1 and otherwise 0.

1.2.3 Pipeline Registers

The datapath contains three 16-bit registers ❸❹❺ with load enable that are not included in the register file. 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.

1.2.4 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 ❻❼. For some instructions added in Lab 6 through 8 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" (described in Lab 6).

1.2.5 Shifter Unit

Some instructions are made more powerful with the ability to quickly multiply one of the inputs to the ALU by a power of 2 or perform integer divide by a power of 2. The shifter unit ❽ is a purely combinational logic block that accomplishes this as follows. The shifter takes one 16-bit input from the Q output of register B ❹ and outputs either the same value or the value shifted one bit to the left or right according 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.

1.2.6 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 ❾.

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

Table 3: Example shift operations starting with 1111000011001111

1.2.7 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 keeps 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.v`. 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. If is inconvenient to meet with your partner in person, meet with them on Zoom and use screen sharing, combined with using a private online repository (see below) can facilitate effective paired programming while working remotely. 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 Revision Control and Regression Tests

Whether you work alone or with a partner, you will save time if you learn to use a revision control system such as "git". There are many tutorials on how to use "git" online. E.g., https://www.atlassian.com/git/tutorials, https://try.github.io/levels/1/challenges/1. Using a revision control system is standard industry practice and more effective than emailing files between partners. Use a (free) **private** online repository such as https://bitbucket.org or https://education.github.com. However you collaborate with your partner it is your responsibility to make sure it is secure and private so no other students in CPEN 211 or at other schools see your code. Revision control goes "hand in hand" with a great engineering practice known as "regression testing". Briefly, the idea of regression testing is that, any time you change something you re-run all of your testbenches to ensure no test case that was passing stopped working (a "regression"). Regression testing can be fast 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 a given hardware unit (e.g., the register file or ALU), you rerun all your tests and only submit your changes to the revision control system if all your 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.v
vlog lab3_top_tb.v
vsim -c work.lab3_top_tb
run -all
quit -f
```

If using Notepad: File>Save As... and for "File Name" enter, **including double-quotes**, `"regress.do"`, then press "Save". Then, launch "`cmd.exe`" and at the `C:\` prompt type `cd <dir>` replacing `<dir>` with the directory containing "`regress.do`" and "`tictactoe.v`". Then, type "`vsim -c -do regress.do`" and hit enter. You should see outputs like you would in the transcript window in ModelSim.

3.2 Recommended Development Sequence

1. Create a new ModelSim project called lab5.

2. Add a file regfile.v and write synthesizable code for your register file in this file. Note that this Verilog must conform to the style guidelines. Compile regfile.v in ModelSim to catch syntax errors. Make sure your register file uses 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.v` 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 and simulate `regfile_tb.v` along with regfile.v in ModelSim. Remember to use the waveform viewer. Even if everything looks OK add some internal signals from inside the register file module you defined in regfile.v to your waveform viewer and rerun the simulation to verify the internal operation is as you expect.

5. 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. See Section 7 for more debugging tips.

6. Once you are satisfied that your register file works in simulation, compile regfile.v in Quartus and verify you see no inferred latches warnings. After synthesis completes, view the resulting logic diagram schematic that Quartus generates using "Tools" > "Netlist Viewers" > "RTL Viewer" to verify the hardware looks as you expect (e.g., combinational logic or flip-flops).

7. *(Optional)* Download the register file to your DE1-SoC and connect it to some top level signals. This step is time consuming. Do this step only if you have time or encounter bugs in Step 13.

8. Only after you have debugged the register file should you go through Steps 2-6 for the ALU. Use the file names `alu.v` and `alu_tb.v`, ensure your testbench module is named `ALU_tb` and ensure your ALU module has the following declaration (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.

9. Only after you have debugged the ALU should you go through Steps 2-6 for the shifter. Use the file names shifter.v and shifter_tb.v, ensure your testbench module is named shifter_tb and ensure your shifter module has the following declaration (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.

10. 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.v. Note that the autograder will assume your register file has the instance label REGFILE and that the input and outputs are consistent with the way they are referenced in lab5_top.v and lab5_autograder_check.v provided on Piazza. Note the autograder and lab5_top.v require that asel, bsel and vsel are binary select inputs. Next, add in the remaining logic blocks 3 4 5 6 7 9 10 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.

11. Write a top level testbench module for your datapath called datapath_tb in datapath_tb.v. 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 datapath modules contining design errors, and these may require more extensive test cases.

12. 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.

13. 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.v 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.v 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.v` is provided on Piazza to enable you to verify your modules are defined consistently with what the autograder expects. **WARNING:** *The file `lab5_autograder_check.v` 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.v` 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 compatibile 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 folder carefully as you will lose marks if your handin 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_leve_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.v
set_global_assignment -name VERILOG_FILE regfile.v
set_global_assignment -name VERILOG_FILE lab5_top.v
set_global_assignment -name VERILOG_FILE datapath.v
set_global_assignment -name VERILOG_FILE alu.v
```

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), 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.v`, `alu.v` and `shifter.v`. 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 3.2. 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.v`, `alu_tb.v` and `shifter_tb.v` 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. It 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.v`. 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.v`. 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 synthesis your datapath module. Your datapath must also use the input output port names and widths implied by `lab5_top.v` and `lab5_autograder_check.v`. 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.v`, `alu.v`, `shifter.v` and `datapath.v` 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. Submit your code using "handin" as described in the document Learning to use Handin using "Lab5".

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 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. 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".

2. 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;
```

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 `Z` are the zero, overflow and zero flags of the status register (only Z is implemented in Lab 5).
- `status` refers to all three of Z, V and Z.
- `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	*opcode*			*op*		*3b*			*8b*								
`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	*opcode*			*ALUop*		*3b*			*3b*			*2b*		*3b*			
`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	*opcode*			*ALUop*		*3b*			*3b*			*5b*					
`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	*opcode*			*op*		*cond*			*8b*								
`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	*opcode*			*op*		*Rn*			*8b*								
`BL <label>`	0	1	0	1	1	1	1	1	im8								`R7=PC+1; PC=PC+1+sx(im8)`
Return	*opcode*			*op*		*unused*			*Rd*			*unused*					
`BX Rd`	0	1	0	0	0	0	0	0	Rd			0	0	0	0	0	`PC=Rd`
Indirect Call	*opcode*			*op*		*Rn*			*Rd*			*unused*					
`BLX Rd`	0	1	0	1	0	1	1	1	Rd			0	0	0	0	0	`R7=PC+1; PC=Rd`
Special Instructions	*opcode*			*not used*													
`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