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

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

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

1 Introduction

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

2 Tutorial

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

2.1 Instruction Memory

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

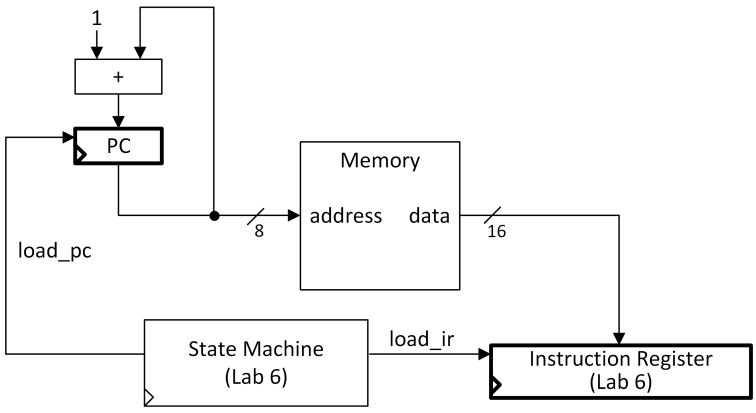

Figure 1: Adding an instruction memory and program counter (simplified).

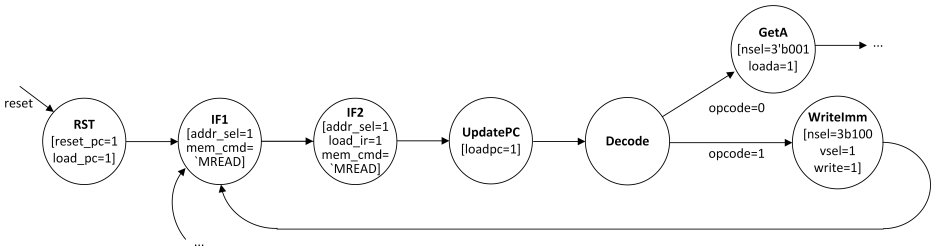

Figure 2: Modified State Machine to Interface with Instruction Memory

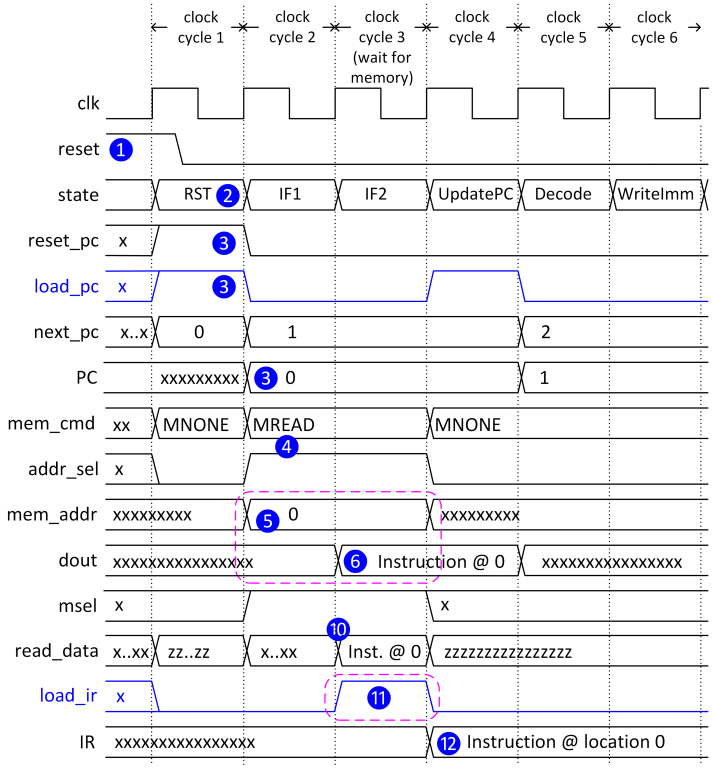

Figure 3: Timing of Instruction Read and PC Update (note one cycle delay on dout).

