CPEN 211 2022W1

LAB 5

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

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 "Lab5" 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 lab 3, 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:

- `tb_regfile.sv`
- `tb_alu.sv`
- `alu.sv`
- `alu.vo`
- `tb_shifter.sv`
- `shifter.sv`
- `shifter.vo`
- `tb_datapath.sv`
- `datapath.sv`
- `datapath.vo`
- `CONTRIBUTIONS.txt` (empty if working alone)

Refer to the Specification section on page 2 and the Tasks section on page 6 for the requirements for each deliverable.

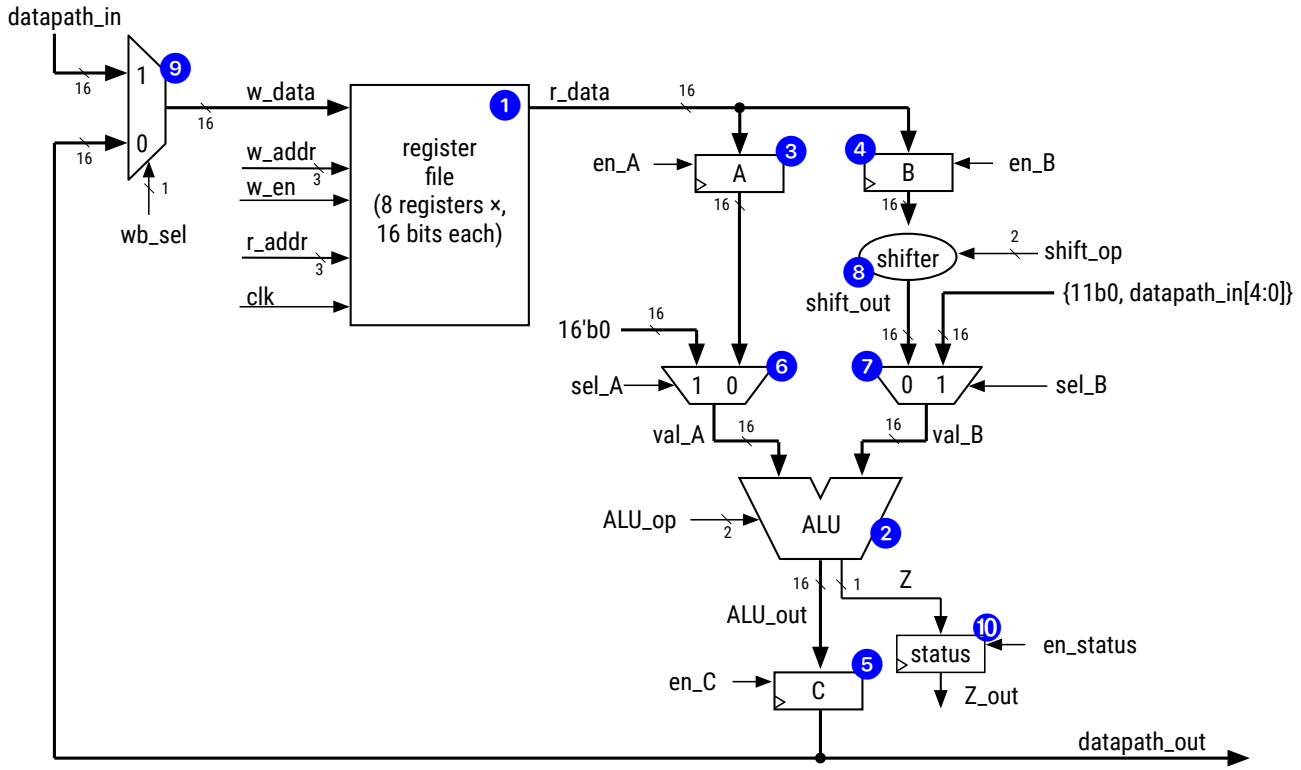

Figure 1: Potato Machine™ Datapath

SPECIFICATION

In the previous lab, you learned how to program the ARMv7 CPU inside your DE1-SoC. In the next few labs, we will build a Potato Machine™ CPU that implements an instruction set architecture (ISA) similar to a very small subset of the ARM ISA. Unlike the real ARMv7 ISA, both data and instructions will be 16-bit.

The Potato Machine™ instructions are shown in appendix A. Even though the ISA is very simple, it is Turing-complete, meaning that you can write all possible programs using only those instructions.

