CPEN 211 2022W1

LAB 7

DUE: `handin` submission 2022-11-27 23:59:59 Vancouver time; lab demo: Week 13

LOGISTICS

Partners. Lab partners (groups of 2) allowed but optional. To work with a partner you must first **register them as a partner** using https://cpen211.ece.ubc.ca/cwl/lab_partners.php. The deadline to sign up a partner is **96 hours before the `handin` deadline**. If you miss this deadline you must do the lab alone.

Submission. You must submit your code using `handin` using "Lab7" by the `handin` deadline above or your mark for this lab will be **zero**. Be sure to submit **all deliverables** with the **exact file names** as required by each task below. **No archive files (e.g., zip) will be accepted**, regardless of contents.

If you are working with a partner, your submission **must** include a file called `CONTRIBUTIONS.txt` that describes in detail each partner's contributions to the submission. If you are missing this file, you may lose all marks in the lab. If either partner contributed less that one third of the effort they may lose all marks.

Lab demos. As in the previous labs, you will need to demo your testbenches, design RTL, and simulations to the TA who is marking you, and answer any questions they have for you. If you are working with a partner, you **must both be present** for *either* partner to receive any marks. Each partner must be able to answer all questions about the lab, RTL, etc, regardless of which part(s) they contributed.

Autograding. Up to 50% marks for each tasks will come from running your code and testbench through our own autograder. This means you must exactly follow task directions about module names, port names, file names, and functionality, and exactly follow all the submission directions. If **any** of the files you submit RTL fail to compile or synthesize, you will receive 0 marks regardless of how many marks the TA assigned.

Academic integrity. This lab **must be done individually or with the registered partner** (see above). You may not communicate with any other students about this lab. Before starting the lab, you must also read the Academic Integrity Policy (available on Piazza), and you must comply with it.

DELIVERABLES CHECK

Before submitting, make sure you have all of the deliverables:

- `regfile.sv` (provided to you)
- `ram.sv` (provided to you)
- `task1.sv`
- `tb_task1.sv`
- `task1.vo`
- `task2.sv`
- `tb_task2.sv`
- `task2.vo`
- `task3.sv`
- `tb_task3.sv`
- `task3.vo`
- `ram_init.txt`
- `CONTRIBUTIONS.txt` (empty if working alone)
- `fib.s` (optional bonus task only)

SPECIFICATION

In labs 5 and 6, you built a datapath and controller FSM for the Potato Machine™ CPU. In this lab, you will add a memory, implement instruction fetching, and add load/store instructions.

This will require three new components:

- A **read/write memory**. This will hold Potato Machine™ instructions as well as data.
- A **program counter** that holds the *address* of the instruction being fetched by the Potato Machine™.
- A **data address register**. This will store the address for the next memory operation.

You will also need to make small **changes to the CPU** you developed in lab 6 to support fetching and the instructions you'll need to implement.

All of the tasks in this lab require you to work with a synchronous RAM memory. Refer to the Appendix to learn how to use the memory during design and testing.

Overall system structure

The overall structure of the system you will implement in this lab is shown in fig. 1. The changes with respect to the last lab:

❶ We have a synchronous RAM connected to the CPU. The RAM has 16-bit words and 256 entries (and so has an 8-bit address).

❷ The instruction register (IR) is now loaded from the memory read output, and its enable signal now comes from the controller FSM.

❸ A program counter (PC) register stores the address of the next instruction to be fetched from memory; the enable is driven by the controller FSM.

❹ With the load enable asserted, the PC can be updated to either PC + 1 or the value on the external `start_pc` port, depending on the `load_pc` output from the controller.

❺ A data address register (DAR) remembers the address that will be used by load and store instructions; its input is the eight least-significant bits of the datapath output, and its enable comes from the controller.

