DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
CPEN 211 – Introduction to Microcomputers

Lab 5: Datapath of the "Simple RISC Machine"

NOTE 1: Work ONLY with your approved partner. Do not show your code to any other students until after BOTH YOU and THEY have submitted Lab 5 via handin. This means you are forbidden to look at any other group's Lab 5 code, including testbench code, until after you or your partner has submitted your final Lab 5 code via handin.

NOTE 2: L1A and L1C will demo on Oct 13; L1B, L1D demo on Oct 15; and L1E demo on Oct 19. All sections demo Lab 6 week of Oct 26. This is the best we can do for Thanksgiving and Midterm timing.

1. Introduction

Starting with this lab you will build a computer that implements a "Reduced Instruction Set Computer" (RISC). The ARM ISA used in iPhone and Android smartphones is an example of a RISC "instruction set architecture" (ISA). The philosophy behind RISC ISA design is to keep things simple. This can both help performance and enable easier compiler development. Other examples of RISC ISAs in common use today are MIPS, PowerPC, Sparc and Altera's NIOS instruction set.

1.1. Motivation

The two goals of the Lab 5 to 7 sequence is to both give you extended practice building (and debugging!) complex working circuits with Verilog and to introduce you to computing at the machine level. This lab should have something of value for all ECE students! For example, if you are interested in writing software, these labs will give you deeper insight than your peers into what might be wrong when a complex software system is not working or if it runs slowly. If you are interested in biomedical engineering, this lab will provide you skills to make your computationally intensive signal processing algorithms for detecting illness better by being able to get more diagnosis done per health care dollar. If you are interested in power systems you will learn how the microcontrollers used in such systems work. The list goes on!

1.2 From C to Assembly

Your "Simple RISC Machine" will *execute* programs written using a small set of instructions that have an *assembly* syntax *similar* to ARM. Appendix B and the end of this handout provides a complete list of instructions your processor will be able to execute by the end of Lab 7 if you manage to do everything. This set of instructions will, in principle, allow your processor to run *any* program given enough time and enough memory (and the development of high-level language compiler tools for our "toy" ISA, which unfortunately do not yet exist).

Before we build anything, consider the following line of code from a C program like those from APSC 160:

```
f = (g + h) – (i + j);
```

To execute this line on the Simple RISC Machine architecture it is necessary to first divide the computation into three separate "instructions". Each individual instruction specifies only a small step of the 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. Each of these steps is analogous to an instruction. We can implement the C code above using the following three 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". Note that "t1" was not a part of our original C program. We added it as a side effect of breaking up a complex C statement into smaller steps. Similarly, the second line adds the variables "i" and "j". Finally, the third line subtracts the temporary value t2 from t1 and puts the result in f. These instructions are not quite ready to execute on the Simple RISC Machine. We need to make one change. Before we do, let's introduce a bit about the computer you are going to build.

1.3 Overview of the Simple RISC Machine Datapath

Part of the hardware that we need to execute these instructions is shown in Figure 1. This portion, which you will build in Lab 5, is called the "datapath". It 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 first describe the datapath components in detail and finish up our assembly example from Section 1.2 before walking through a simple example of how all the parts of the datapath work together.

While at first Figure 1 may look intimidating you should find it is all fairly straight-forward ***if you read this entire handout before starting to write any Verilog***. This handout is long because it gives you detailed instructions and information you will need to save time.

1.3.1 Register File 1

First, consider how the register file 1 should be implemented and how it is used to execute the simple example C program from Section 1.2.

