Project 3: CS61CPU

Part A Deadline: Thursday, April 3, 11:59:59 PM PT

Part B Deadline: Thursday, April 17, 11:59:59 PM PT

In this project, you will be building a CPU that runs actual RISC-V instructions.

Setup

You must complete this lab on your local machine. See Lab 0 if you need to set up your local machine again.

For common errors with Logisim or Project 3-specific errors, please look at the common errors page.

Setup: Git

This assignment can be done alone or with a partner.

Warning: Once you create a group on Gradar, you will *not* be able to change (add, remove, or swap) partners for this project (both Project 3A and 3B), so please be sure of your partner before starting the project. You must add your partner on both Gradar and to every Gradescope submission.

If there are extenuating circumstances that require a partner switch (e.g. your partner drops the class, your partner is unresponsive), please reach out to us privately.

1. **Visit** Gradar . **Log in** and **register** your Project 3 group (and **add** your partner, if you have one), then **create** a GitHub repo for you or your group. If you have a partner, one partner should create a group and invite the other partner to that repo. The other partner should accept the invite without creating their own group.
2. **Clone** the repository on your workspace. Please use your **local machine** (you don't need the hive machine at all for this project). Windows users should clone outside WSL (Git Bash

is recommended).

```
git clone git@github.com:61c-student/sp25-proj3-USERNAME.git 61c-proj3
```

(**replace** `USERNAME` with your GitHub username)

3. **Navigate** to your repository:

```
cd 61c-proj3
```

4. **Add the starter repo** as a remote:

```
git remote add starter https://github.com/61c-teach/sp25-proj3-starter.git
```

If you run into `git` issues, please check out the common errors page.

Setup: Logisim

This project is done in Logisim. In the `61c-proj3` directory, **run** **`bash test.sh download_tools`** to download Venus and Logisim for this project. (You only need to run this once.)

For the rest of the project, to open Logisim, **run** **`java -jar tools/logisim-evolution.jar`**.

Restoring Starter Files

To restore starter files, please check out the common errors page.

Part A: addi

> Lab 5 is required for Project 3A, and lectures 23-27, Discussion 8, and Homework 7 are highly recommended.

In this part, you will design a skeleton CPU that can execute the `addi` instruction.

Task 1: Arithmetic Logic Unit (ALU)

Fill in the ALU in `alu.circ` so that it can perform the required arithmetic calculations.

Input Name	Bit Width	Description
`A`	32	Data to use for Input A in the ALU operation
`B`	32	Data to use for Input B in the ALU operation
`ALUSel`	4	Selects which operation the ALU should perform (see the list of operations with corresponding switch values below)

Output Name	Bit Width	Description
`ALUResult`	32	Result of the ALU operation

Below is the list of ALU operations for you to implement, along with their associated ALUSel values. `add` is already made for you. You are allowed and encouraged to use built-in Logisim components to implement the arithmetic operations.

ALUSel Value	Instruction
0	add: `Result = A + B`
1	sll: `Result = A << B[4:0]`
2	slt: `Result = (A < B (signed)) ? 1 : 0`

3	Unused
4	xor: `Result = A ^ B`
5	srl: `Result = (unsigned) A >> B[4:0]`
6	or: `Result = A | B`
7	and: `Result = A & B`
8	mul: `Result = (signed) (A * B)[31:0]`
9	mulh: `Result = (signed) (A * B)[63:32]`
10	Unused
11	mulhu: `Result = (A * B)[63:32]`
12	sub: `Result = A - B`
13	sra: `Result = (signed) A >> B[4:0]`
14	Unused
15	bsel: `Result = B`

Some additional tips:

- When performing shifts, only the lower 5 bits of `B` are needed, because only shifts of up to 32 are supported.
- The result of multiplying 2 32-bit numbers can be up to 64 bits of information, but we're limited to 32-bit data lines, so `mulh` and `mulhu` are used to get the upper 32 bits of the product. The `Multiplier` component has a `Carry Out` output, with the description: "the upper bits of the product". This might be particularly useful for certain multiply operations.
- The comparator component might be useful for implementing instructions that involve comparing inputs.
- A multiplexer (MUX) might be useful when deciding between operation outputs. In other words, consider simply processing the input for all operations, and then outputting the one

of your choice.

- The ALU tests for Part A only use ALUSel values for defined instructions, so your design doesn't need to worry about the unused values.

Testing

On your **local machine**, start by **running `bash test.sh`** in the `61c-proj3` directory on your local machine. This gives you an overview of the commands you can run for testing. In particular, `bash test.sh part_a` runs all the tests for Part A. You can also provide the name of a specific task to run all the tests for that particular task.

To test this task, on your **local machine**, **run `bash test.sh test_alu`**.

If you fail a test, the test runner will print the difference between the expected and actual output. To view the complete reference output (`.ref` file) and your output (`.out` file), you can use run `bash test.sh format` with the name of the output file. For this task:

```
bash test.sh format tests/unit-alu/out/alu-add.ref
bash test.sh format tests/unit-alu/out/alu-add.out

bash test.sh format tests/unit-alu/out/alu-all.ref
bash test.sh format tests/unit-alu/out/alu-all.out

bash test.sh format tests/unit-alu/out/alu-logic.ref
bash test.sh format tests/unit-alu/out/alu-logic.out

bash test.sh format tests/unit-alu/out/alu-mult.ref
bash test.sh format tests/unit-alu/out/alu-mult.out

bash test.sh format tests/unit-alu/out/alu-shift.ref
bash test.sh format tests/unit-alu/out/alu-shift.out

bash test.sh format tests/unit-alu/out/alu-slt-sub-bsel.ref
bash test.sh format tests/unit-alu/out/alu-slt-sub-bsel.out
```

Debugging

See the Testing and Debugging section for a more detailed debugging guide.

All the testing `.circ` circuit files are in the `tests` folder. These circuits feed a sequence of inputs to your ALU circuit (one per clock cycle) and records the outputs from your circuit.

In Logisim, **open one of the testing circuits** for this task:

```
tests/unit-alu/alu-add.circ
tests/unit-alu/alu-all.circ
tests/unit-alu/alu-logic.circ
tests/unit-alu/alu-mult.circ
tests/unit-alu/alu-shift.circ
tests/unit-alu/alu-slt-sub-bsel.circ
```

To view your circuit, **right-click your ALU**, and **select** `View alu`. To step through the inputs to your circuit at each time step, **click** `File -> Manual Tick Full Cycle`. As you step through the inputs, **use the Poke Tool** to check the values in each wire.

Note: Avoid making edits in the test circuit, as they may be lost!

Task 2: Register File (RegFile)

Fill in `regfile.circ` so that it contains 32 registers that can be written to and read from.

Input Name	Bit Width	Description
`ReadIndex1`	5	Determines which register's value is sent to the `ReadData1` output
`ReadIndex2`	5	Determines which register's value is sent to the `ReadData2` output
`WriteIndex`	5	The register to write to on the next rising edge of the clock (if `RegWEn` is 1)
`WriteData`	32	The data to write into `rd` on the next rising edge of the clock (if `RegWEn` is 1)
`RegWEn`	1	Determines whether data is written to the register file on the next rising edge of the clock

`clk`	1	Clock input

Output Name	Bit Width	Description
`ReadData1`	32	The value of the register identified by `ReadIndex1`
`ReadData2`	32	The value of the register identified by `ReadIndex2`
`ra`	32	The value of `ra` (`x1`)
`sp`	32	The value of `sp` (`x2`)
`t0`	32	The value of `t0` (`x5`)
`t1`	32	The value of `t1` (`x6`)
`t2`	32	The value of `t2` (`x7`)
`s0`	32	The value of `s0` (`x8`)
`s1`	32	The value of `s1` (`x9`)
`a0`	32	The value of `a0` (`x10`)

- The 8 constant output registers are included in the output of the `regfile` circuit for testing and debugging purposes. Make sure to connect these 8 output pins to their corresponding registers.
- The `x0` register should always contain the 0 value, even if an instruction tries writing to it.