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

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

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

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

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

2.2 Using Memory for Data

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

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

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

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

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

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

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

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

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

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

3 Lab Procedure

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

3.1 Stage 1: Executing Instructions from Memory

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

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

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

3.1.1 Implementing and Initializing Memory

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

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

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

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

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

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

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

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

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

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

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

3.1.2 Executing Instructions Stored in Memory

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

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

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

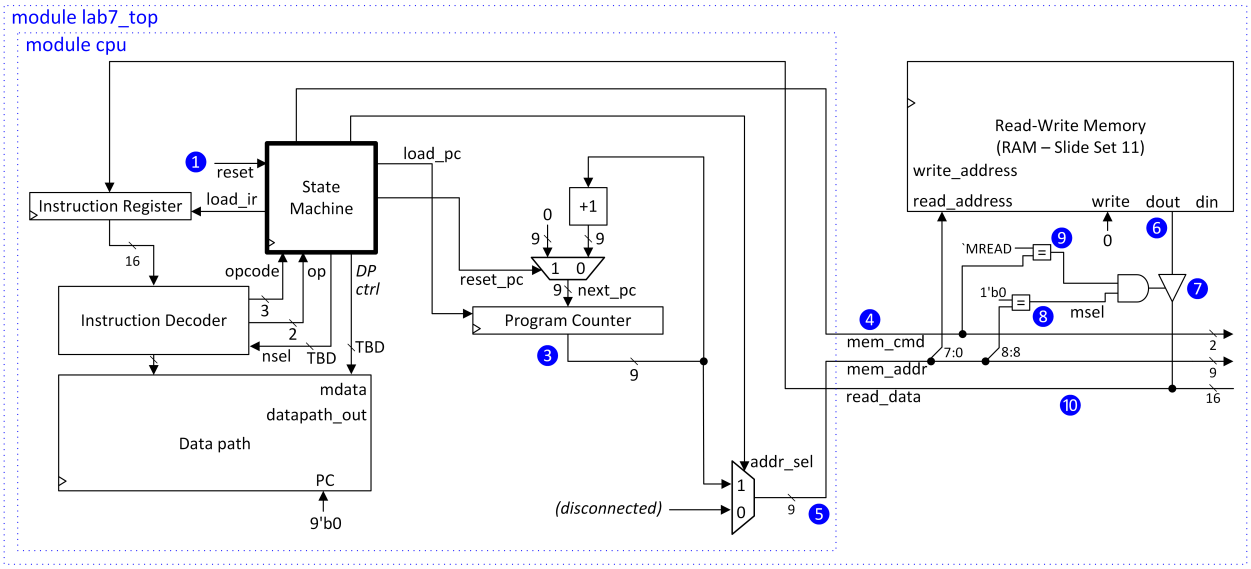

Figure 4: Changes for executing instructions from memory

copy "`in`" to "`out`". When "`enable`" is 0 all 16 tri-state drivers are disconnected (high impedance) and "`out`" is disconnected. The blocks labeled with "`=`" are equality comparison operations that test if the two inputs are equal and output 1 if they are and 0 otherwise. The block labeled with "`+1`" adds 1 to the input.

Next, you should change your state machine so it includes the `RST`, `IF1`, `IF2` and `UpdatePC` states shown in Figure 2. Before doing this it is helpful to understand in more detail the sequence of steps required to get an instruction out of memory into the instruction register using a datapath like that shown in Figure 4. To do that let's revisit the waveform in Figure 3 in greater detail this time considering it alongside Figure 4.

In Cycle 0 the reset input (1 in Figure 3 and 4) is asserted causing the state machine to enter the *RST* state (2 in Figure 3) 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 Figure 3 and 4). 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 *mem_cmd* to `MREAD` to indicate a memory read operation is desired (4 in Figure 3 and 4).The other possible values for *mem_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 we leave for you to design. Setting *addr_sel* causes the program counter value of 0 to be driven to the *mem_addr* bus (5 in Figure 3 and 4).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 in Figure 3 and 4).

At this point we have our instruction at the *dout* output of memory and we could just connect *dout* to the input of our instruction register. The reason we include the tri-state drivers is that, similar to real computer systems, we would like to use the same "bus" for both reading from memory and connecting to I/O devices (Stage 3).

Let's consider in more detail how the circuit connected to the tri-state driver enable input works. In Figure 4 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 4 *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 addresses 0x100 through 0x1FF for accessing input/output devices. Thus, if the Simple RISC Machine tries to read or write to an address from 0x100 to 0x1FF we do not want the memory to respond. To achieve this objective we control the enable input of the tri-state driver with two comparators and an AND gate as described below.

