CPEN 211 2022W1

LAB 6

DUE: `handin` submission 2022-11-20 23:59:59 Vancouver time; lab demo: Week 12

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 "Lab6" 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 labs 3 and 5, 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:

- `datapath.sv`
- `alu.sv`
- `shifter.sv`
- `regfile.sv`
- `datapath.vo`
- `tb_datapath.sv`
- `idecoder.sv`
- `idecoder.vo`
- `tb_idecoder.sv`
- `controller.sv`
- `controller.vo`
- `tb_controller.sv`
- `cpu.sv`
- `cpu.vo`
- `tb_cpu.sv`
- `CONTRIBUTIONS.txt` (empty if working alone)

SPECIFICATION

In lab 5, you built a datapath for the Potato Machine™ CPU. In this lab, you will add an instruction register and create control logic that will allow the datapath to execute some Potato Machine™ ISA instructions.

This will require three new components:

- An **instruction register** that holds the current instruction being executed by the Potato Machine™. This is necessary because most instructions require multiple cycles to be executed in your datapath*.
- An **instruction decoder**. This will "unpack" the 16-bit encoding we use for Potato Machine™ instructions into the various signals that go into your datapath.
- A **controller FSM** that will drive the various datapath signals to execute a small subset of Potato Machine™ instructions.

You will also need to make small **changes to the datapath** you developed in lab 5 to support the instructions you'll need to implement.

At the end, you will put everything together to create a **small CPU** that can execute single instructions.

Subset of ISA implemented in this lab

In this lab, you will implement two types of instructions: (i) ALU instructions, which operate on values stored in the architectural registers (i.e., the RF), and (ii) move instructions, which either copy one register to another or copy an immediate value (a constant encoded in the instruction) into a register. The instructions you need to implement are 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	
Move Instructions	*opcode*			*op*		*3b*			*8b*								
`MOV Rn,#<im8>`	1	1	0	1	0	Rn			im8								`R[Rn] = sx(im8)`
`MOV Rd,Rm{,<sh_op>}`	1	1	0	0	0	0	0	0	Rd			sh		Rm			`R[Rd] = sh_Rm`
ALU Instructions	*opcode*			*ALUop*		*3b*			*3b*			*2b*		*3b*			
`ADD Rd,Rn,Rm{,<sh_op>}`	1	0	1	0	0	Rn			Rd			sh		Rm			`R[Rd]=R[Rn]+sh_Rm`
`CMP Rn,Rm{,<sh_op>}`	1	0	1	0	1	Rn			0	0	0	sh		Rm			`status=f(R[Rn]-sh_Rm)`
`AND Rd,Rn,Rm{,<sh_op>}`	1	0	1	1	0	Rn			Rd			sh		Rm			`R[Rd]=R[Rn]&sh_Rm`
`MVN Rd,Rm{,<sh_op>}`	1	0	1	1	1	0	0	0	Rd			sh		Rm			`R[Rd]= ~sh_Rm`

Table 1: Assembly instructions implemented in this lab

Encoding details

The bitfields in the table 1 have the meaning described below.

Opcodes. Some bits that are constant but different for each instruction identify the instruction. For example, an opcode of `110` identifies the instruction as `MOV` and an op of `10` indicates that the source operand is an immediate encoded in the instruction. In assembly syntax, immediate values are preceded with #, as in `MOV R3, #42`.

Register numbers. Each instruction either reads or writes some registers; these are specified as `Rd`, `Rm`, and `Rn`. For example, `ADD R2, R5, R3` has `Rd=3'b010`, `Rn=3'b101`, and `Rm=3'b011`. In the instruction description, `R[Rd]`, `R[Rm]`, and `R[Rn]` refer to the *values* in those registers; that is, we are adding the values inside register R5 to the value in register R3, and storing the result in register R2.

*e.g., because your RF only has one read port and most instructions read two registers. Refer to the example in lab 5 if you need a refresher.

Immediates. Some instructions contain an *immediate* operand that is encoded directly in the instruction in the `im8` field. This is an 8-bit *signed* value; `sx(im8)` in the instruction description indicates that the value is *sign-extended* into a 16-bit signed value before computing on it.