Some additional tips:

- Take advantage of copy-paste! It might be a good idea to make one register completely and use it as a template for the others to avoid repetitive work. You can duplicate a selected component or group of components in Logisim using `Ctrl/Cmd + D`.
- The `Enable` pin on the built-in register may come in handy.

Testing and Debugging

To test your function, in your local terminal, run `bash test.sh test_regfile`.

To view the reference output and your output, you can run these formatting commands:

```
bash test.sh format tests/unit-regfile/out/regfile-more-regs.ref
bash test.sh format tests/unit-regfile/out/regfile-more-regs.out

bash test.sh format tests/unit-regfile/out/regfile-read-only.ref
bash test.sh format tests/unit-regfile/out/regfile-read-only.out

bash test.sh format tests/unit-regfile/out/regfile-read-write.ref
bash test.sh format tests/unit-regfile/out/regfile-read-write.out

bash test.sh format tests/unit-regfile/out/regfile-x0.ref
bash test.sh format tests/unit-regfile/out/regfile-x0.out
```

To debug your circuit, open the following test circuits, click into your regfile circuit, and tick full cycles to step through inputs:

```
tests/unit-regfile/regfile-more-regs.circ
tests/unit-regfile/regfile-read-only.circ
tests/unit-regfile/regfile-read-write.circ
tests/unit-regfile/regfile-x0.circ
```

Task 3: Immediate Generator

For the rest of Part A, we will be creating just enough of the CPU to execute the `addi` instruction. In Part B, you will revisit these circuits and expand them to support more instructions.

Fill in the immediate generator in `imm-gen.circ` (not the `imm_gen` subcircuit in `cpu.circ`) so that it can generate immediates for the `addi` instruction. You can ignore other immediate types for now.

Input Name	Bit Width	Description
`Instruction`	32	The instruction being executed

`ImmSel`	3	Value determining how to reconstruct the immediate (you can ignore this for now)

Output Name	Bit Width	Description
`Immediate`	32	Value of the immediate in the instruction (assume the instruction is `addi` for now)

Testing and Debugging

You'll have to complete the next task before debugging this one!

Task 4: Datapath

Fill in `cpu.circ` so that it contains a datapath for a single-cycle (not pipelined) processor that can execute the `addi` instruction.

Here are the inputs and outputs to the processor. You can leave most of them unchanged in this task, since they are not needed for the `addi` instruction.

Input Name	Bit Width	Description
`MemReadData`	32	Data at `MemAddress` from memory
`Instruction`	32	The instruction at memory address `ProgramCounter`
`clk`	1	Clock input

Output Name	Bit Width	Description
`ra`	32	The value of `ra` (`x1`)
`sp`	32	The value of `sp` (`x2`)
`t0`	32	The value of `t0` (`x5`)

`t1`	32	The value of `t1` (`x6`)
`t2`	32	The value of `t2` (`x7`)
`s0`	32	The value of `s0` (`x8`)
`s1`	32	The value of `s1` (`x9`)
`a0`	32	The value of `a0` (`x10`)
`MemAddress`	32	The address in memory to read from or write to
`MemWriteData`	32	Data to write to memory
`MemWriteMask`	4	The write enable mask for writing data to memory
`ProgramCounter`	32	Address of the `Instruction` input

We know that trying to build a datapath from scratch might be intimidating, so the rest of this section offers more detailed guidance for creating your processor.

Recall the five stages for executing an instruction:

1. Instruction Fetch (IF)
2. Instruction Decode (ID)
3. Execute (EX)
4. Memory (MEM)
5. Write Back (WB)

Task 4.1: Instruction Fetch

We have already provided a simple implementation of the program counter. It is a 32-bit register that increments by 4 on each clock cycle. The `ProgramCounter` is connected to IMEM (instruction memory), and the `Instruction` is returned from IMEM.

Nothing for you to implement in this sub-task!

Task 4.2: Instruction Decode

In this step, we need to break down the `Instruction` input and send the bits to the right subcircuits.

▸ What type of instruction is `addi`? What are the different fields in the instruction, and which bits correspond to each field?

▸ In Logisim, what tool would you use to split out different groups of bits?

▸ Which fields should connect to the register file? Which inputs of the register file should they connect to?

▸ What needs to be connected to the immediate generator?

Task 4.3: Execute

In this step, we will use the decoded instruction fields to compute the actual instruction.

▸ What two data values (`A` and `B`) should the `addi` instruction input to the ALU?

▸ What `ALUSel` value should the instruction input to the ALU?

Task 4.4: Memory

The `addi` instruction doesn't use memory, so there's nothing for you to implement in this sub-task!

The memory stage is where the memory can be written to using store instructions and read from using load instructions. Because the `addi` instruction does not use memory, we do not have to worry about it for Part A. Please ignore the DMEM and leave its I/O pins undriven.

Task 4.5: Write Back

In this step, we will write the result of our `addi` instruction back into a register.

▸ What data is the `addi` instruction writing, and where is the instruction writing this data to?

Testing and Debugging

See the Testing and Debugging section for a more detailed debugging guide.

To test your function, in your local terminal, run `bash test.sh test_addi`.

To view the reference output and your output, you can run these formatting commands:

```
bash test.sh format tests/integration-addi/out/addi-basic.ref
bash test.sh format tests/integration-addi/out/addi-basic.out

bash test.sh format tests/integration-addi/out/addi-negative.ref
bash test.sh format tests/integration-addi/out/addi-negative.out

bash test.sh format tests/integration-addi/out/addi-positive.ref
bash test.sh format tests/integration-addi/out/addi-positive.out
```

To debug your circuit, open the following test circuits, click into your CPU circuit, and tick full cycles to step through inputs:

```
tests/integration-addi/addi-basic.circ
tests/integration-addi/addi-positive.circ
tests/integration-addi/addi-negative.circ
```

Submission and Grading

Submit your repository to the Project 3A assignment on Gradescope. The autograder tests for Part A are the same as the tests you are running locally. Part A is worth 20% of your overall Project 3 grade.

- ALU (7)
- RegFile (8)
- `addi` (5)

Total: 20 points

Part B: More Instructions

> Lab 6 is required for Project 3B, and lectures 23-27, Discussion 8-9, and Homework 7 are highly recommended.

In this part, you will expand your CPU to support more instructions and pipelining.

Before starting this part, please **run** `git pull starter main` to download any changes we may have pushed. If you see an error about "unrelated histories" run `git fetch starter main-fix && git merge starter/main-fix` instead.

You can implement the instructions in any order you want. This spec is organized to build up your CPU with small groups of instructions at a time.

Note: Your circuit may not rely on floating (undefined) values. While tests may pass locally, it will violate design rule checks on Gradescope.

Task 5: I-type Instructions

The instructions you need to implement for this task are listed below:

Instruction	Type	Opcode	Funct3	Funct7	Operation
`addi rd, rs1, imm`	I	`0x13`	`0x0`		`rd = rs1 + imm`
`andi rd, rs1, imm`			`0x7`		`rd = rs1 & imm`
`ori rd, rs1, imm`			`0x6`		`rd = rs1 | imm`
`xori rd, rs1, imm`			`0x4`		`rd = rs1 ^ imm`
`slli rd, rs1, imm`	I*		`0x1`	`0x00`	`rd = rs1 << imm`
`srli rd, rs1, imm`			`0x5`	`0x00`	`rd = rs1 >> imm` (Zero-extend)
`srai rd, rs1, imm`			`0x5`	`0x20`	`rd = rs1 >> imm` (Sign-extend)

`slti rd, rs1, imm`	I		`0x2`		`rd = (rs1 < imm) ? 1 : 0`

Task 5.1: Datapath

Recall that you already implemented `addi` in Part A. Other I-type instructions use the same datapath as `addi`, except that each I-type instruction needs the ALU to perform a different operation. In Part A, we hard-coded the `ALUSel` input to the ALU subcircuit to be `0b0000` so that the ALU always performs the addition selection, but now you should **change `ALUSel` input to the ALU subcircuit** to use the value from the control logic subcircuit (which you'll implement in the next task).