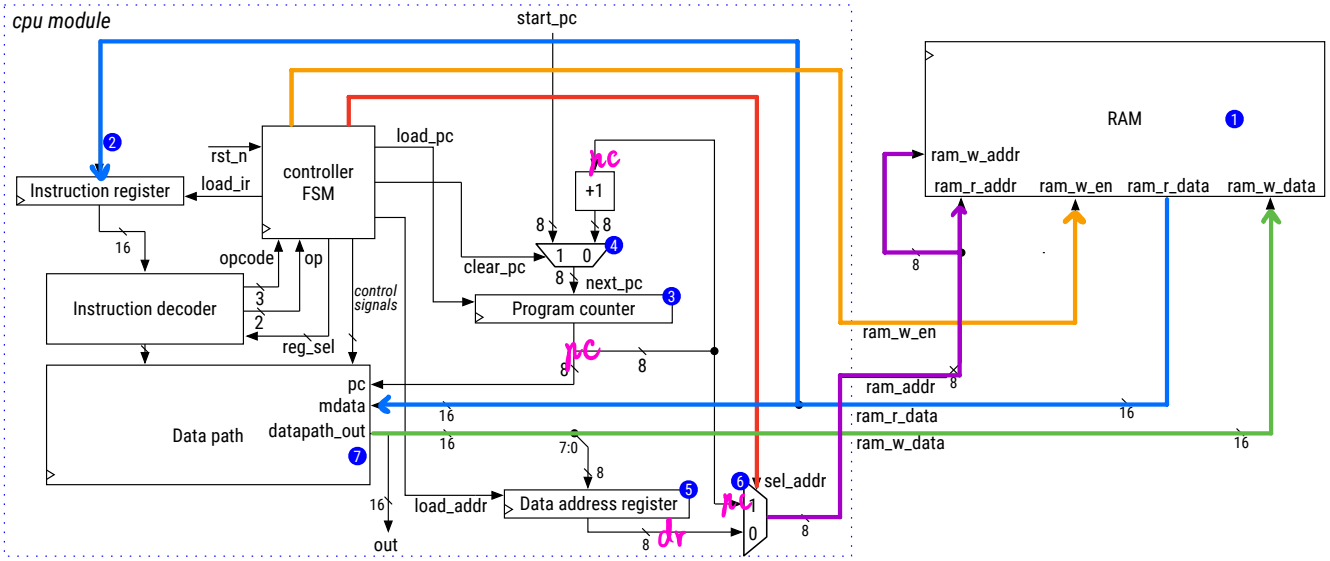
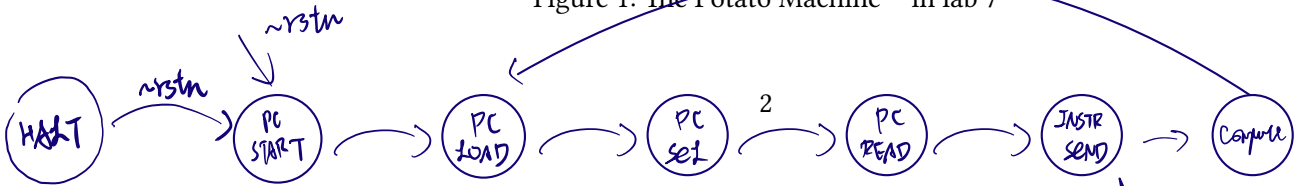


Figure 1: The Potato Machine™ in lab 7



❻ The address presented to the RAM can come either from the PC or the DAR (depending on whether we wish to fetch instructions or load data).

❼ The datapath can now input both the PC or the memory read output, and the datapath output appears on the external out port.

New instructions in this lab

In this lab, you will implement three new Potato Machine™ instructions, listed in table 1:

Assembly syntax	Potato Machine™ 16-bit encoding																Operation
	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

HALT When executed, this stops further instructions from being fetched or executed, and the Potato Machine™ does nothing until reset.

LDR Reads a 16-bit word from memory address *m*, where *m* is the sum of the contents of register Rn and the sign-extended five-bit immediate.

STR Takes the value in register Rd and stores it in RAM at memory address *m*, where *m* is computed as for LDR.

Instruction fetch details

Instruction fetch operates according to three simple rules:

1. The first instruction fetched and executed after reset must be from address on `start_pc`.
2. If the current instruction was fetched from address *n*, the next instruction is fetched from address $n + 1$.
3. Following a HALT instruction, no more instructions are fetched or executed until reset.
4. All instructions fetched are fully executed.

To implement this, you will have to modify your control FSM to control the operation of the PC, the IR, the data address register, and the memory, adding appropriate delays for memory reads.

Load/store implementation details

Executing memory instructions occurs in two phases, described below.

Computing the effective address. First, the required memory address is computed by adding the value in register Rn to the sign-extended five-bit immediate (see table 1). The result (called the *effective address*) is stored in the Data Address Register (DAR).

Accessing memory: LDR. For loads, the memory already drives the value stored in memory at the address found in the DAR. All we need to do is correctly configure the RF writeback mux and assert the RF writeback enable at the right time. (Note that the memory timing shown in the Appendix does not consider the DAR.)

Accessing memory: STR. For stores, we need to read another value from the RF, and pass it through the datapath unmodified. Once that value makes it to the datapath output, you'll want to assert the memory write enable for

one cycle; since you already have the effective address in the DAR from the first phase, you will be writing the value you just sent through the datapath to the memory at the desired address.

