DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
CPEN 211 Introduction to Microcomputers, Fall 2020

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

Lab 7 is autograded only, the handin deadline is 9:59 PM Thu Nov 5 (all lab sections).

REMINDER: *As per UBC policy, all suspected cases of academic misconduct will be reported to the APSC Dean's office. As outlined in the CPEN 211 Lab Academic Integrity Policy, you must* **NOT** *share or describe any code you write for this assignment with anyone except your* **authorized** *lab partner for Lab 7 and* **NOT** *ask for or use any code offered to you by anyone other than your authorized lab partner. Your partner is "authorized" to work with you for Lab 7 if https://cpen211.ece.ubc.ca/cwl/lab_partners.php says they are your "current lab partner" at the time you start working together on Lab 7 up until you demo your code. The deadline to sign up or change lab partners using the above URL is 96 hours before your lab section. Your code* **will** *be checked for plagiarism using very effective plagiarism detection tools. Software plagiarism detection tools are* **NOT** *fooled/tricked by changing signal or module names, adding/removing comments, changing spaces, etc.... There are infinitely many correct solutions to Labs 3 onwards (and a "larger" infinite number of incorrect solutions). What that means is any given solution is far more unique than your intuition probably suggests.*

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 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 to note this in your CONTRIBUTIONS file (required in Lab 7 for all submissions regardless of whether you are working with a partner or alone).

1.1 Instruction Memory

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. To keep Lab 7 and 8 simple we use a memory with 16-bit memory locations. This choice makes reading an instruction simpler than it might otherwise be because each Simple RISC Machine instruction is 16-bits long.

The Simple RISC Machine needs to keep track of which instruction to execute next in the program. This requirement is fulfilled by adding a special register called a program counter (or PC) that is connected to the address input of the memory as illustrated in Figure 1. The program counter is a register with load enable that contains the address of the next instruction that should be executed in a program. The 16-bit value stored in the memory location associated with this address is the next instruction to execute encoded in binary as described in Table 1 in the Lab 6 handout. To execute the 16-bit instruction at that location your Simple RISC Machine must first read the 16-bit value out of that memory location and place it in the instruction register you built in Lab 6. Notice that the program counter is only 8-bits wide. This is because we want our memory to contain 256 memory locations. Recall 8-bits can be used to specify one of $2^8 = 256$ different things. In this case those things are different memory locations, where each individual memory location can contain either a single 16-bit Simple RISC Machine instruction or, as we will see later, 16-bits of data. Once the instruction is in the instruction register the finite state machine you built in Lab 6 can execute it on the datapath you built in Lab 5.

Figure 1 is a simplified illustration of how the program counter register is connected to memory, the instruction register and the finite state machine. One thing we omitted was a way to reset the PC to the address of the memory location containing the first instruction in the Simple RISC Machine program that you want to run. We can fix this by adding a multiplexer. To keep things simple we will always put the first instruction in the memory location with address zero, which means that initially we want PC to contain 0. Another simplification in Figure 1 is omitting logic that enables the Simple RISC Machine to store data in memory and to connect to input/output devices. Your Simple RISC Machine can be modified to do those things by adding a multiplexer between the program counter and address input of the memory and a tri-state driver between the output of the memory and instruction register as shown in Figure 2, which will be described in more detail later. (To see how to describe a tri-state driver in Verilog refer to Slide Set 8 on CMOS Logic Gates.) As shown in Figure 1, to control updating of the program counter the load enable input of the program counter is is connected to a signal called *load_pc*. The signal *load_pc* is set to 1 or 0 by a modified version of the state machine you built in Lab 6. The signal *load_pc* should be set to 1 by your state machine when you want to update the program counter on the next rising edge of the clock. When the program counter is updated it will contain the address of a new instruction to execute. Similarly, to load the instruction into the instruction register you can add an output called *load_ir* to your finite state machine and connect it to the load enable input of the instruction register. The signal *load_ir* should be set to 1 only when the datapath has completed executing the previous instruction and is ready to execute a new instruction.