Remember to also **change the `RegWEn` input to the regfile subcircuit** to use the value from the control logic subcircuit.

Task 5.2: Control Logic

As you add logic to support more instructions in the next few tasks, you will need to add control logic to enable the relevant datapath components depending on the instruction being executed.

Modify `control_logic.circ` to output the correct control logic signals for I-type instructions. See the control logic section for more details.

Testing and Debugging

We don't have any provided tests for I-type instructions, so you'll need to write your own tests to find bugs in your implementation. Before requesting help from staff, please make sure you have some tests written, or we'll ask you to write some tests first before helping you.

1. Navigate to `tests/integration-custom/in`.
2. Write a RISC-V test and save it in a filename ending in `.s`.
3. Run `bash test.sh test_custom`.

`test_custom` compiles your RISC-V test code to a Logisim circuit and runs it. If you want to only compile your test, run `bash test.sh create_custom`. If you want to only run your test, run `bash test.sh run_custom`.

To debug your circuits, you can step through the debugging circuits (similar to what you did in Project 3A).

1. Navigate to the `tests` folder, then navigate to the folder of the relevant test, e.g. `tests/integration-custom`.
2. Open the generated `.circ` file in Logisim. Click into the circuits you made, and tick full cycles to step through inputs.

Task 6: R-type Instructions

The instructions you need to implement for this task are listed below:

Instruction	Type	Opcode	Funct3	Funct7	Operation
`add rd, rs1, rs2`	R	`0x33`	`0x0`	`0x00`	`rd = rs1 + rs2`
`sub rd, rs1, rs2`			`0x0`	`0x20`	`rd = rs1 - rs2`
`and rd, rs1, rs2`			`0x7`	`0x00`	`rd = rs1 & rs2`
`or rd, rs1, rs2`			`0x6`	`0x00`	`rd = rs1 | rs2`
`xor rd, rs1, rs2`			`0x4`	`0x00`	`rd = rs1 ^ rs2`
`sll rd, rs1, rs2`			`0x1`	`0x00`	`rd = rs1 << rs2`
`srl rd, rs1, rs2`			`0x5`	`0x00`	`rd = rs1 >> rs2` (Zero-extend)
`sra rd, rs1, rs2`			`0x5`	`0x20`	`rd = rs1 >> rs2` (Sign-extend)
`slt rd, rs1, rs2`			`0x2`	`0x00`	`rd = (rs1 < rs2) ? 1 : 0`
`mul rd, rs1, rs2`			`0x0`	`0x01`	`rd = (rs1 * rs2)[31:0]`
`mulh rd, rs1, rs2`			`0x1`	`0x01`	`rd = (rs1 * rs2)[63:32]` (Signed)
`mulhu rd, rs1, rs2`			`0x3`	`0x01`	`rd = (rs1 * rs2)[63:32]` (Unsigned)

Task 6.1: Datapath

Modify your datapath in `cpu.circ` so that it can support R-type instructions.

If you're stuck, read further for some guiding questions. As with Task 4, it may help to think about each of the five stages for executing an instruction.

▸ Instruction Fetch: How do R-type instructions affect the program counter?

▸ Instruction Decode: What do we need to read from the register file?

▸ Execute: What two data values (`A` and `B`) should an R-type instruction input to the ALU?

▸ Memory: Do R-type instructions write to memory?

▸ Write back: What data is the R-type instruction writing, and where is the instruction writing this data to?

Task 6.2: Control Logic

Modify `control_logic.circ` to output the correct control logic signals for R-type instructions. See the control logic section for more details.

Testing and Debugging

We don't have any provided tests for R-type instructions, so you'll need to write your own tests to find bugs in your implementation. Before requesting help from staff, please make sure you have some tests written, or we'll ask you to write some tests first before helping you.

1. Navigate to `tests/integration-custom/in`.
2. Write a RISC-V test and save it in a filename ending in `.s`.
3. Run `bash test.sh test_custom`.

Task 7: B-type Instructions

The instructions you need to implement for this task are listed below:

Instruction	Type	Opcode	Funct3	Operation
`beq rs1, rs2, offset`	B	`0x63`	`0x0`	`if(rs1 == rs2) PC = PC + offset`
`bge rs1, rs2, offset`			`0x5`	`if(rs1 >= rs2 (signed)) PC = PC + offset`
`bgeu rs1, rs2, offset`			`0x7`	`if(rs1 >= rs2 (unsigned)) PC = PC + offset`
`blt rs1, rs2, offset`			`0x4`	`if(rs1 < rs2 (signed)) PC = PC + offset`
`bltu rs1, rs2, offset`			`0x6`	`if(rs1 < rs2 (unsigned)) PC = PC + offset`
`bne rs1, rs2, offset`			`0x1`	`if(rs1 != rs2) PC = PC + offset`

Task 7.1: Branch Comparator

Fill in the branch comparator subcircuit in `branch-comp.circ`. This subcircuit takes two inputs and outputs the result of comparing the two inputs. We will use the output later for implementing branches.

Signal Name	Direction	Bit Width	Description
`BrData1`	Input	32	First value to compare
`BrData2`	Input	32	Second value to compare
`BrUn`	Input	1	`1` when an **unsigned** comparison is wanted, and `0` when a

			signed comparison is wanted
`BrEq`	Output	1	Set to `1` if the two values are equal
`BrLt`	Output	1	Set to `1` if the value in rs1 is less than the value in rs2

We've provided some unit tests for the branch comparator subcircuit. These are not comprehensive. You can run these tests with `bash test.sh test_branch_comp`.

Task 7.2: Immediate Generator

Edit the immediate generator in `imm-gen.circ` so that it can generate immediates for B-type instructions in addition to immediates for I-type instructions (which you implemented in Part A).

Recall that the bits of the immediate are stored in different bits of the instruction, depending on the type of the instruction. The `ImmSel` signal, which you will implement in the control logic, will determine which type of immediate this subcircuit should generate.

The immediate storage formats are listed below:

Type	ImmSel (default)	Bits 31-20	Bits 19-12	Bit 11	Bits 10-5	Bits 4-1	Bit 0
I	`0b000`	`inst[31]`			`inst[30:20]`		
S	`0b001`	`inst[31]`			`inst[30:25]`	`inst[11:7]`	
B	`0b010`	`inst[31]`		`inst[7]`	`inst[30:25]`	`inst[11:8]`	`0`
U	`0b011`	`inst[31:12]`		`0`			
J	`0b100`	`inst[31]`	`inst[19:12]`	`inst[20]`	`inst[30:21]`		`0`

For this project, you may treat I*-type immediates as I-type immediates, since the ALU should only use the lowest 5 bits of the `B` input when computing shifts.

Recall that all immediates are 32 bits and sign-extended. (Sign extension is shown in the table as `inst[31]` repeated in the upper bits.)

We've provided some unit tests for the immediate generator subcircuit. These are not comprehensive. You can run these tests with `bash test.sh test_imm_gen`.

Note that if you only implement generating B-type immediates now, some tests for other immediate types will fail, but make sure that the `imm-gen-b-type` test passes.