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

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

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

3.2 Stage 2: Putting Data in Memory

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

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

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

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

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

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

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

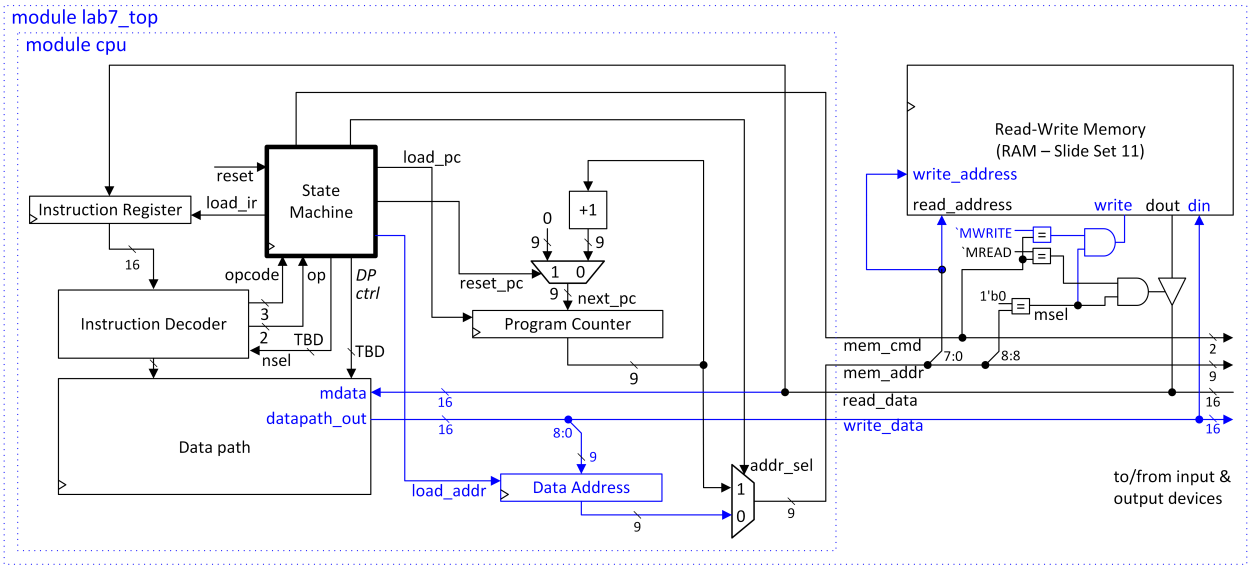

Figure 5: Storing both instructions and data in memory.

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

load instruction "`LDR R1, [R0]`" reads the value `0xABCD` into register `R1`.

3.3 Stage 3: Memory mapped I/O

In Stage 3 you add memory mapped I/O that enables your computer to interact with the outside world using regular load and store instructions. Figure 7 illustrates the additions. We use the slider switches for input and the red LEDs for output. After this stage you should 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. You will need to design combinational logic circuits for the two boxes in Figure 7 labeled "design this circuit". For the slider switches you want to enable the tri-state drive when the memory command (*mem_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. 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.

4 Marking Scheme

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

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

Figure 6: Example test program for Stage 2.

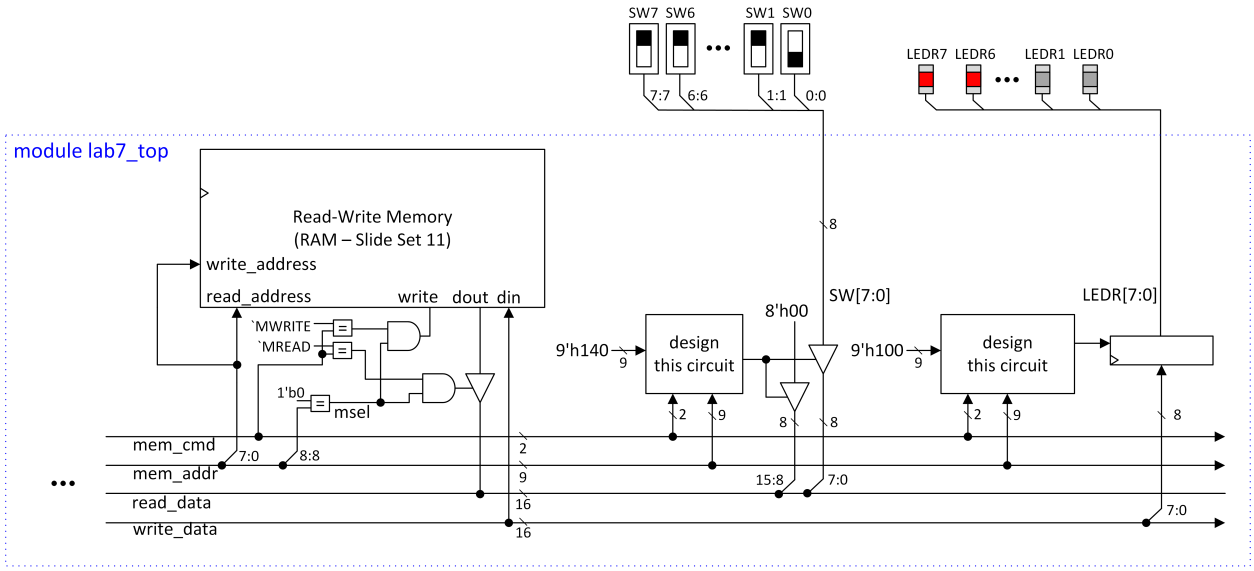

Figure 7: Changes for adding memory mapped input and output devices.

explaining when the transition occurs.

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

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

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

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

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

Figure 8: Example test program for Stage 3.

Stage 2 (data memory) [2 Marks autograded; 2 Marks assigned by TA] Marks will be determined by the autograder based upon the number of passing test cases. Your TA will assign you up to one mark for this portion if you can explain how your Verilog implements load and store instructions.
Stage 3 (memory mapped I/O) [2 Marks assigned by TA] Your TA will assign you up to two marks for this portion based upon demonstrating a test case using the I/O capabilities and/or for explaining your code if it is not working. You can use the test program in Figure 8 or one of your own design.

5 Lab Submission

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

6 Lab Demonstration Procedure

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

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