Figure 3 illustrates how one might change the finite state machine from Figure 4 in the Lab 6 handout to use *load_pc* and *load_ir* to update the program counter and load an instruction into the instruction register at the appropriate times. The other signals in this figure (*reset_pc*, *addr_sel*, *m_cmd*) will be described in Section 2.1.1 in connection with Figure 2. The four states *Reset*, *IF1*, *IF2* and *UpdatePC* replace the *Wait* state from Lab 6. The *Reset* state is used to reset the program counter to zero. In state *IF1* the address stored in the program counter is sent to the instruction memory. As described below and in class, unlike the register file you designed in Lab 5, the RAM blocks inside the Cyclone V FPGA in your DE1-SoC support synchronous reads. That means the contents of the memory do not appear on the data output (*dout*) until after the *next* rising edge of the clock after we input the address. Thus, we cannot load the instruction register in state *IF1*. On the next rising edge of the clock the state machine transitions unconditionally to state *IF2* (because there is no label on the arrow from *IF1* to *IF2*). When the state machine is in state *IF2* the 16-bit encoded instruction for the instruction at the address specified by the contents of the PC register should be available at the data output of the memory (*dout*). Thus, in *IF2* we set *load_ir* to 1. On the next rising edge

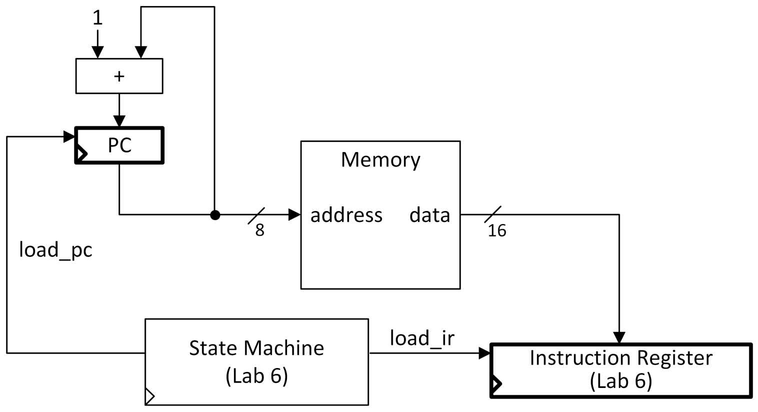

Figure 1: Adding an instruction memory and program counter (simplified).

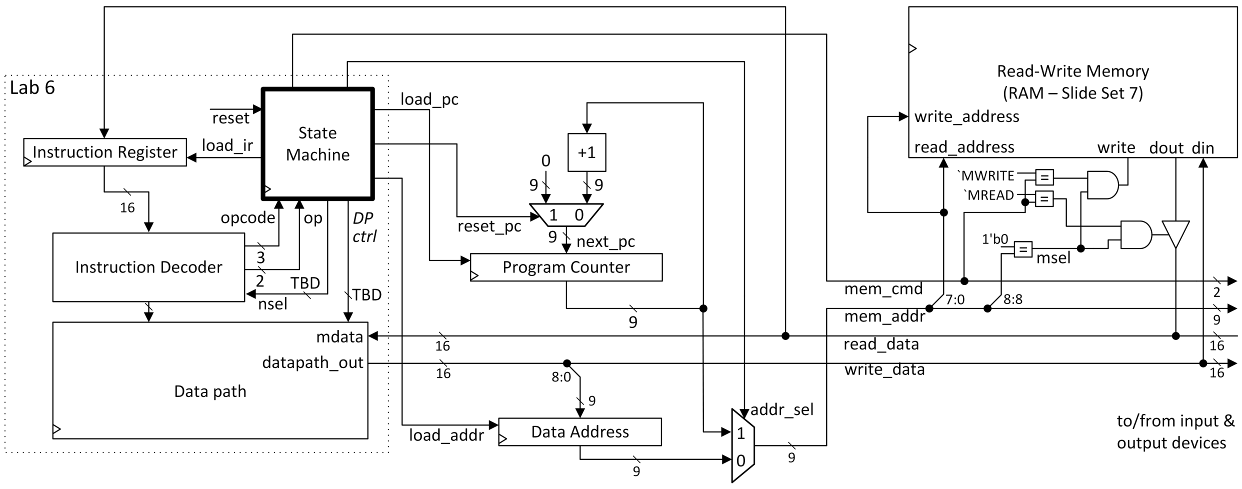

Figure 2: Connecting memory to your CPU from Lab 6 (this figure described later).