The `ImmSel` values in the table represent the default encoding (mapping of `ImmSel` values to immediate types). If you choose to use a different encoding:

1. Navigate to `tests/unit-imm-gen`.
2. Open `imm-gen-encoding.csv`.
3. Replace the numbers with your selected encoding (in decimal). For example, if you're using `ImmSel = 0b110` to denote an I-type instruction, the second line should say `I,6`.
4. Run the unit tests with `bash test.sh test_imm_gen`.

Task 7.3: Datapath

Modify your datapath in `cpu.circ` so that it can support B-type instructions.

If you're stuck, read further for some guiding questions. As with Task 4, it may help to think about each of the five stages for executing an instruction.

▸ Instruction Fetch: How do B-type instructions affect the program counter?

▸ Instruction Decode: What do we need to read from the register file?

▸ Execute: What two data values (`A` and `B`) should an B-type instruction input to the ALU?

▸ Memory: Do B-type instructions write to memory?

▸ Write back: What data is the B-type instruction writing, and where is the instruction writing this data to?

Task 7.4: Control Logic

Modify `control-logic.circ` to output the correct control logic signals for B-type instructions.

See the control logic section for more details.

Testing and Debugging

We have provided some tests for B-type instructions. You can run them with:

```
bash test.sh test_integration_branch
```

These tests are not comprehensive, so you should write your own tests to find bugs in your implementation.

1. Navigate to `tests/integration-custom/in`.
2. Write a RISC-V test and save it in a filename ending in `.s`.
3. Run `bash test.sh test_custom`.

Task 8: Loading and Storing

The instructions you need to implement for this task are listed below:

Instruction	Type	Opcode	Funct3	Operation
`lb rd, offset(rs1)`	I	`0x03`	`0x0`	`rd =` 1 byte of memory at address `rs1 + imm`, sign-extended
`lh rd, offset(rs1)`			`0x1`	`rd =` 2 bytes of memory starting at address `rs1 + imm`, sign-extended
`lw rd, offset(rs1)`			`0x2`	`rd =` 4 bytes of memory starting at address `rs1 + imm`
`sb rs2, offset(rs1)`	S	`0x23`	`0x0`	Stores least-significant byte of `rs2` at the address `rs1 + imm` in memory
`sh rs2, offset(rs1)`			`0x1`	Stores the 2 least-significant bytes of `rs2` starting at the address `rs1 + imm` in memory
`sw rs2,`			`0x2`	Stores `rs2` starting at the address `rs1 + imm` in

`offset(rs1)`				memory

Task 8.1: Immediate Generator

Edit the immediate generator in `imm-gen.circ` so that it can generate immediates for S-type instructions in addition to all the instruction types from previous tasks. See the earlier immediate generator task for details.

We've provided some unit tests for the immediate generator subcircuit. These are not comprehensive. You can run these tests with `bash test.sh test_imm_gen`.

Note that if you only implement generating S-type immediates now, some tests for other immediate types will fail, but make sure that the `imm-gen-s-type` test passes.

Task 8.2: Partial Loads and Stores

See Partial Loads and Stores to implement this task.

Task 8.3: Datapath

With the help of the partial load and partial store circuits you've just made, modify your datapath in `cpu.circ` so that it can support loads and stores.

You should provide an address input `MemAddress` to DMEM. Remember that the ALU calculates this address by adding the address in `rs1` and the offset immediate.

You should also provide `MemWriteMask` and `MemWriteData` to DMEM. These are calculated by your partial load and partial store subcircuits.

For load instructions, you should also add functionality in the write-back stage so that the DMEM output data, processed by your partial load subcircuit, is written back to the `rd` register.

Task 8.4: Control Logic

Modify `control-logic.circ` to output the correct control logic signals for loads and stores. See the control logic section for more details.

Testing and Debugging

You'll need to write your own tests to find bugs in your implementation. Before requesting help from staff, please make sure you have some tests written, or we'll ask you to write some tests first before helping you.

1. Navigate to `tests/integration-custom/in`.
2. Write a RISC-V test and save it in a filename ending in `.s`.
3. Run `bash test.sh test_custom`.

We have provided some tests for load and store instructions, but they require `lui` to be implemented first. You can run them with:

```
bash test.sh test_integration_mem
```

Task 9: Jumps and U-type Instructions

The instructions you need to implement for this task are listed below:

Instruction	Type	Opcode	Funct3	Operation
`jal rd, imm`	J	`0x6f`		`rd = PC + 4` `PC = PC + offset`
`jalr rd, rs1, imm`	I	`0x67`	`0x0`	`rd = PC + 4` `PC = rs1 + imm`
`auipc rd, imm`	U	`0x17`		`rd = PC + imm`
`lui rd, imm`		`0x37`		`rd = imm`

Task 9.1: Immediate Generator

Edit the immediate generator in `imm-gen.circ` so that it can generate immediates for U-type instructions and J-type instructions. See the earlier immediate generator task for details.

We've provided some unit tests for the immediate generator subcircuit. These are not

comprehensive. You can run these tests with `bash test.sh test_imm_gen`.

Task 9.2: Datapath

Modify your datapath in `cpu.circ` so that it can support these instructions. Most of these instructions are already supported by your datapath so far.

Note that the U-type instructions require left-shifting the immediate by 12 bits (e.g. `lui` is written as `rd = imm << 12` on the reference card), but this should already be done by your immediate generator, so your datapath doesn't need to perform any extra shifting.

To support `jalr`, you should connect PC+4 to your multiplexer in the write-back stage so that PC+4 can be written back to `rd`.

Task 9.3: Control Logic

Modify `control_logic.circ` to output the correct control logic signals for jumps and U-type instructions. See the control logic section for more details.

Hint: Be careful about which ALU operation you're performing for the `lui` instruction. One of the ALU operations you made in Part A but didn't use anywhere else will come in handy here.

Testing and Debugging

We have provided some tests for jump instructions and `lui` (but not `auipc`). You can run them with:

```
bash test.sh test_integration_jump
bash test.sh test_integration_lui
```

These tests are not comprehensive, so you should write your own tests to find bugs in your implementation.

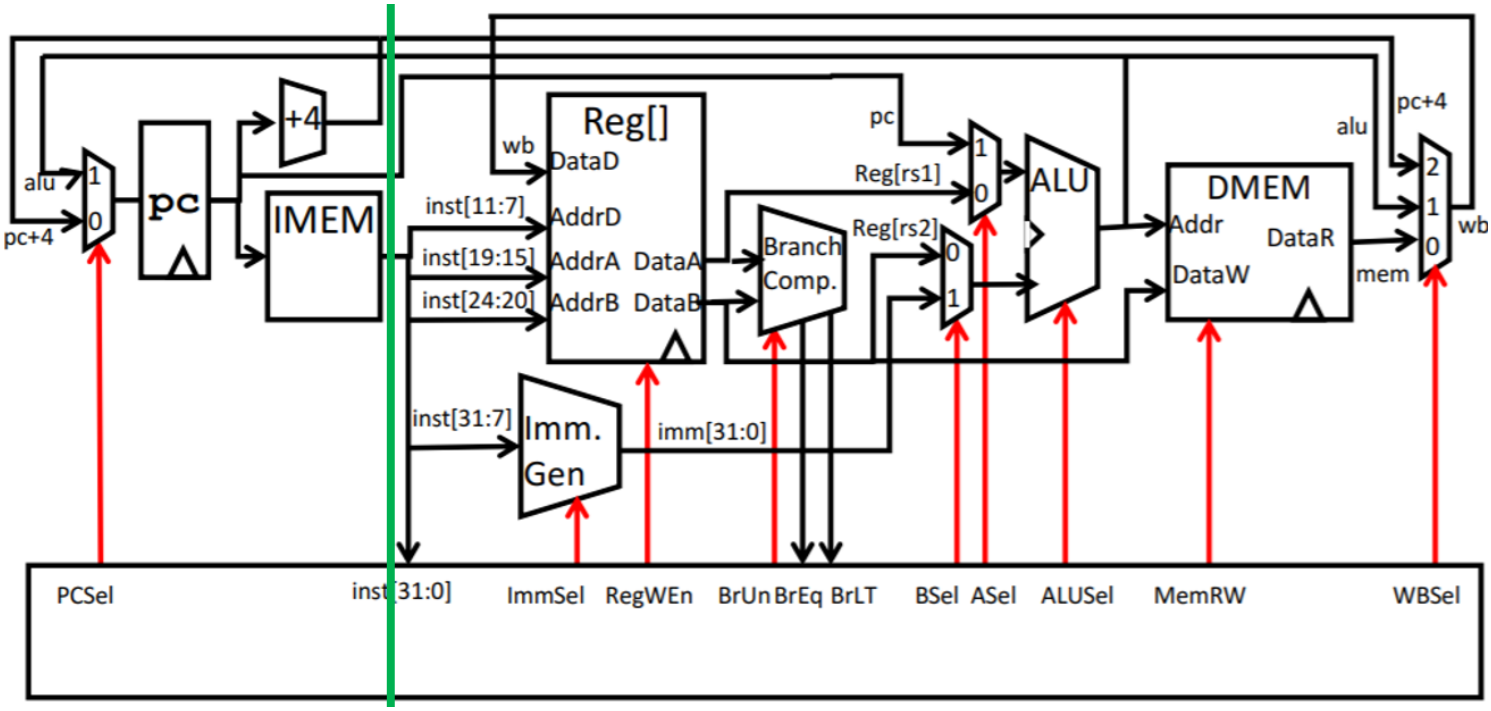
1. Navigate to `tests/integration-custom/in`.
2. Write a RISC-V test and save it in a filename ending in `.s`.
3. Run `bash test.sh test_custom`.