Operand shift. Some instructions can optionally shift one of the operands (`Rm`, corresponding to register B in your lab 5 datapath) by one bit. This is indicated by the `sh` field, which has the following meaning:

sh	operation
00	no shift
01	left shift
10	logical right shift
11	arithmetic right shift

Table 2: `sh` encoding

In the instruction description, `sh_Rm` indicates `R[Rm]` shifted according to the value in the `sh` field. In assembly syntax, these are indicated like in ARM as `LSL#1`, `LSR#1`, or `ASR#1`, as in for example, `ADD Rd, Rn, Rm, LSL#1`.

Instruction semantics

The operation performed by each instruction is described in table 1. A slightly more verbose description follows:

`MOV Rn, #<im8>` takes bits 0 to 7 of the instruction (the `im8` field), sign extends these bits to a 16-bit value, and stores this value in the register identified by `Rn`.

`MOV Rd, Rm{,<sh_op>}` copies the value in register `Rm` (optionally shifted as specified by `sh`) into register `Rd`.

`ADD Rd, Rn, Rm{,<sh_op>}` adds the value in register `Rn` to the value in register `Rm` (optionally shifted as specified by `sh`), and stores the result in register `Rd`.

`CMP Rn, Rm{,<sh_op>}` takes the value in register `Rn` and subtracts from it the value in register `Rm` (optionally shifted as specified by `sh`). The `status` output is modified depending on the subtraction result: `Z` if the result was 0, `N` if the result was negative, and `V` if the operation overflowed or underflowed.[†] The actual subtraction result is then discarded. (You might want to think about how this implements a comparison operation; the instruction is called `CMP`, after all.)

`AND Rd, Rn, Rm{,<sh_op>}` performs bitwise AND between the value in register `Rn` and the value in register `Rm` (optionally shifted as specified by `sh`), and stores the result in register `Rd`.

`MVN Rd, Rm{,<sh_op>}` performs bitwise negation of all the bits of the value in register `Rm` (optionally shifted as specified by `sh`), and stores the result in register `Rd`. The shift operation is performed *before* the bitwise negation.

Note that only the `CMP` instruction modifies the `status` value.

Overall CPU structure

The overall structure of your lab 6 Potato Machine™ is shown in fig. 1.

After reset, the CPU is in a *wait* state, indicated by the `waiting` outputting logical 1.

An instruction can be loaded into the CPU and executed as follows:

cycle 1: The 16 bits that encode the instruction (per table 1) are presented on the `instr` bus, and `load` is simultaneously asserted for one cycle to load the instruction into the instruction register.

[†]Lab 5 only had the `Z` status bit; you will add `N` and `V` in this lab.

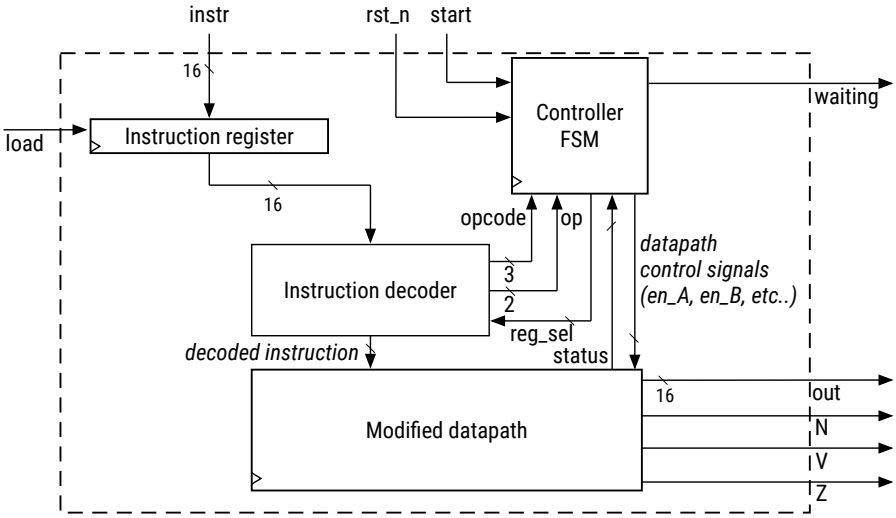

Figure 1: Overall CPU structure

cycle 2: The start signal `start` is asserted **for exactly one cycle** to indicate that the instruction should be executed.

cycles 3 to 3+*n*: The CPU is busy executing the instruction for the next *n* cycles (*n* varies by instruction). During this time, the `waiting` signal is logical 0.

remaining cycles: The execution has finished, and the CPU is back in the wait state. The `waiting` signal is logical 1. In this state, another instruction may be loaded and executed by starting this sequence over again.