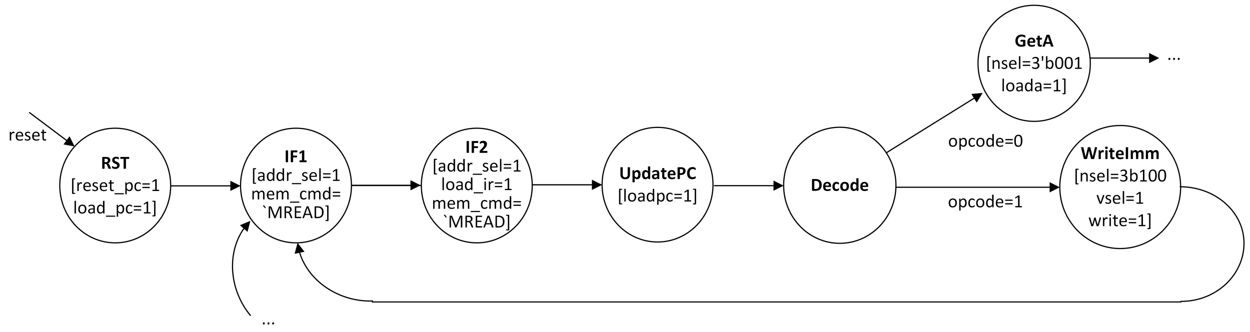

Figure 3: Modified State Machine to Interface with Instruction Memory

of the clock the 16-bits encoding the instruction at address 0 is finally loaded into the instruction register. Then, in state *LoadPC* we set *load_pc* to update the program counter to the address of the next instruction to execute. The timing of *load_pc* and *load_ir* can be seen in Figure 4 which also includes several other important signals from Figure 2.

1.2 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:

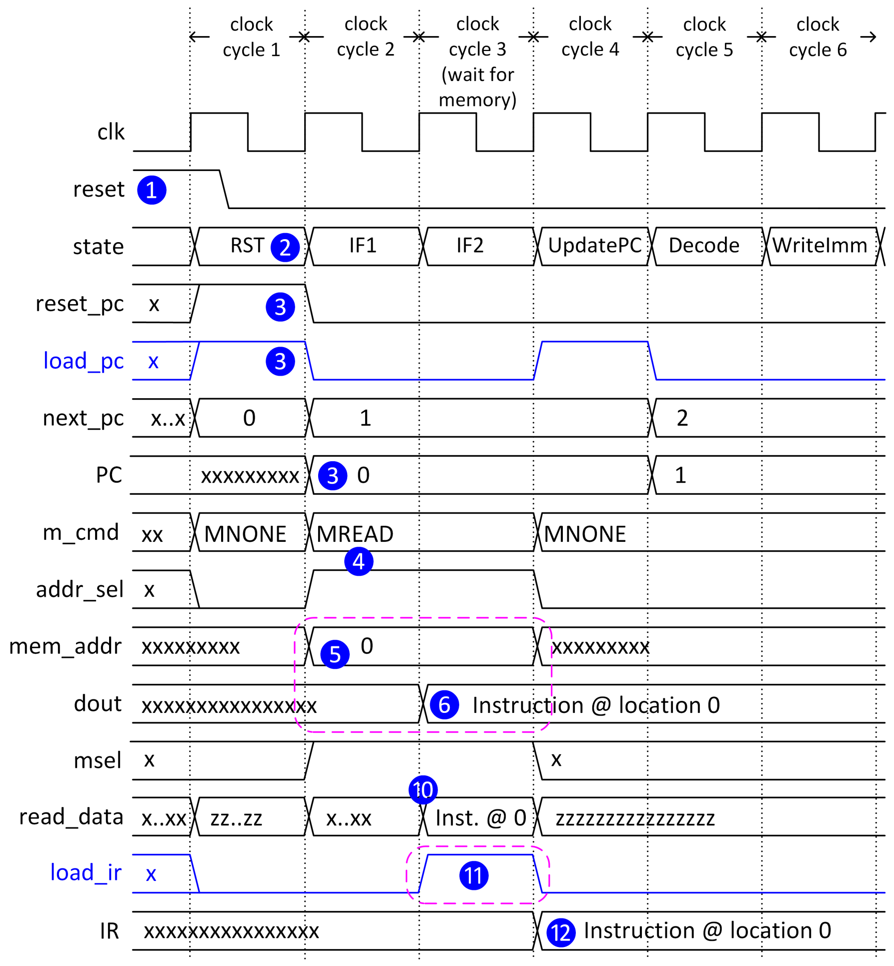

Figure 4: Timing of Instruction Read and PC Update (note one cycle delay on dout).

```
        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.

2 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.

2.1 Stage 1: Executing Instructions from Memory

Implement the hardware shown in Figure 5. 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 below as it will be auto-graded:

```
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;
```

For the memory block you should simply use the Verilog shown in class at the end of Slide Set 7. Make sure the instance name of your memory is `MEM` for the auto-grader. As discussed in class, the Verilog for the Read-Write Memory module follows guidelines provided by Altera to ensure the special memory blocks inside your FPGA are used for our memory rather than logic blocks. You should be able to explain the Verilog for the memory to your TA in case they ask you to.

You will need to modify your state machine to include the four new states that replace the wait state as described in the prior section. The description further below of Figure 4 is intended to help you make these changes. After you make the changes for Stage 1 be sure you can get every instruction introduced in Lab 6 to work. To help you quickly generate `data.txt` files for initializing your memory with your program you can use the Simple RISC Machine assembler that is available at `~cpen211/bin/sas` on `ssh.ece.ubc.ca`. To use this program, you write a text file containing human readable assembly such as the following:

```
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
```

Then you would transfer this file to somewhere in your ECE account and log into ssh.ece.ubc.ca (as you would to use handin). Then, at the shell prompt you would type:

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

where `<file>.s` is the assembly file you want to compile. Two files are output: `<file>.lst` shows both the encoded instructions and the human readable assembly and `<file>.txt` is a file you can use in place of `data.txt` when initializing your memory. You must copy your memory initialization file to 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.

2.1.1 Instruction Fetch Details

The operation of Figure 5 can best be understood with the timing diagram in Figure 4 which illustrates the detailed timing of reading an instruction from memory using the state machine in Figure 3. In cycle 0 the reset input (1 in Figure 5 and Figure 4) is asserted causing the state machine to enter the *RST* state (2) in cycle 1. In Cycle 1, the state machine asserts *reset_pc* which causes zero to be input the program counter register and at the same time asserts *load_pc* which causes this zero value to be loaded by the program counter register (3). In Cycle 2, the state machine sets *addr_sel* to select the output of the program counter (the other input to this multiplexer will be used for supporting load and store instructions in Stage 2) and at the same time, the state machine sets *m_cmd* to `MREAD` to indicate a memory read operation is desired (4). The other possible values for *m_cmd* are `MNONE`, for no memory operation and `MWRITE` for memory write. The value of the constants `MNONE`, `MREAD` and `MWRITE` and the width of the bus `mem_cmd` are something you need to design. This causes the program counter to be driven to the *mem_addr* bus (5). One cycle later (Cycle 3), the synchronous read operation completes and the 16-bit value of the instruction at address 0 is available on bus *dout* (6).

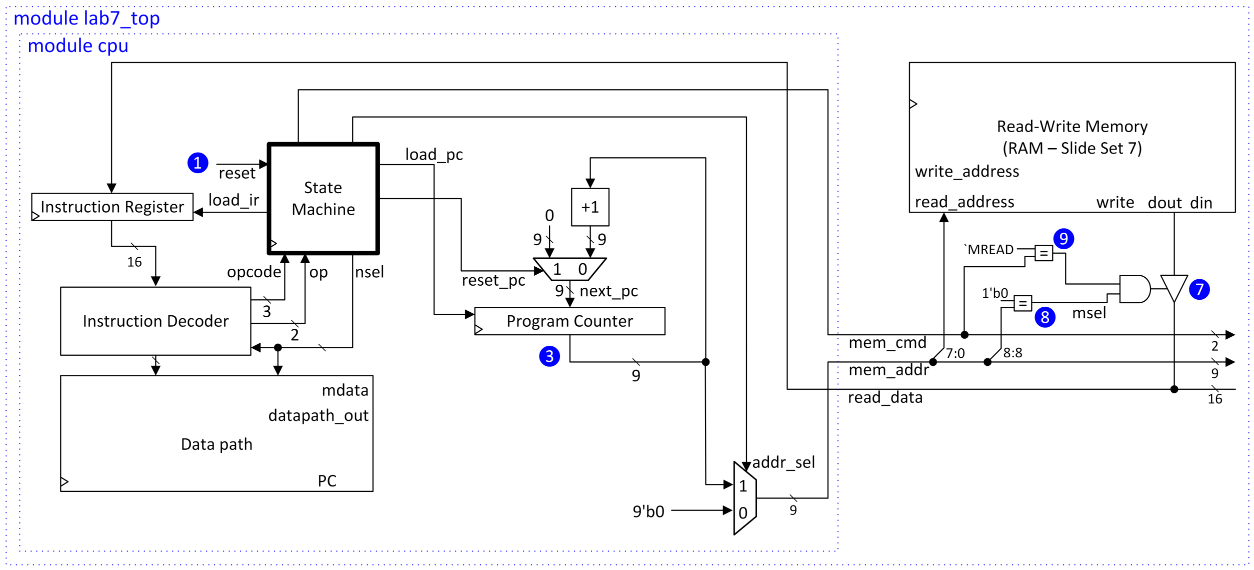

Figure 5: Changes for executing instructions from memory

Next, we examine how values are returned from the memory back to the CPU. To enable input/output devices to share the same address space as memory, we use a tri-state driver (7 in Figure 5). If a memory request is not meant to be handled by the memory, this tri-state driver should have a high-impedance output which is indicated by the value 'z' in Verilog. The enable input to the tri-state driver is a combinational logic circuit designed to enable the tri-state driver ONLY if the CPU is both trying to read something from memory and the address it is trying to read from is a location inside of the memory block. Since the memory contains 256 locations and starts at address zero, this corresponds with addresses from 0 to 255 (decimal) or 0x00 to 0xFF (note this is standard notation for hexadecimal–the "`0x`" means hexadecimal; it does NOT multiply by zero). In Figure 2 and 5 *mem_addr* is 9-bits which corresponds to address from 0 to 511 (decimal) or 0x000 to 0x1FF (hexadecimal). In Stage 3 we will use the remaining addresses from 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 we control the enable input of the tri-state driver with two comparators and an AND gate as described below.