Task 10: Pipelining

In this task, you will implement a 2-stage pipeline in your CPU:

1. **Instruction Fetch**: An instruction is fetched from the instruction memory.
2. **Execute**: The instruction is decoded, executed, and committed (written back). This is a combination of the remaining four stages of a classic five-stage RISC-V pipeline (ID, EX, MEM and WB).

The separation between the two pipeline stages (highlighted by the green dividing line on the datapath) is illustrated below.



Task 10.1: Getting Started

To get started, first think about which paths will have intermediate pipeline registers in them. Look at the provided illustration above and consider all the paths that intersect the dividing line. Paths that transfer data to the rest of the datapath (data going from left to right) will have corresponding pipeline registers in them, while feedback paths (data going from right to left) will not.

Think about which values are now different between the two stages of the pipeline. For example, will stage 1 and stage 2 have the same or different PC values? If the stages need

different PCs, then you now need two different PC values in your circuit at any given time step.

Once you've listed out which values are different between the stages (hint: there aren't many), you'll need to store those values between the pipelining stages.

Finally, go through your entire circuit and make sure that you specify which stage's value you want to use for any values that are different between stages. For example, if the stages need different PCs, then any time you use PC in your circuit, you should specify whether you want to use the stage 1 PC, or the stage 2 PC.

Note: During the first cycle, the instruction register sitting between the pipeline stages won't contain an instruction loaded from memory. What should the second stage do? Luckily, Logisim automatically sets registers to zero on reset, so the instruction pipeline register will automatically start with a no-op! If you wish, you can depend on this behavior of Logisim.

Task 10.2: Hazards

Since your CPU will support branch and jump instructions, you'll need to handle control hazards that occur when branching.

The instruction immediately after a branch or jump should not be executed if a branch is taken. By the time you send a branch/jump instruction into stage 2, stage 1 has already fetched (possibly) the wrong next instruction. Therefore, you will need to *flush* the instruction fetched in stage 1 by replacing it with a no-op. You should flush the stage 1 instruction only if a branch is taken in the stage 2 instruction (do not flush if it is not taken). You should always flush the stage 1 instruction when the stage 2 instruction is a jump.

Hint: One of the control logic signals will tell you whether a branch or a jump is taken. You can use this control logic signal (from stage 2) in your stage 1 logic to determine when you need to flush the pipeline.

To flush an instruction, your stage 1 logic should send a no-op instruction into stage 2 instead of using the fetched instruction. You can use `addi x0, x0, 0` (`0x00000013`) as a no-op.

Some more things to consider:

- To MUX a no-op into stage 2, do you place it *before* or *after* the instruction register?
- What address should be requested next while the EX stage executes a no-op? Is this

different than normal?

Testing and Debugging

Add the **`--pipelined`** or **`-p`** flag to the testing commands to run the tests from the previous tasks on your pipelined CPU. For example:

```
bash test.sh run_custom -p
bash test.sh test_integration_branch -p
bash test.sh test_integration_immediates -p
```

Note that your pipelined CPU will **no longer** pass the non-pipelined tests (i.e. if you run tests without `-p`, they'll fail).

Task 11: Partner/Feedback Form

Congratulations on finishing the project! We'd love to hear your feedback on what can be improved for future semesters.

Please fill out this short form , where you can offer your thoughts on the project and (if applicable) your partnership. Any feedback you provide won't affect your grade, so feel free to be honest and constructive.

Submission and Grading

Submit your assignment to the Project 3B submission on Gradescope. Part B is worth 80% of your overall Project 3 grade.

- Instructions (1.5 points each, 54 points total)
- Integration tests (2 points each, 24 points total)
- Feedback form (2 points)

Total: 80 points

Control Logic

The control logic subcircuit takes the instruction bits and outputs all the control signals needed to execute that instruction. Here is a summary of the control signals you should implement:

Signal	Bit Width	Purpose
`PCSel`	1	Selects the ALU input for all B-type instructions where the branch is taken (according to the branch comparator output) and all jumps. Selects the PC+4 input for all other instructions.
`ImmSel`	3	Selects the instruction format so the immediate generator can extract the immediate correctly. The default encoding is `[0b000 = I]`, `[0b001 = S]`, `[0b010 = B]`, `[0b011 = U]`, `[0b100 = J]`, though you're welcome to pick your own (see below).
`RegWEn`	1	1 if the instruction writes to a register, and 0 otherwise.
`BrUn`	1	1 if the branch instruction is unsigned, and 0 if the branch instruction is signed. Don't care for all other instructions.
`ASel`	1	Selects whether to send the data in `RegReadData1` or the PC to the ALU.
`BSel`	1	Selects whether to send the data in `RegReadData2` or the immediate to the ALU.
`ALUSel`	4	Selects the correct operation for the ALU. See Task 1 for the mapping of `ALUSel` values to operations.
`MemRW`	1	1 if the instruction writes to memory, and 0 otherwise.
`WBSel`	2	Selects whether to write the memory read from DMEM, the ALU output, or PC+4 to `rd`.

There are two main approaches to implementing this logic:

1. **Read-only memory (ROM)**: Type out the control signals in a table. Use logic gates to

determine which row of signals from the table should be outputted for a given instruction. This involves less Logisim wiring and more filling in values in a spreadsheet.

2. **Hard-wired control**: Use logic gates (e.g. AND/OR/NOT gates, MUXes) to calculate the control bits from the instruction. This involves more Logisim wiring.

Regardless of which approach you choose, you should modify `control_logic.circ` in each task to implement your control logic.

ROM Approach

Setup

1. **Make a copy of this spreadsheet**: [61C SP25] Project 3B ROM Control Logic .
2. For each instruction, you'll **fill in the cells** under the "Control Signals" header. Do not modify any cells under other headers.
3. **Copy the data** in the "ROM Output" column (not including the headers).
4. **Open `control_logic.circ`** in Logisim, and **click on the ROM**.
5. In the **properties tab** on the left-hand sidebar, **click on "(click to edit)"** next to the "Contents" label.
6. **Click on the upper-left-most data cell** (which will be a collection of 4 hex digits without a preceding `0x`) and **paste your values from earlier into the ROM**. You should see the ROM control bits change to the values from your spreadsheet.
7. **Click "Close Window"** to exit the ROM programming view.