You will need to alter your control FSM and likely add new states to implement these new instructions. For `LDR` and `STR` only, the clock cycle budget is 10 cycles; for all other instructions, it remains at 5 cycles.

Testbenches

As part of the lab, you will again be developing unit tests. We will autograde your testbench by seeing whether it can detect faulty implementations. In order to do this, each testbench template has a single-bit output called `err`, which should be 0 if all of your tests passed, and become 1 (and stay there) as soon as any of the tests fail.

Each testbench you submit must:

- interact with the modules **only via the module ports** (i.e., must not reference internal signals);
- use only **named port connections** (the syntax that looks like `.foo(bar)`) to instantiate the DUT, to ensure that it works with our autograder;
- output 1 on `err` if any of your tests failed, or 0 if all tests passed;
- your testbench must fully run when `run 100000` is entered in the Modelsim Tcl console after compiling (i.e., it must take no longer than 100,000 simulation ticks).

To support your tests, you can use a `ram_init.txt` file to initialize the memory. We will read this into our own memory when we mark your testbenches; don't forget to hand it in. You only have one file, so you will have to reset the machine with different `start_pc` values to start different tests, ending each test with a `HALT` instruction.

Your tests should be descriptive, and your testbench should report the pass / fail status of *each test case* as well as the total count of tests passed / failed. The autograder will ignore any text output, but your TA won't.

TASKS and DELIVERABLES

Single file per task. Because you will hand in design files for different tasks separately, all of your modules (**except the regfile and memory**) must be declared **in the same file.** Otherwise the autograder won't know which files to compile for which task, and you will lose all autograder marks. For the regfile and memory, you must use the designs we provided, so that we can replace them when testing.

Memory initialization. The memory initialization values for *all* of your testbenches must be in `ram_init.txt`. You can use the `start_pc` input to distinguish different testcases. We may of course provide a different initialization file, so your hardware may not depend on any memory contents.

Marks breakdown. For each task, you will submit an RTL design file, a testbench, and the gate-level netlist corresponding to your design. Half of the marks in each task will come from your design, and half from the testbench that you've written. As in labs 5 and 6, up to 50% of your marks will come from an autograder.

Non-synthesizable RTL. If either (i) your RTL does not synthesize in Quartus, or (ii) the synthesized design has different functionality than your RTL, you will receive **zero marks** for your design. If your synthesized design has precisely the same functionality as in RTL simulation but contains latches, you will lose some (but not all) marks for your design. In either case, you can still receive marks for your testbench.

Common requirements

- All RTL and testbenches must be in (System)Verilog.
- All sequential logic must be triggered on the rising edge of input `clk`.
- All resets are **synchronous** and **active low**.

- The (System)Verilog you write for each task **must** be entirely in the skeleton files provided for that task.
- Your **must not** modify the module names, port names, or port types / declarations in the skeleton files (for example, you may not add **reg** to port names, etc.). Remember that (System)Verilog is case-sensitive.
- You **must** use the RF and memory definitions provided to you as-is, without modification. (This is because we will be replacing them to test your design.)
- You may not *redefine* the RF or the memory in your own files. (This is because we will be replacing them to test your design.)
- Your design **must** be synthesizable and free of latches.
- Your testbenches **must not** interact with the relevant DUT other than via its module ports.
- Your testbenches must automatically check and report whether each test case succeeds or fails.
- Your testbenches must correctly report test failures on the `err` port.
- Your netlist must be generated for Modelsim-Altera using the Verilog target, not any other language.

Task 1: Instruction fetch [4 marks]

Add the PC, implement the instruction fetch operation, and add the `HALT` instruction.

Handin deliverables: `task1.sv`, `task1.vo`, `tb_task1.sv`, `ram_init.txt`.

Task 2: Loads [3 marks]

Implement `LDR` as described in the Specification section. Synthesize your design and thoroughly test it.

Handin deliverables: `task2.sv`, `task2.vo`, `tb_task2.sv`, `ram_init.txt`.

Task 3: Stores [3 marks]

Implement `STR` as described in the Specification section. Synthesize your design and thoroughly test it.

Handin deliverables: `task3.sv`, `task3.vo`, `tb_task3.sv`, `ram_init.txt`.

BONUS: Fibonacci [1 mark]

For this task, write a Potato Machine™ assembly program that (i) reads value n from memory at address 255, (ii) iteratively computes the n^{th} Fibonacci number*, and (iii) stores it in memory at address 254. Your program will be started at PC = 0.

This would of course be trivial if it weren't for the fact that we don't have any branches, and we don't have a way to compute with the result of `CMP`. But notice that: (i) our data and instructions reside in the same memory, and (ii) you can use arithmetic to simulate conditional operations.