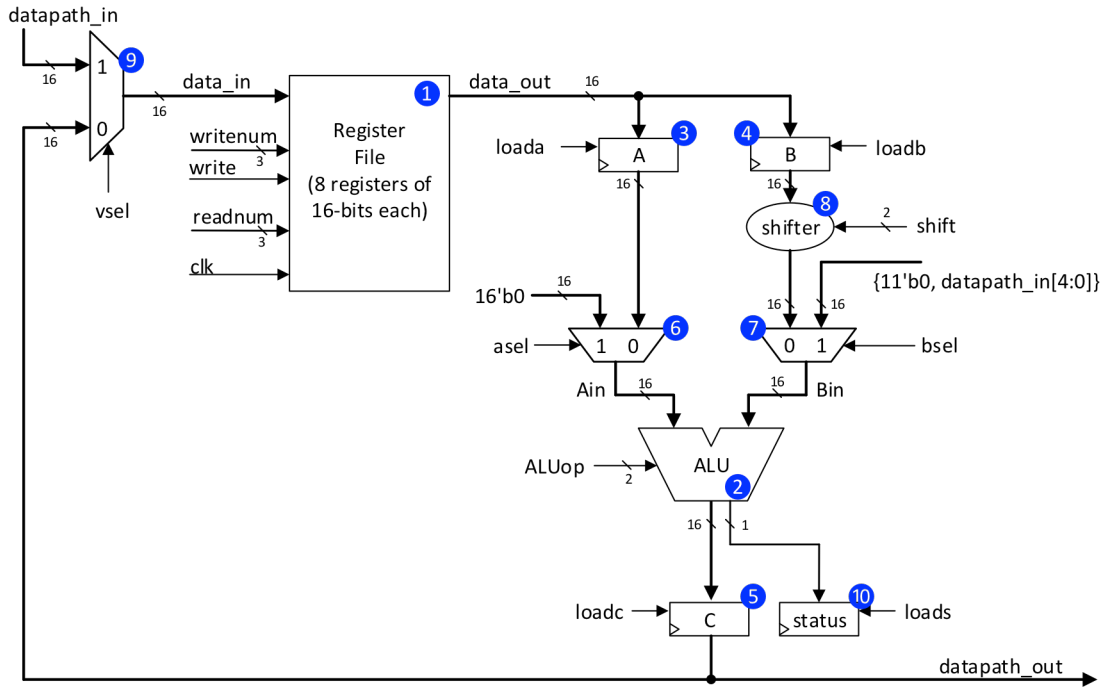

Figure 1: "Simple RISC Machine" Datapath

To run the program the computer needs to "remember" the value of variables such as "g", "h", "i" and "j". There are two locations your Simple RISC Machine will use for this: The first is *main memory*, which you will add in Lab 6, and the second is the register file. In a typical computer it takes roughly 100 clock cycles or longer to access main memory and only one cycle to access the register file. On the other hand typically

main memory can store billions of bits of data whereas the register file can store much less. To run programs fast, computers copy the values they need at any given point of the program from main memory into the register file. Once they are finished updating the variables they copy them back to memory.

The register file for the Simple RISC Machine is a collection of eight 16-bit registers each with a "load enable". Thus, the entire register file can store only 128 bits at a time. You will add a main memory in Lab 6 to enable you to execute more interesting programs on your Simple RISC Machine.

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 following circuit:

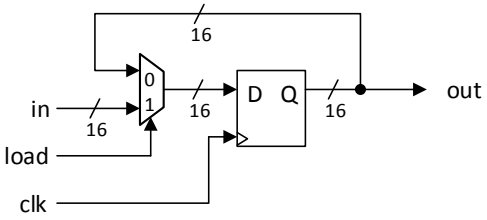

Figure 2: Register with Load Enable Circuit

Referring to the above figure, when the input `load` is 0, the 16-bit value `out` is passed through the top input of the 2-input binary-select multiplexer back into the 16-bit D input. The effect of this is that when `load` is 0 and there is a rising edge of the clock, the value stored in the register does not change. When `load` is 1 the value of `out` is updated to `in` on the rising edge of the clock. We will represent a 16-bit register with load enable using the following symbol:

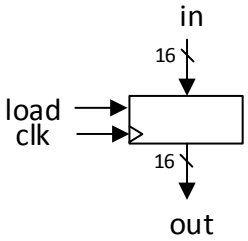

Figure 3: Register with Load Enable Symbol (Note: this is used in both Figure 1 and 4)

To implement the register file, we use eight separate instances of this circuit. Since there are eight of them, we can hold copies of up to eight variables in our program. Since each register holds only 16-bits, our variables can only be 16-bits wide, corresponding to the "short int" data type in C. The circuit to implement the register file looks like shown in Figure 4. Note that, to keep this diagram readable, it does not explicitly show R4, R5 and R6.