Filling in Spreadsheet

Some things to keep in mind when filling out the spreadsheet:

- Enter binary digits without the leading `0b` (type `01` instead of `0b01`).
- If the upper-right corner of the control signal's cell turns red, or if you see a warning when you hover over the cell, your cell contents are invalid. Common sources of errors include entering a control signal that has too many or too few bits, or entering characters other than `0` or `1` in a control logic cell.
- If a control signal is "don't care" for a certain instruction, you can put any valid value in the

corresponding cell.

- Make sure to fill in every cell in a given row. If a row has empty cells, the ROM output value is unreliable (garbage).
- You can use any encoding (which binary digits correspond to which control logic cases) for your control logic, as long as your encoding is consistent with your circuit. For example, suppose you choose to set `ASel = 0` for instructions that input the data in `RegReadData1` to the ALU, and `ASel = 1` for instructions that input the PC to the ALU. When you make a mux with `ASel` as the select bit, input 0 should be `RegReadData1` and input 1 should be the PC. You could also use the opposite encoding (`ASel = 0` for PC and `ASel = 1` for `RegReadData1`) and flip the mux inputs, and the circuit would still behave the same way.

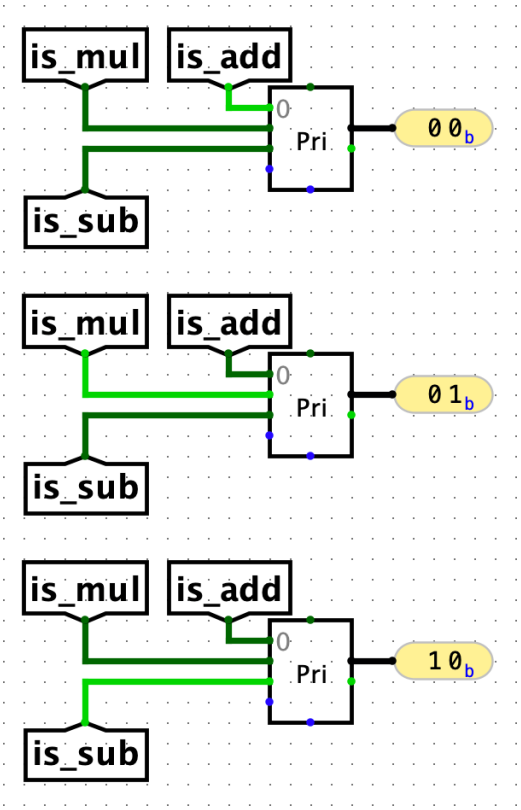
Wiring ROM Inputs with a Priority Encoder

Once you have the spreadsheet ready, you'll need to add wiring to select which row of the ROM table you want to use for a given instruction.

The "ROM Input" column on the spreadsheet indicates the input that must be passed into the ROM for each instruction. For example, your control logic, when passed in an `addi` instruction, needs to pass `15` (`0b1111`) into the ROM so the correct entry of control signals can be selected.

To determine what instruction is being inputted into the control logic subcircuit, it might help to use comparators and constants to compare bits of the instruction against some fixed constant value.

To map instructions to their specific input, a Priority Encoder might be helpful. To see how the Priority Encoder works, let's consider a toy example of an ALU that only performs 3 operations: add (ALUSel=`0`), mul (ALUSel=`1`), and `sub` (ALUSel=`2`). Suppose that we have `is_add`, `is_mul`, and `is_sub` tunnels that tell us whether an instruction requires the ALU to perform that operation. To generate the ALUSel control signal, we can simply connect the `is_add`, `is_mul`, and `is_sub` tunnels to their corresponding ALUSel position (say position 2 for sub) on the Priority Encoder. Now, the Priority Encoder will return the position of the active input signal, which in this case is the correct ALUSel value.



Note: In case there are multiple active signals at the same time, a Priority Encoder returns the largest active position. However, you should try to avoid having multiple active signals at once for this use case. After all, a single instruction can't require the ALU to perform both add and multiply at once.

Wiring PCSel

The `PCSel` control signal cannot be encoded in the ROM since it depends on `BrEq` and `BrLt`, which are not passed into the ROM unit. To complete the control logic, write hard-wired control to drive the PCSel output.

Testing and Debugging

Unit Tests

The provided unit tests check functionality of your subcircuits (e.g. immediate generator, branch comparator). They do not check your entire CPU implementation. You don't need to write any unit tests yourself.

As an example, let's debug the `alu-add` unit test. First, **run the ALU unit tests with `bash test.sh test_alu`**. If the test doesn't pass, this will print out the difference between your subcircuit output and the reference output.

Viewing Output Files

The next step to debug unit tests is to compare the expected reference output to your subcircuit output. These output files will always be in the `out` directory. You can use `bash test.sh format filename` (replacing `filename`) to view the output files.

View the reference output with `bash test.sh format tests/unit-alu/out/alu-add.ref`. You should see this reference output:

```
Time ALUSel A        B        ALUResult
00   0      00002020 00000f0f 00002f2f
01   0      ffffdead ffffbeef ffff9d9c
02   0      00007fff 00000001 00008000
03   0      00000000 00000000 00000000
```

This shows the inputs (`A`, `B`, and `ALUSel`) sent to your subcircuit at each time step, and the expected output (`ALUResult`).

Next, **run `bash test.sh format tests/unit-alu/out/alu-add.out` to see the output from your subcircuit**. Here's the output when the test is run on unmodified starter code:

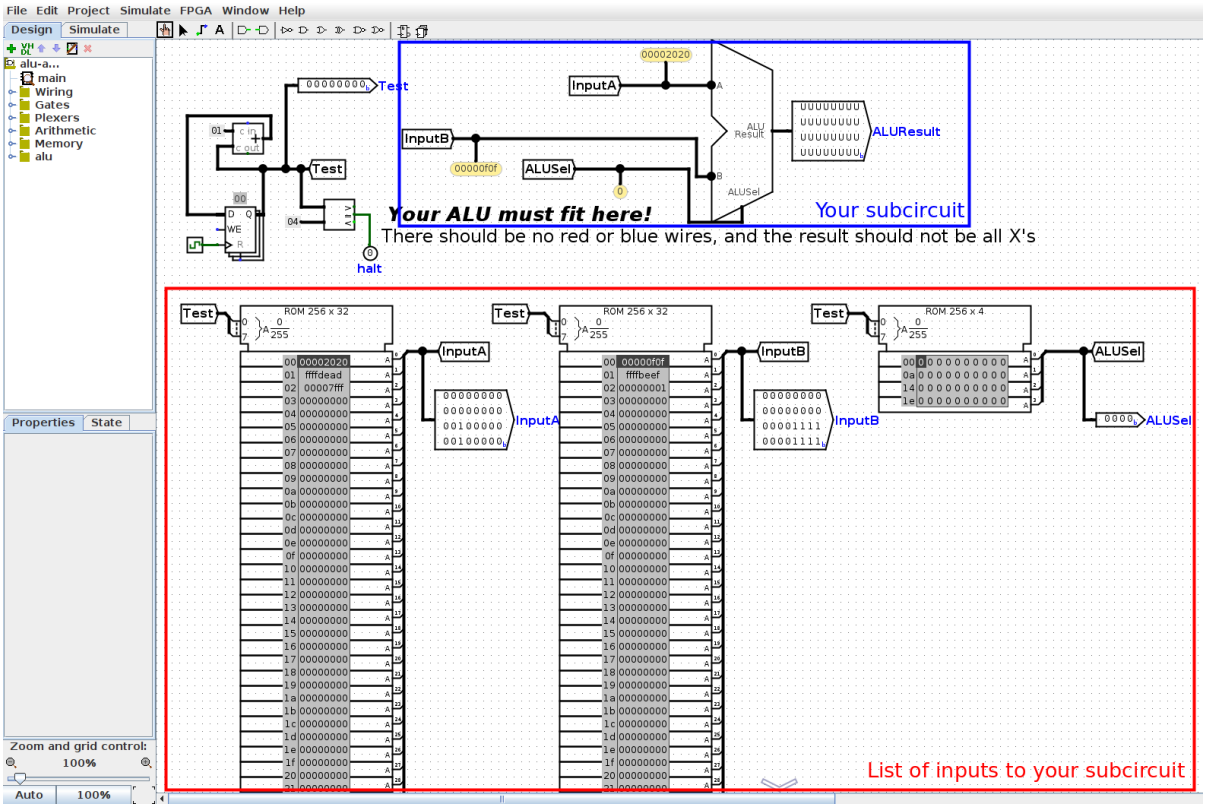
```
Time ALUSel A        B        ALUResult
00   0      00002020 00000f0f UUUUUUUU
01   0      ffffdead ffffbeef UUUUUUUU
02   0      00007fff 00000001 UUUUUUUU
```

```
03   0      00000000 00000000 UUUUUUUU
```