In addition to `waiting`, the CPU also outputs the datapath output value (`datapath_out`) as `out`, as well as the three datapath status bits `N`, `V`, and `Z`.

Modified datapath

The datapath from lab 5 will need to be modified to support the instructions you implement in this lab and the following labs. The modifications are shown in fig. 2.

Writeback mux. The mux in front of the RF now has four inputs, with the corresponding select values of `wb_sel` shown in fig. 2. In this lab, you will use the `sximm8` input to implement the move-immediate instruction; this value will come from your instruction decoder. The other inputs (`mdata` and `pc`) are not used in this lab (in a complete Potato Machine™, they will be needed to implement memory operations and branches); you should set both of them to 0.

Signed immediate offset. The mux in front of the B register will use the value `sximm5` as its constant input. This signal will come from your instruction decoder. In a complete Potato Machine™, this will be used as an immediate offset to compute memory addresses.

Three-bit status. The status output now has three bits: `Z` (zero), `N` (negative), and `V` (overflow), which are 1 if and only if the result is zero, is negative, or overflowed/underflowed, respectively.

Instruction decoder

The instruction decoder, shown in fig. 3, selects specific fields (bit ranges) of the instruction register, and applies sign extension to some of them as shown in the figure. It takes two inputs: the output of the instruction register (`ir`), and a `reg_sel` input from your control FSM that indicates which of the three possible register fields (`Rd`, `Rn`, `Rm`) should appear on `r_addr` and `w_addr`. The values of `reg_sel` that select each input are specified in fig. 3.