In this lab, we will not worry about how the instructions are encoded. Rather, we will focus on designing a **datapath** that can do all computations supported by the Potato Machine™ ISA (fig. 1). In later labs, you will implement the control logic

To start you off, we will provide an implementation of the Potato Machine™ register file (RF), which it will be your job to properly test. Then, you will design and test the ALU and the shifter. Finally, you will connect all the components to create the overall datapath.

The Potato Machine™ datapath elements

The overall Potato Machine™ datapath is shown in fig. 1. It comprises the following components:

- one register file 1 containing 8 registers, each holding 16 bits;
- a shifter unit 8;
- an arithmetic logic unit (ALU) 2.
- three multiplexers 6 7 9;
- three 16-bit registers with enable* 3 4 5; and

*Refer to the RTL lectures if you are not sure how a register with an enable can be implemented.

- one 1-bit register with enable (10).

Each component is described in detail below.

The register file (RF)

The Potato Machine™ has eight 16-bit registers, `r0` through `r7`. The RF is a small memory that stores the current values of these registers, and allows them to be read and written as each instruction requires.

The registers are accessed by their *address*, corresponding to the register number; that is, `r0` is at address 0, `r5` is at address 5, and so on. Because there are eight registers, an address in the RF will be $\log_2 8 = 3$ bits.

In our RF, addresses can be specified separately for reading (as `r_addr`) and for writing (as `w_addr`) — we say that the RF has two ports: one read port, and one write port.

To **read** a register in the RF, we provide the register address (e.g., 5 for `R5`) on the 3-bit `r_addr` bus; the register's current value will then appear on the 16-bit `r_data`. RF reads are *combinational*, so they are not coordinated with the clock input.

To **write** a register in the RF, we provide the register address on the `w_addr` bus, the value to be written on `w_data`, and set the write enable `w_en` to high. The value on `w_data` will be stored in the register identified by `w_addr` on the next rising edge of the clock — that is, RF writes are *sequential*.

We have provided the RF design for you in `regfile.sv`. In real processors, RFs are implemented as SRAMs, but in this case we have built ours out of flip-flop registers.

In the datapath, registers `A` (3) and `B` (4) will temporarily store values we read from the RF as an instruction is being executed.

The Arithmetic-Logic Unit (ALU)

The ▽ symbol labelled (2) in fig. 1 represents an *arithmetic logic unit* (ALU). This is the main hardware element that actually computes things inside the Potato Machine™.

The type of operation that can be performed by the ALU is indicated by the binary value on the `ALU_op` input as shown in table 1:

`ALU_op`	operation	`ALU_out`
`00`	addition	`val_A + val_B`
`01`	subtraction	`val_A - val_B`
`10`	bitwise AND	`val_A & val_B`
`11`	bitwise negation	`~val_B`

Table 1: ALU operations

The ALU is purely **combinational** — there is no clock input. Whenever one of the inputs or `ALU_op` change, the output changes appropriately regardless of the clock.

In addition to the value of the arithmetic operation produced on `ALU_out`, the ALU produces a single-bit status output that is true if and only if `ALU_out` is equal to zero. (We will need this output in later labs to implement conditional branches.)

Once the ALU finished computing, its 16-bit result is captured in register `C` (5), and the operation status (i.e., whether the output was zero) is captured in register `status` (10).

Source operand muxes

Each of the inputs to the ALU can come from two sources, selected by muxes (6) and (7) in fig. 1. In later labs, we will drive the select inputs of these muxes to implement various Potato Machine™ instructions:

- for some instructions, we'll use these multiplexers are ensure that `val_A` is zero;
- for some instructions, we'll use them to shift `val_B` before it reaches the ALU (see below); and
- for some instructions, we'll use them to input an immediate operand as `val_B`.

Shifter unit

The shifter unit ((8) in fig. 1) is another purely combinational logic block; it allows one of the ALU sources to be shifted before the ALU operation.

The shifter takes one 16-bit input from register B ((4) in fig. 1) and outputs either the same value, or the value shifted one bit to the left or right according the two-bit value on `shift_op` as specified in table 2:

`shift_op`	operation	output
`00`	no shift	`B`
`01`	left shift	`B` shifted left by one bit, with the LSB equal to 0
`10`	logical right shift	`B` shifted right by one bit, with the MSB equal to 0
`11`	arithmetic[†] right shift	`B` shifted right by one bit, with the MSB a copy of `B[15]`

Table 2: Shift operation encoding

For example, suppose the value stored in `B` were `1111000011001111`. Then, depending on `shift_op`, the output of the shifter would be as shown in Table 3:

`shift_op`	output of shifter
`00`	`1111000011001111`
`01`	`1110000110011110`
`10`	`0111100001100111`
`11`	`1111100001100111`

Table 3: Example shift operations starting with `1111000011001111`

Writeback multiplexer

The writeback multiplexer ((9) in Figure 1) selects the value that will be written to the register identified by `w_addr`: it can either be the ALU result (stored in register `C`) or directly from the datapath input.