Note that in the example, the inputs to your subcircuit are the same, but the output (`ALUResult`) of your subcircuit is different (undefined).

Using Debugging Circuits

Each unit test has a `.circ` test circuit you can use for debugging. As an example, **open `tests/unit-alu/alu-add.circ`**, which corresponds to the failed test from the previous section. The first thing you'll see in this circuit is the testing harness:

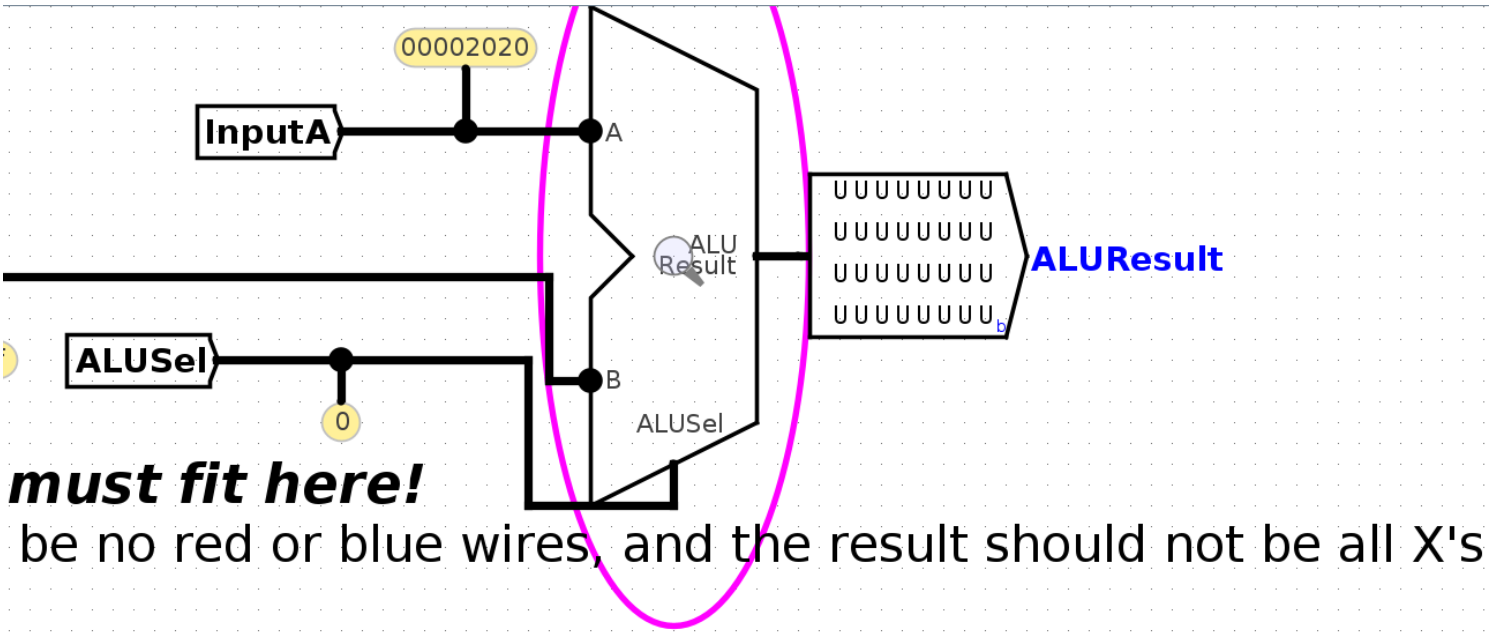


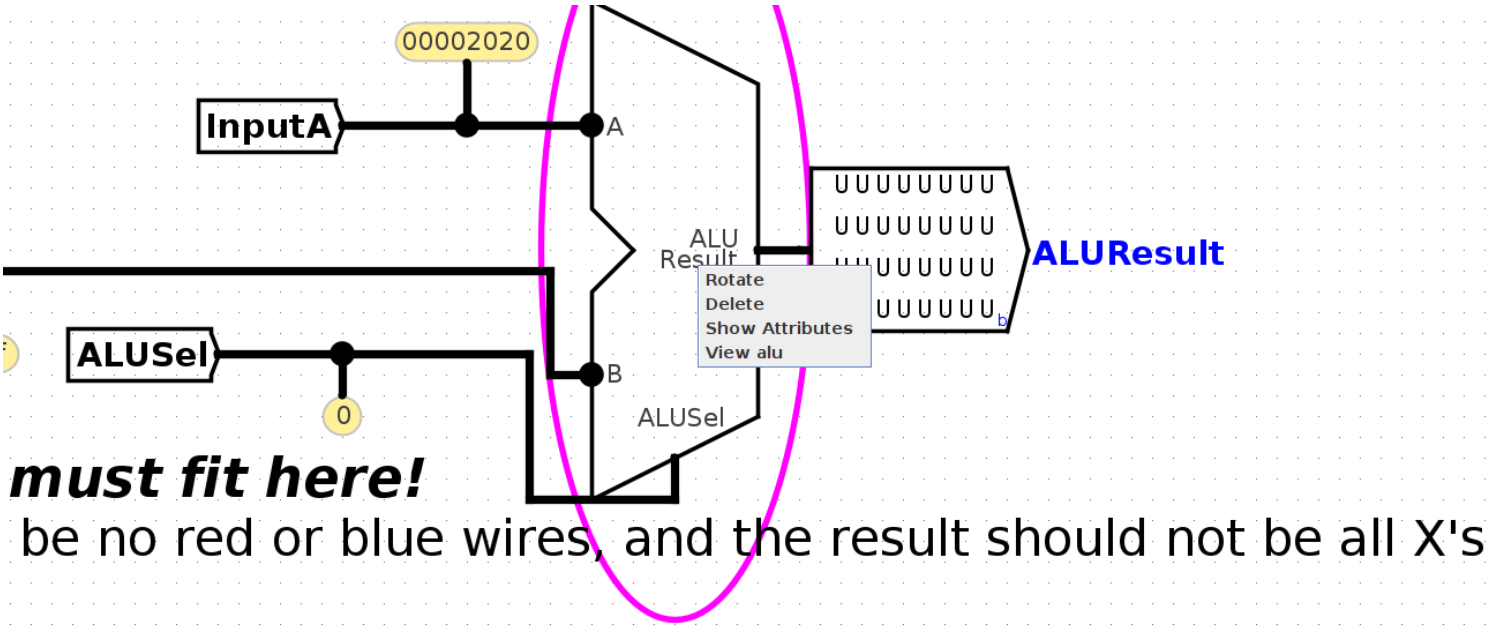
This feeds a sequence of inputs (`InputA`, `InputB`, and `ALUSel`) to your ALU.