A first equality comparator (8) 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) determines whether the operation is a read command. Note in Stage 2 you will also need to add support for `mem_cmd` equal to the constant `MWRITE` shown in Figure 2. 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). The enable input to the tri-state drive is true during Cycle 2 and 3 in Figure 4, thus causing whatever value is on *dout* to be driven to *read_data* (10).

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

2.2 Stage 2: Putting Data in Memory

In this stage you should first implement all of the logic in Figure 2 that you did not implement for Stage 1. Next, you should add support for the load and store instructions to your state machine datapath controller based upon the specification in the table below. Refer to the Lab 6 handout for a refresher on the notation used in this table. In the column "Operation" the notation "`M[x]`" refers to either reading or writing the

16-bit memory location with address `x`. If you are starting this lab early you might want to watch the videos for Flipped Lecture #3, which describe the operation of load and store instructions in ARM, before starting Stage 2.

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 2. Then, it should set *addr_sel* to 0 so that *mem_addr* is set to this value and set *m_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 2. Then, the contents of `R[Rd]` should be read from the register file and output to `datapath_out` which should now 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 *m_cmd* to indicate a memory write operation.

The `HALT` instruction 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 unconditionally. From this state you will need to reset the program counter using *reset*. This instruction is useful for Lab 7 and 8 because we do not have an operating system to return to when our program ends.

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	
Memory Instructions	*opcode*			*op*		*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]`
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 1: Assembly instructions introduced in Lab 7

Figure 6 shows an example of a test program you can compile with `~cpen211/bin/sas` on ssh.ece.ubc.ca that will exercise your load and store instructions. The `X` first line, “`MOV R0, X`” is interpreted by the Simple RISC Machine assembler (`~cpen211/bin/sas`) as a label. The label is associated with the line “`X:`”. If you examine the `.lst` file generated by the assembler you will see it contains the line:

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

Recall from Lab 6 that the last 8-bits of the MOV immediate instruction encode an immediate value. Here we see `X` has been converted to `00000101` which is 5 in decimal. The assembler translates this symbolic `X` value into an immediate value 5. Why 5? From the `.lst` file you will notice that the instruction `HALT` is placed in memory at address 4. The next memory location after it is 5 and this is the line with the label `X:` on it. The line after says `.word 0xABCD`. This line places the value `0xABCD` in memory at address 5. Thus, the load instruction “`LDR R1, [R0]`” reads the value `0xABCD` into register `R1`. The next two lines causes this value “`0xABCD`” to be written to memory location 6 (you should be able to explain why to your TA). Once you get this test program working you should create additional test programs to ensure your `LDR`, `STR` and `HALT` instruction pass the autograder.

2.3 Stage 3: Memory mapped I/O

Figure 7 illustrates the final changes you will add to your Simple RISC Machine for this lab. For this stage you extend the memory bus and add two *memory mapped* input/output devices. We will use the slider

```
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.