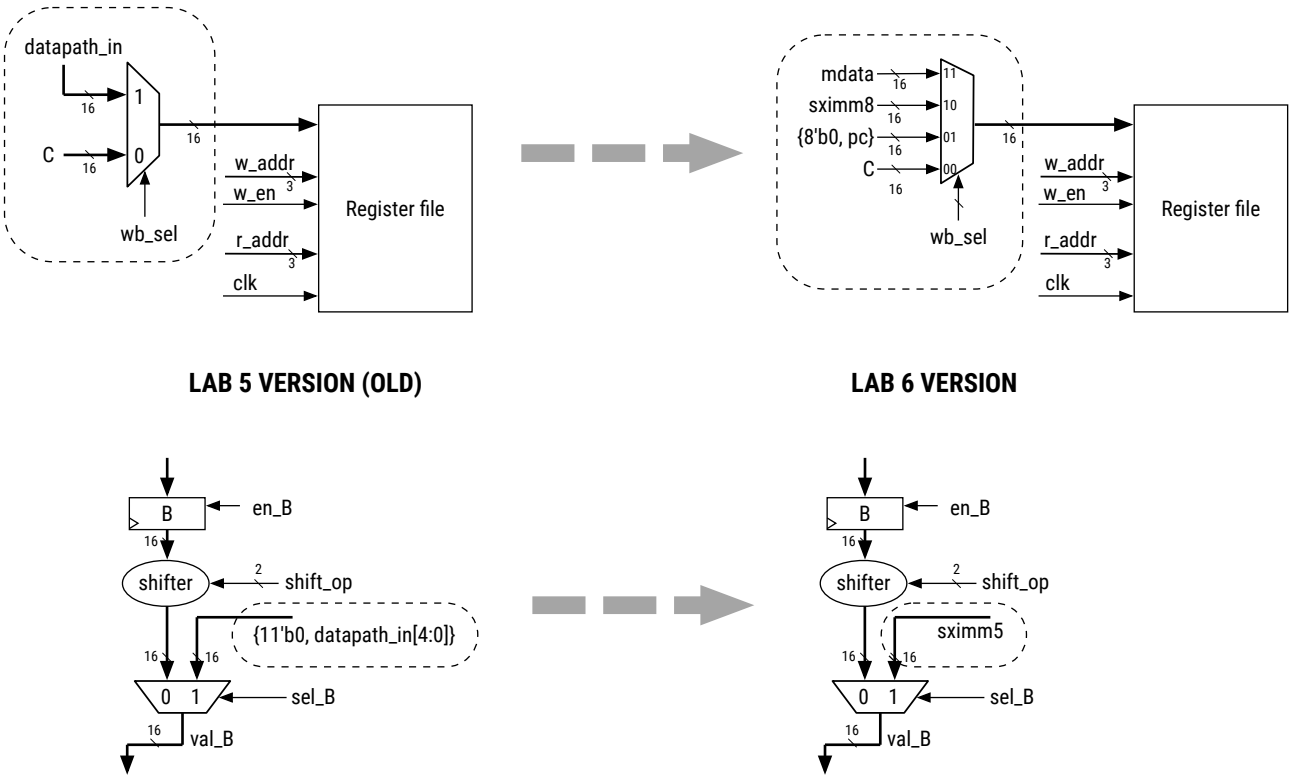

Figure 2: Datapath modifications; changes inside the dashed areas

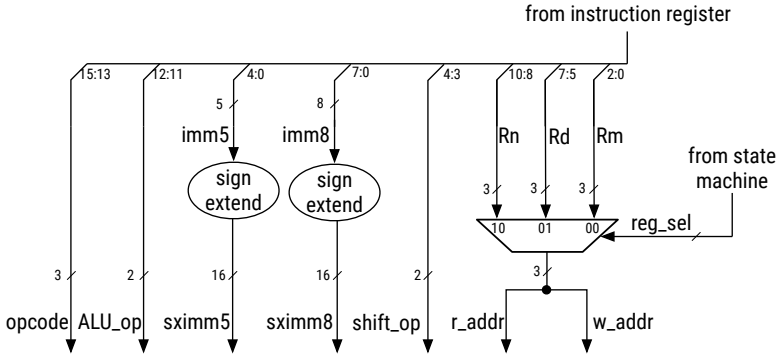

Figure 3: Instruction decoder

Controller FSM

The controller FSM orchestrates the decoder and datapath to execute the instructions from in table 1. In addition to clock and reset, it interacts with the enclosing CPU module via `start` and `waiting`, receives some fields from the instruction decoder (`opcode`, `ALU_op`, and `shift_op`), and the status bits from the datapath (`Z`, `N`, an `V`). In turn, it outputs the `reg_sel` signal to the decoder, and the various select and enable signals to the datapath. Note that the FSM template may have more inputs than you actually need to fully implement all instructions for this lab.

When reset is asserted (remember that `rst_n` is **active-low**), the FSM must enter the waiting state and stay there until `start` is asserted (see the overall CPU structure section above).

You will need to design the controller FSM state sequences for each instruction based on table 1 and your knowledge of the datapath. However, your controller must be capable of controlling any datapath that correctly implements the datapath spec. For full marks, you should implement each instruction in the minimum number of cycles possible (e.g., move-immediate needs fewer cycles than add); regardless, in no event should any instruction take more than five 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).

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

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 lab 5, 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 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.
- 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: Datapath changes [1 mark]

Implement the modified datapath as described in the Specification section, synthesize it, and thoroughly test it. For both the RTL and the testbench, it's easiest to start with your datapath from lab 5 and modify it; ensure that any select inputs are exactly as in the spec. Note that your testbench must *fully* test the new datapath, not just the modified parts.

Handin deliverables: `datapath.sv`, `datapath.vo`, `tb_datapath.sv`, `alu.sv`, `shifter.sv`, `regfile.sv`.

Task 2: Instruction decoder [3 marks]

Implement the instruction decoder described in the Specification section, synthesize it, and thoroughly test it. Be sure that any select inputs are exactly as in the spec.

Handin deliverables: `idecoder.sv`, `idecoder.vo`, `tb_idecoder.sv`.

Task 3: Controller FSM [3 marks]

Implement the instruction decoder described in the Specification section, synthesize it, and thoroughly test it.

Handin deliverables: `controller.sv`, `controller.vo`, `tb_controller.sv`.

Task 4: CPU [3 marks]

Build a complete Potato Machine™ CPU as described in the Specification section, instantiating the datapath, decoder, and controller you built. Make sure that this module interacts with each of these only via their specified interfaces; we may replace them with our own versions for testing. As before, synthesize your design, and create a thorough testbench.

Handin deliverables: `cpu.sv`, `cpu.vo`, `tb_cpu.sv`.

ADVICE

Implement one instruction at a time. In this lab, the hardest part is the controller FSM. If you try to implement everything at once and it turns out that it doesn't work right away, you will have a debugging nightmare. It's *much* easier to build the FSM via iterative refinement: implement one instruction, test it, implement the next instruction, and so on. Start from the move-immediate instruction; it does not need multiple cycles so it's easiest to implement and test.

Debugging technique. Review the lab 3 and lab 5 advice sections for debugging hints.