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 Lab 5, 6 and 7, 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;
```

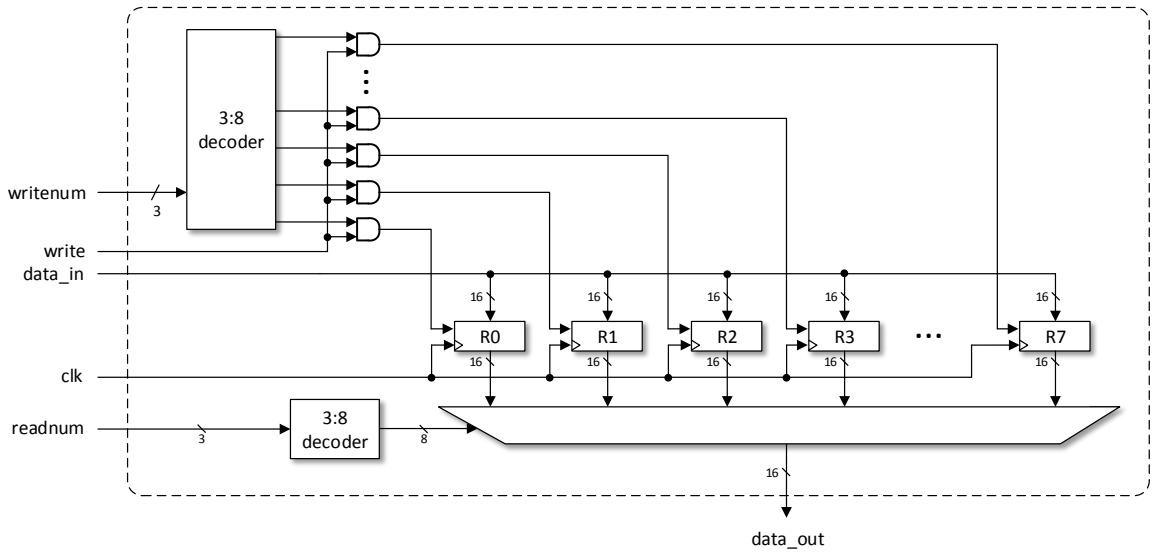

Figure 4: Register File Internal Structure

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. Can you reuse one of R0, R1, R2 or R3? Remember that all variables will be stored in main memory and you only keep a copy of the variable in the register file. Another factor to consider is whether the program still needs the value of "g", "h", "i", and "j" *after* the first, second and third instruction. 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 to 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 "i" gets into and out of R2 inside the register file.

Suppose we want "i" to be 42 *before* executing the instructions above. To put 42 in R2 you would place the 16-bit value 0000000000101010 (binary for 42) on ***data_in,*** set the 3-bit input ***write_num*** to 010 (binary for 2), set ***write*** to 1, and input a rising edge on ***clk***. This causes, the output of the upper 3:8 decoder to be driven to 00000100. Each output bit of the decoder is AND'ed with ***write***. As ***write*** is 1, the load input to R2 is set to 1. On the rising edge of ***clk*** 0000000000101010 will be copied to the 16-bit Q output of R2. 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 the "MOV" instruction that is used to write a constant into a register.

To *read* the value of "j", which is now stored in R2, we set the 3-bit bus ***readnum*** to 010. The 8-bit output of the lower 3:8 decoder will be 00000100 and this will cause the 8-input one-hot select mux to copy the value of R2 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.

1.3.2 Arithmetic Logic Unit 2

The ALU can perform arithmetic or logical operations. This is the main piece of hardware that actual "computes" things inside a computer. With the exception of the shifter unit, you will notice all of the other circuitry in Lab 5, 6 and 7 is used to remember values or get values into and out of the ALU. Which operation should be performed by your ALU is indicated by the value on the ALUop input:

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

Table 1: ALU operations

Note that it is important you use the values in the above table or in Lab 6 and 7 the assembler tool we give you will generate code that does not work with your Simple RISC Machine. Note that the ALU is 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).

1.3.3 Pipeline Registers 3 4 5

The datapath contains three 16-bit registers 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 7 but for now you should keep them.

1.3.4 Source Operand Multiplexers 6 7

To enable more complex instructions besides addition and subtraction it is helpful if we can change the inputs to the ALU. For some of the Simple RISC Machine instructions you will add in Lab 6 and 7 you will want to set the 16-bit `Ain` input to the ALU to zero. For other instructions you will want to use a so-called "immediate operand", which will be described later.

1.3.5 Shifter Unit 8

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 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 7 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 the following table:

`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]

For example, if the input to the shifter was `1111000011001111`, then the output of the shifter would be:

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

1.3.6 Writeback Multiplexer 9

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.

1.7 Status Register 10

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 was exactly zero. If so, the status register will be set to 1 otherwise it will be set to 0.

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.

3. Lab Procedure

You will get through the lab far more quickly if you break down the overall work into smaller parts and complete each one before moving on to the next. 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". To easily collaborate with your partner remotely use a *password protected* online service such as https://bitbucket.org, which should be free for groups of two. ***Whatever you use to collaborate with your partner it is your responsibility to make sure it is secure.*** Revision control goes hand in hand with regression testing. If you follow Tip #11 in Appendix A you can make your tests "self checking" so that you do not need to inspect the waves for every test condition. Then, every time you make a change to a unit, you rerun all your tests and if they all pass you know you are good. This is how testing during development is done in industry.

3.2 Working alone (but read also if working with a partner)

a) Create a new ModelSim project called lab5.
b) 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.
c) Add a file regfile_tb.v to your project for your register file testbench module. Appendix B 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.
d) 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.
e) **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 (e) starting with the block that drives that signal. See Appendix A for more debugging tips.
f) 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).
g) ***[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 (m).
h) Only after you have debugged the register file should you go through Steps (b)-(f) for the ALU.
i) Only after you have debugged the ALU should you go through Steps (b)-(f) for the shifter.
j) 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 ❶, ALU ❷ and Shifter ❽) inside datapath.v. Then, add in the remaining logic blocks ❸ ❹ ❺ ❻ ❼ ❾ ❿ 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.

k) Write a top level testbench for your datapath in datapath_tb.v. It should implement at least the following 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
```