switches for our input device and the red LEDs for our output device. The goal for this stage is to be able to read the values input on the slider switches using the `LDR` instruction and output a value to the red LEDs using the `STR` instruction. To complete this part you should design combinational logic circuits for the two boxes that say "design this circuit". For the slider switches you want to enable the tri-state drive when the memory command (*m_cmd*) is a read operation and the address on *mem_addr* is 0x140. For the red LEDs you want to load the register when the memory command indicates a write operation and the address on *mem_addr* is 0x100. Figure 8 provides a test program for your memory mapped I/O. The instruction "`LDR R2, [R0]`" reads the value on switches SW0 through SW7 into register R2. The instruction "`STR R3, [R1]`" displays the lower 8-bits of register R3 on the red LEDs.

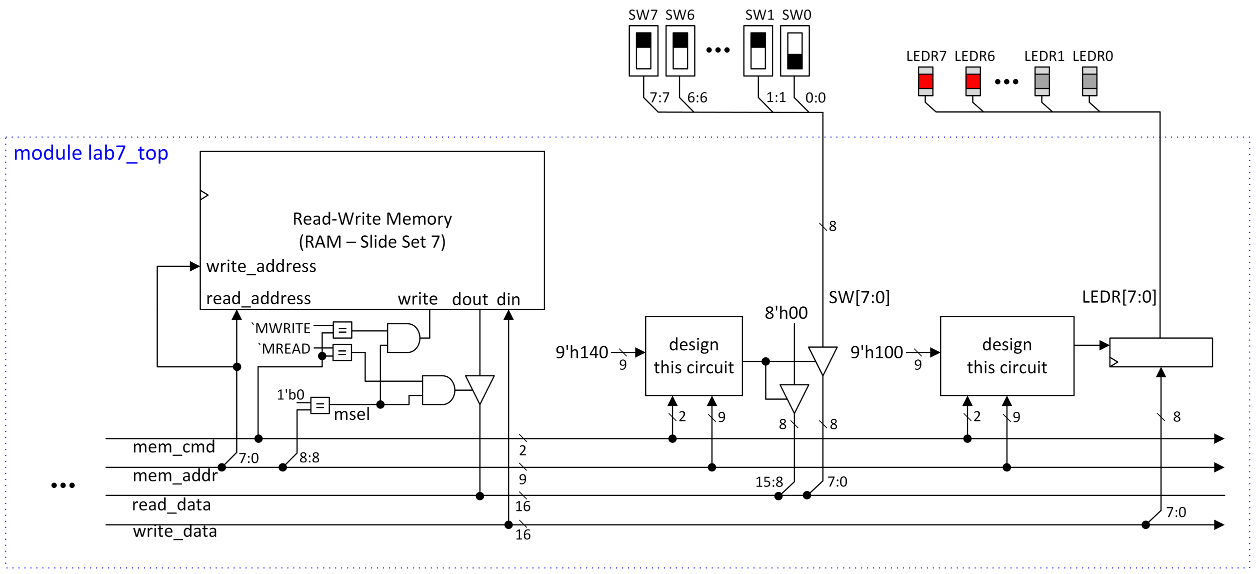

Figure 7: Changes for adding memory mapped input and output devices.

3 Marking Scheme

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.

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

```
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.

code) or waveform format files.

Your mark will be computed using an auto grader to evaluate your synthesizable code.

Use `lab7_autograder_check.v` 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 0/10.

Stage 1 (instruction memory) [4 marks autograded 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.

Stage 2 (data memory) [4 marks autograded] 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 autograded] Marks will be determined by the autograder. Note that both LDR and STR must work to get any marks here. We recommend you get the test program in Figure 8 to work on your DE1-SoC.

4 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 inform the instructor by sending email to . 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 "Lab7".

IMPORTANT: All students including those working alone MUST include a CONTRIBUTIONS file. Your CONTRIBUTIONS file MUST accurately and truthfully report the contributions of each student. While one partner may draft the CONTRIBUTIONS file, by submitting a CONTRIBUTIONS file Partner 1 is officially stating for the record that BOTH partners have affirmed the contents of the submitted CONTRIBUTIONS file are accurate and truthful. If you are Partner 1 you should obtain an email from your partner confirming their approval of the version of the CONTRIBUTIONS file you are submitting before you submit it and you should retain that email for your records. We may ask to see this email at a later date. **If no CONTRIBUTIONS file is submitted via handin your grade for the lab will be at most 5 out of 10 (even if you are working alone).**