Handin deliverables: `fib.s`

test 2

MOV R1, #12
LDR R2, [R[R1], #4]
R2 = 100

test 3

MOV R0 #20
MOV R1 #28
STR R1, [R[R0], #15]

addr = 35 ⇒ m[35] = 28
addr = 28

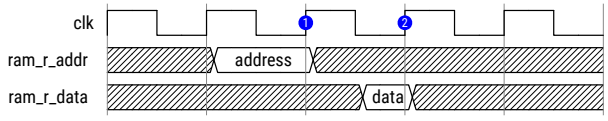
*n is defined so that when $n = 5$ the output should be 5.

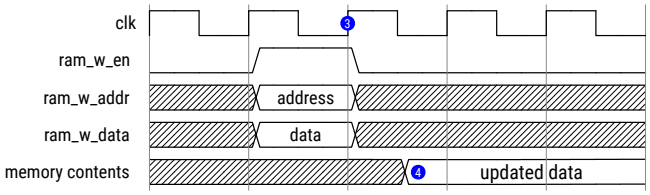
APPENDIX

FPGA memories

FPGAs typically contain on-chip SRAM memories (called *block RAMs* or *BRAMs*)[†]; as you'll have learned from the memory lecture, using these is much more area-efficient than building large arrays of flip-flops. We will use one of these to implement a memory for our Potato Machine™ processor. To do this, we first have to understand how these memories work.

The timing of the reading and writing operations in these BRAMs is shown below. Both reading and writing are *synchronous*, meaning that the results (or effects) appear in the cycle *after* the address is presented. (Note that this is unlike the RF from lab 5, which had combinational reads.)





Reading. Provide address on `ram_r_addr` so the RAM can capture it at clock edge ❶. Before the next clock edge ❷, the corresponding data will appear on `ram_r_data`.

Writing. Provide the address on `ram_w_addr` and concurrently set `ram_w_en` so that the RAM can capture them at clock edge ❸. Before the next clock edge ❹, the corresponding memory entry will have been updated.

We have provided a BRAM block implementation of the correct size in `ram.sv`. You may **may not modify this file**, because Quartus can only infer memories from RTL written using a very specific pattern.

Working with memories in simulation

For debugging, we will want to initialize this memory to some known contents at the start of simulation. To do this, the memory we provided will read its contents from `ram_init.txt` at RTL simulation start time, so you would modify this file and restart simulation. (This file must be in the current working directory from the simulator's perspective, which is by default where your Modelsim project is.)

This initialization technique also works in Quartus, except the file is read at *synthesis time* rather than simulation time, and turns into constants in the gate-level `.vo` file. Therefore, you will have to re-synthesize the entire design if you update the init file with new testcases.

The format for the `ram_init.txt` file is the format used by the Verilog `$readmemb` system task. For example:

```
@00 1101001100101010
@01 1101011100010001
@02 1010001110100111
@03 1110000000000000
```

This file describes a memory with four words initialized. The numbers prefixed with `@` are hexadecimal addresses, and the bit strings that follow are the contents at each address in binary.

Memory operation example

The lab files contain `ramtest.sv` that illustrates how to use the `ram` module. You might want to try this out and study the waveforms to understand how the RAM operates, including which cycle things happen on.

[†]You can learn more at https://www.intel.com/programmable/technical-pdfs/654378.pdf.

Assembler

We have provided a Potato Machine™ assembler for you so you don't have to encode instructions by hand. The sources are on Piazza if you wish to compile it yourself; otherwise you can run it on `ssh.ece.ubc.ca`.

For example, if this is your `foo.s`:

```
MOV R3, #42
MOV R7, #17
ADD R5, R3, R7
MOV R2, ADDR
LDR R1, [R2]
HALT
ADDR:
.word 0xF00D
```

You would run

```
~cpen211/bin/sas foo.s
```

This would produce `foo.txt` with

```
@00 1101001100101010
@01 1101011100010001
@02 1010001110100111
@03 1101001000000110
@04 0110001000100000
@05 1110000000000000
@06 1111000000001101
@07 0000000000000000
... lots more zeros ...
```

You can rename this to `ram_init.txt` and use it to initialize your RAM. The assembler also produces a `foo.lst` file that lists both the encodings and the instructions at each address.

The assembler allows using labels to denote addresses, which is especially useful for memory operations. For example, in the assembly code above, the label `ADDR` identifies address 6 (because there are instructions at addresses 0 through 5), and the `.word` directive places the 16-bit hexadecimal value `0xF00D` at that address. This way we can use an immediate-type `MOV` instruction to load `ADDR` into `R2`, and then use that for the `LDR` instruction to load the value at that address (`0xF00D`) into `R1`.