Datapath operation walkthrough

In this section we will look at how the instruction `ADD R5, R2, R3` is executed by the Potato Machine™ datapath. This instruction adds the values in two registers, `R2` and `R3`, and stores the result in `R5`.

Note that you do not need to implement the control unit for this operation, just the datapath. This walkthrough example is only intended to help you understand how the datapath works.

The execution of this instruction takes four cycles:

cycle 1 loads `A` from the RF:

[†]Why is the right shift that fills the new MSB with the previous MSB called arithmetic? If you're not sure, review the two's complement lecture.

- `r_addr` is 2 to indicate we want to read the 16-bit contents of `R2` from the RF;
- `en_A` is 1 to indicate that register `A` should be updated on the next rising edge of the clock with the value read from the RF;
- `en_B` is 0;
- `w_en` is 0 to avoid writing junk to the RF.

cycle 2 loads `B` from the RF:

- `r_addr` is 3 to indicate we want to read the contents of `R3` from the RF;
- `en_A` is 0;
- `en_B` is 1 to load the RF output into `B`;
- `w_en` is 0 to avoid writing junk to the RF.

cycle 3 performs the computation:

- `en_A` and `en_B` are 0;
- `ALU_op` is `00` to indicate addition (see table 1);
- `sel_A` is 0 to ensure that the left ALU input comes from register `A`;
- `sel_B` is 0 to ensure that the right ALU input comes from `B`;
- `shift_op` is `00` to indicate the value read from `B` should *not* be shifted;
- `en_C` is 1 to capture the ALU result in register `C`;
- `en_status` is 1 to capture the ALU zero status in `status`;
- `w_en` is 0 to avoid writing junk to the RF.

cycle 4 writes back the result to the RF:

- `en_A`, `en_B`, `en_C`, and `en_status` are 0;
- `w_en` is 1 to indicate that we want to write into the RF;
- `wb_sel` is 0 to indicate that the written value is coming from `C`;
- `w_addr` is 5 to indicate that the value should be written to `R5`.

Not all Potato Machine™ instructions need four cycles. To check your understanding of the datapath, think about how the `MOV` instruction with an immediate operand could execute in just one cycle.

Testbenches

As part of the lab, you will be developing unit tests[‡] for the register file, the ALU, the shifter, and the entire datapath. 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 1 if any of the tests failed.

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.

[‡]If you don't remember how to write decent testbenches, the vending machine code from lecture has a reasonable example.

TASKS and DELIVERABLES

Common requirements

- All RTL and testbenches must be in (System)Verilog.
- All sequential logic must be triggered on the rising edge of input `clk`.
- The (System)Verilog you write for each task **must** be entirely in the skeleton file provided for that task.
- Your **must not** modify the module names, port names, or port types / declarations in the skeleton files. 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: RF testbench [1 mark]

In the file `tb_regfile.sv`, develop a testbench for the register file you were given in `regfile.sv`, based on the Specification section above

Handin deliverables: `tb_regfile.sv`.

Task 2: ALU testbench [1 mark]

In the file `tb_alu.sv`, develop a testbench for the ALU, based on the Specification section above. For full marks, your tesbench must automatically check and report whether each test case succeeds or fails, and must correctly report test failures on the `err` port.

Handin deliverables: `tb_alu.sv`.

Task 3: ALU design [2 marks]

In the file `alu.sv`, develop the ALU based on the Specification section above. Verify it with your testbench. Synthesize your design in Quartus to obtain a `.vo` file as in lab 3, and verify the netlist.

You **must** use the operation encoding in table 1 while designing your ALU; otherwise you will receive no credit for your implementation.

Handin deliverables: `alu.sv` and `alu.vo`.

Task 4: shifter testbench [1 mark]

In the file `tb_shifter.sv`, develop a testbench for the shifter, based on the Specification section above. For full marks, your tesbench must automatically check and report whether each test case succeeds or fails, and must correctly report test failures on the `err` port.

Handin deliverables: `tb_shifter.sv`.

Task 5: shifter design [2 marks]

In the file `shifter.sv`, develop the shifter based on the Specification section above. Verify it with your testbench. Synthesize your RTL and verify the netlist.