l) 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.

m) 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".

3.3 Working with a partner

Both partners *must* understand the purpose and high level operation of all of the parts of the datapath. Divide up writing the Verilog among both partners. Specifically, each pair of synthesizable + testbench "file" should have a single partner who is the ***owner***. A roughly even division is one partner writes regfile.v, regfile_tb.v, shifter.v, shifter_tb.v and the other writes alu.v, alu_tb.v, datapath.v, datapath_tb.v. The register file is the hardest unit to implement. To enable writing datapath.v *before* regfile.v, alu.v, and shifter.v are completed both partners should agree upon the module inputs and outputs for these units. Each partner follows the procedure outlined in Section 3.1 but only for the units they "own". Both partners should do Step "l" and "k" together. Schedule a time and place for this. It is also good practice to do a "review" of each other's Verilog code. ***This review can only be done by your authorized partner!***

4. Demo Your Code in MCLD 112

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.

Your mark will be computed as the sum of the following. 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.

[3 Marks] One mark for each of regfile.v, alu.v and shifter.v. Up to half of the marks may be deducted here for violations of the style guidelines.

[3 Marks] For each of the unit level testbenches for your ALU, shifter and the register file. To receive full marks you must test both basic functionality and some "corner cases". A corner case is an input you are not expecting to see or not expecting to see often.

[1 Marks] For your datapath.v. Your Verilog must both be completed and conform to the style guidelines to receive this mark.

[1 Marks] For your datapath_tb.v. You should at least test the sequence described in Section 3.1(k).

[2 Marks] Demonstrate 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.

6. Lab submission instructions

IMPORTANT: Partner 2 must complete this by the end of your lab session or your TA will not submit a grade and you will not receive credit. If Partner 1 must submit because Partner 2 is absent then notify your TA. DO NOT include any information that could identify you inside your submitted files. For example, do NOT include student number or name for either partner. The handin program knows the files are Partner 2's because of the ECE account you use to run the handin program. We know who Partner 1 is because this information was submitted to us when you completed the Lab 5-7 Partner Agreement.

The following instructions assume you are logged into a computer in the lab in MCLD 112. As not everyone has a computer in the lab you will need to ask the person closest to you if you can use the lab computer. ***Make sure they log out and you log in***. If you are that person and your neighbor asks, you must allow them neighbor to use your computer for 5 minutes to use handin.