You **must** use the operation encoding in table 2 while designing your shifter; otherwise you will receive no credit for your implementation.

Handin deliverables: `shifter.sv` and `shifter.vo`.

Task 6: full datapath testbench [1 mark]

In the file `tb_datapath.sv`, develop a testbench for the entire datapath, based on the Specification section above.

Handin deliverables: `tb_datapath.sv`.

Task 7: full datapath design [2 marks]

In the file `datapath.sv`, develop the full datapath based on the Specification section above, instantiating the RF we provided as well as the ALU and shifter you designed. Verify the full datapath with your testbench, then synthesize your RTL and verify the netlist.

Your testbench must instantiate the ALU and shifter using the specified interface — for example, we may test your datapath by instantiating it with our own implementations of the shifter and ALU.

Handin deliverables: `datapath.sv` and `datapath.vo`.

ADVICE

- Do the tasks one at a time.
- Break down task 7 into smaller subtasks that you **test separately**.
- Develop by **iterative refinement**: implement some one part, test, implement more, test more, and so forth.
- To debug efficiently, learn how to use Modelsim to **trace where signals are being driven from**. See the (short) CPEN311 debugging video on Canvas if you don't know how to do this.
- Don't try to fix things unless you **know for sure** what has gone wrong. It's easy to keep guessing for a very long time and get nowhere. If you don't know what's wrong, keep tracing the signals in Modelsim.
- If you attempt to fix a bug and the bug is still there, **undo your "fix"** (or comment it out). Otherwise you will keep introducing new bugs as you try to fix old ones.
- Run **all of your tests** whenever you make *any* changes to make sure you haven't broken anything. (This is called regression testing.)
- You may want to use a revision control system (like `git`) to keep track of your development. If you do so, however, make sure the repository is **private** to your group to avoid violating the academic integrity policy.

A The Potato Machine™ Instruction Set Architecture

The information in Table 4 and 5 is only relevant to later labs. An assembler will be provided to you for these later labs. Each row in these tables 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 handouts for the later labs.

- `Rn`, `Rd`, `Rm` are 3-bit register number specifiers.
- `im8` is an 8-bit immediate operand encoded as part of the instruction.
- `im5` is a 5-bit immediate operand encoded as part of the instruction.
- `sh` is a 2-bit immediate encoded as part of the instruction used to control the shifter. Legal values for `sh_op` are "`LSL#1`", "`LSR#1`", or "`ASR#1`".
- `sx(f)` sign extends the immediate value `f` to 16-bits.
- `sh_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.
- `M[x]` is the 16-bit value stored in main memory (added in later labs) at address x.
- `PC` refers to the program counter register (added in later labs).
- `<label>` refers to a textual marker in the assembly that indicates an instruction address

Assembly syntax (see text)	Potato 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*		*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`
memory instructions	*opcode*			*ALUop*		*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]`

Table 4: The Potato Machine™ data processing and data movement instructions

Assembly syntax (see text)	Potato Machine™ 16-bit encoding																Operation (see text)
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
branches	*opcode*			*op*		*cond*			*8b*								
`B <label>`	0	0	1	0	0	0	0	0	im8								`PC = PC+1+sx(im8)`
`BEQ <label>`	0	0	1	0	0	0	0	1	im8								`if Z = 1 then` `  PC = PC+1+sx(im8)` `else` `  PC = PC+1`
`BNE <label>`	0	0	1	0	0	0	1	0	im8								`if Z = 0 then` `  PC = PC+1+sx(im8)` `else` `  PC = PC+1`
`BLT <label>`	0	0	1	0	0	0	1	1	im8								`if N != V then` `  PC = PC+1+sx(im8)` `else` `  PC = PC+1`
`BLE <label>`	0	0	1	0	0	1	0	0	im8								`if N!=V or Z=1 then` `  PC = PC+1+sx(im8)` `else` `  PC = PC+1`
u call	*opcode*			*op*		*Rn*			*8b*								
`BL <label>`	0	1	0	1	1	1	1	1	im8								`R7=PC+1; PC=PC+1+sx(im8)`
return	*opcode*			*op*		*unused*			*Rd*			*unused*					
`BX Rd`	0	1	0	0	0	0	0	0	Rd			0	0	0	0	0	`PC=Rd`
indirect call	*opcode*			*op*		*Rn*			*Rd*			*unused*					
`BLX Rd`	0	1	0	1	0	1	1	1	Rd			0	0	0	0	0	`R7=PC+1; PC=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 5: The Potato Machine™ control instructions