On a lab computer create a directory "Z:\cpen211\Lab5-<section>" (e.g., Z:\cpen211\Lab5-L1A if you are in section L1A) and copy at least the files you added or modified for Lab 5. If your files are on your laptop, transfer them to this directory by sending the files to yourself at an online email account such as Gmail then opening a web browser on the lab computers and download and save the files to "Z:\cpen211\Lab5-<section>".[1]

Then, open up Cygwin **on a lab computer** by going to the start menu and typing "cygwin" and hitting enter or clicking on the Cygwin icon. In the Cygwin window type: `ssh <username>@ssh.ece.ubc.ca` after replacing "<username>" with your ECE account username. E.g., "ssh aamodt@ece.ubc.ca"). If you see:

```
The authenticity of host 'ssh-linux.ece.ubc.ca (142.103.83.22)' can't be established.
RSA key fingerprint is 8e:95:cc:cf:66:9b:da:0f:67:72:28:94:a1:f7:33:1a.
Are you sure you want to continue connecting (yes/no)?
```

type "yes" and hit enter. Type "`chmod go-rx ~/cpen211`" to secure your files. To submit type:

```
handin cpen211 Lab5-<section>
```

For example, if you are in section L1A, type: `handin cpen211 Lab5-L1A`. You will be asked to acknowledge the following prompt:

> ***PLEASE READ THIS STATEMENT CAREFULLY AND ENSURE THAT YOU UNDERSTAND IT BEFORE SUBMITTING YOUR WORK.***
> ***By submitting these files, I indicate that I am fully aware of the rules and consequences of plagiarism, as set forth by the Department of Electrical and Computer Engineering and the University of British Columbia. I hereby certify that the work in the submitted file(s) was performed *only* by me (the owner of the account used to submit this work), except as acknowledged in the work submitted.***
>
> ***Are you sure you want to continue? (y/n)***

To resubmit use the command "handin –o cpen211 Lab5-<section>" (e.g., "`handin -o cpen211 Lab5-L1A`" if you are in section L1A) to overwrite your previous and/or trial submission.

---

[1] If you know Linux/UNIX you can use the unix/linux program "scp" to transfer files to your ECE account under the directory ~/cpen211/Lab5-<section>. If you want to submit your code remotely from home before the lab, this is the best way.

Appendix A: More Debugging Tips

1. Rule 1 "Understand the System". Read the entire handout at least once.
2. Rule 2 "Make it Fail". Think of each test case or test vector like asking a question. The input is the question and the output is the answer. Which questions should you "ask"? Start with testing each basic feature. But good tests ask tough questions. You want to make your register file or ALU or shifter fail with your testbench so that you can fix it before combining it with the other units. A fun warm up exercise: http://www.nytimes.com/interactive/2015/07/03/upshot/a-quick-puzzle-to-test-your-problem-solving.html?_r=0 **(DO NOT COPY TEST VECTORS FROM SOMEONE ELSE)**
3. Rule 3 "Quit thinking and look". Add internal signals to your waveform viewer to verify your theory about what is wrong BEFORE changing any Verilog.
4. Rule 4 "Divide and conquer". For example, how can you verify you correctly write a value into the register file? How would you know if it was written correctly *without* reading the value out again? Again, the only way is to look at the *internal* value of the 16-bit registers. If you just try a test that writes and then reads and it doesn't work, you won't know which part (reading or writing) is broken.
5. Rule 5 "Change One Thing at a Time". If your fix does not work, undo it!
6. Rule 6 "Keep an audit trail." Especially if you are tired, write down what you are trying or the things you are observing so you don't forget.
7. Rule 7 "Check the plug": After hours of debugging it is not uncommon to hear someone say "Why didn't I check that first?"
8. Rule 8 "Get a Fresh Perspective": If you are really stuck call your authorized partner. If you are both stuck go to TA or instructor office hours or make an appointment with the instructor. Please note that the TAs are **not** paid to meet with you outside of lab or regularly scheduled office hours. If they do this it is purely voluntary they are not full time employees of the university. They are students a bit more senior than you are.
9. Rule 9 "If you didn't fix it it ain't fixed". If you think you fixed it, undo your change and verify the bug comes back. Oftentimes a change only appears to make the bug go away.
10. In your top level testbench datapath_tb.v, you may want to use Verilog hierarchical path names to simplify checking whether signals internal to the datapath are acting as you expect. A hierarchical path name is the name of the top level module follows by the module instance names separated by periods. For example, consider code for a binary select multiplexer in Figure 8.14 of Dally which has 3-bit internal signal "s" connecting the output of a decoder to a one-hot select mux. If an instance of Muxb3 was instantiated inside your datapath with label MUX1, and the datapath is called DUT inside your testbench module which is called datapath_tb, you could print out the value of the signal "s" inside of the mux using the following line inside the script part of datapath_tb.v:

    ```
    $display("%b", datapath_tb.DUT.MUX1.s);
    ```

11. Make your testbenches self checking. There are two approaches. One is simply to use an error signal and use if conditions (see test_q84 solution from Problem Set #2). Alternatively, 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. Continuing the above example, 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".
12. 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;
    ```

Appendix B: The Simple RISC Machine Instruction Set Architecture

The information below is only relevant to Lab 6 and 7. Look at it now only if you are curious about those labs.

Instructions you will add in Lab 6:

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*		*3-bits*			*8-bits*								
`MOV Rn,#<imm8>`	1	1	0	1	0	Rn			imm8								R[Rn] = sx(<imm8>)
`MOV Rd,Rm{,<sh_op>}`	1	1	0	0	0	0	0	0	Rd			shift			Rm		R[Rd] = shifted_Rm
ALU Instructions	*opcode*			*ALUop*		*3-bits*			*3-bits*			*2-bits*			*3-bits*		
`ADD Rd,Rn,Rm{,<sh_op>}`	1	0	1	0	0	Rn			Rd			shift			Rm		R[Rd] = R[Rn]+shifted_Rm
`CMP Rn,Rm{,<sh_op>}`	1	0	1	0	1	Rn			0	0	0	shift			Rm		status = f(R[Rn]-shifted_Rm)
`AND Rd,Rn,Rm{,<sh_op>}`	1	0	1	1	0	Rn			Rd			shift			Rm		R[Rd] = R[Rn] & shifted_Rm
`MVN Rd,Rm{,<sh_op>}`	1	0	1	1	1	0	0	0	Rd			shift			Rm		R[Rd] = ~ shifted_Rm
Memory Instructions	*opcode*			*ALUop*		*3-bits*			*3-bits*			*5-bits*					
`LDR Rd,[Rn{,#<imm5>}]`	0	1	1	0	0	Rn			Rd			imm5					R[Rd]= MEM[R[Rn]+sx(imm5)]
`STR Rd,[Rn{,#<imm5>}]`	1	0	0	0	0	Rn			Rd			imm5					MEM[R[Rn]+sx(imm5)]= R[Rd]

Instructions you will add in Lab 7:

Assembly Syntax	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*			*unused*		*cond*			*8-bits*								
`B <label>`	0	0	1	0	0	0	0	0	imm8								PC=PC+sx(<imm8>)
`BEQ <label>`	0	0	1	0	0	0	0	1	imm8								if Z=1 then PC=PC+sx (<imm8>)
`BNE <label>`	0	0	1	0	0	0	1	0	imm8								if Z=0 then PC=PC+sx (<imm8>)
`BLT <label>`	0	0	1	0	0	0	1	1	imm8								if N/=V then PC=PC+sx (<imm8>)
`BLE <label>`	0	0	1	0	0	1	0	0	imm8								if N/=V or Z=1 then PC=PC+sx (<imm8>)
Call & Return	*opcode*			*op*		*Rn*			*8-bits*								
`BL <label>`	0	1	0	1	1	1	1	1	imm8								R7=PC; PC=PC+sx(<imm8>)
`BLX Rd`	0	1	0	1	0	1	1	1	Rd			0	0	0	0	0	R7=PC; PC=Rd
`BX Rd`	0	1	0	0	0	0	0	0	Rd			0	0	0	0	0	PC=Rd

Each row 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 and Lab 7 handouts***.

- Rn, Rd, Rm are 3-bit register number specifiers.
- <imm8> is an 8-bit *immediate operand* encoded as part of the instruction.
- <imm5> is a 5-bit immediate operand encoded as part of the instruction.
- <sh_op> and shift are 2-bit immediate operands encoded as part of the instruction.
- sx(f) *sign extends* the immediate value f to 16-bits.
- shifted_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.
- MEM[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