The ROM (in the red box) contains a list of inputs to your circuit. The first input (`InputA = 0x00002020`, `InputB = 0x00000f0f`, `ALUSel = 0b0000`) is highlighted in dark gray. You can also see these values being passed into your ALU (in the blue box) with the probes.

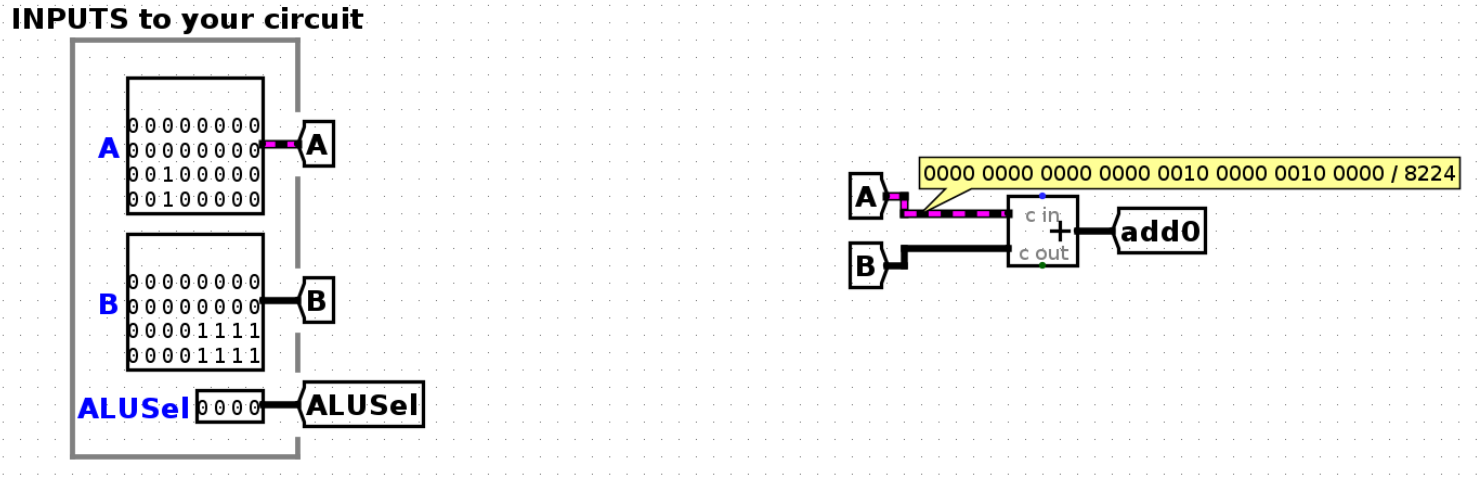
In this picture, the `ALUResult` output from your ALU is undefined (all `U`s). To see why, we can view our ALU subcircuit to see what logic it's doing. To click into your ALU, you can either **right-click the ALU** and **select "View alu"**, or **click the ALU** and **click the magnifying glass**, as

shown below:





Inside your ALU subcircuit, you can see inputs (`A`, `B`, and `ALUSel`) provided from the harness to your subcircuit. You can **click on wires to see the values in those wires.**



If the output of your subcircuit isn't what you expect, you can probe wires to investigate where the incorrect output is coming from. In the starter circuit, the `ALUResult` output is undefined. In this case, note that the `ALUResult` tunnel is undefined, so we probably want to send a value to this tunnel.

To return to the harness, you can click on `main` in the Simulate → Active Simulations tab in the top-left corner.

To view later inputs, click Simulate → Manual Tick Full Cycle, which will highlight the next row of the ROM blocks and send this next input into your subcircuit. You can tick cycles while viewing your ALU subcircuit to see later inputs.

To reset the simulation, click Simulate → Reset Simulator. You can also close and re-open the debugging circuit.

Integration Tests

Integration tests execute RISC-V instructions on your entire CPU and compare the outputs to the result of running those instructions on Venus. You'll need to make your own integration tests by writing out RISC-V instructions (the testing framework will then create the testing and debugging circuit for you).

In addition to the example below, please check out this video regarding writing integration tests.

As an example, let's debug the `addi-basic` unit test. First, **run the addi integration tests with `bash test.sh test_addi`**. If the test doesn't pass, this will print out the difference between your

CPU output and the reference output.

Viewing Testing Code

Before we can figure out why this test failed, we should first figure out what code this test tried to run on your CPU.

The RISC-V instructions being executed by your CPU will be inside the `in` directory. In this example, **view `tests/integration-addi/in/addi-basic.s`** to see what RISC-V instructions are being run in this test. You should see these instructions:

```
addi t0, x0, 1
addi t0, x0, 42
addi t0, x0, 256
addi t0, x0, 2047
```

Reading Test Output

If an integration test fails, you should see some terminal output like this:

```
$ bash test.sh test_addi
FAIL: tests/integration-addi/addi-basic.circ (Did not match expected output)
             Time PC       Instruc. ra (x1)  sp (x2)  t0 (x5)  t1 (x6)  t2 (x7)  s
  Reference: 0001 00000004 02a00293 00000000 00000000 00000001 00000000 00000000 0
  Student:   0001 00000004 02a00293 00000000 00000000 00000000 00000000 00000000 0
  ---
  Reference: 0002 00000008 10000293 00000000 00000000 0000002a 00000000 00000000 0
  Student:   0002 00000008 10000293 00000000 00000000 00000000 00000000 00000000 0
  ---
  Reference: 0003 0000000c 7ff00293 00000000 00000000 00000100 00000000 00000000 0
  Student:   0003 0000000c 7ff00293 00000000 00000000 00000000 00000000 00000000 0
  ---
```

Each row of output shows the program counter (PC), instruction, and values in the 8 debug registers at each time step of running the input program.

You might see a lot of rows of output, but let's **focus on the first two rows**, which shows the first time step when your CPU doesn't match the reference output. In this case, that's time step

1.

In the first set of Reference/Student output, **look for registers with mismatched values**. In the above output, it looks like t0 should hold the value 1, but in your CPU, t0 holds the value 0.

Using the terminal output and the RISC-V code, try to work out what failed on the time step immediately before the first failing time step. In this output, since incorrect output showed up at time 1, something must have gone wrong at time 0. At time 0, the RISC-V code executed `addi t0, x0, 1` - now we can see why the reference output has a 1 in t0. However, our implementation did not put a 1 in t0, so it looks like the very first `addi` instruction didn't execute correctly.

Reading Jump/Branch Test Output

In tests like `addi-basic`, the RISC-V instructions execute in sequence, so it's simple to work out which instruction failed. For example, if the first line of terminal output shows time step `0003`, you know that the instruction at time step 2 failed, and this instruction must be `addi t0, x0, 256` (the second instruction in the testing code, zero-indexed).

However, when we're running code with jumps and branches, the RISC-V instructions don't execute in sequence, so we have to be more careful to figure out which instruction failed. Consider the output below:

```
$ bash test.sh test_integration_branch
FAIL: tests/integration-branch/branch-basic.circ (Did not match expected output)
             Time PC       Instruc. ra (x1)  sp (x2)  t0 (x5)  t1 (x6)  t2 (x7)  s
  Reference: 0005 00000018 fe948ce3 00000000 00000000 00000000 00000000 00000000 
  Student:   0005 00000024 fe944ce3 00000000 00000000 00000000 00000000 00000000 
  ---
  Reference: 0006 00000010 fe04cce3 00000000 00000000 00000000 00000000 00000000 
  Student:   0006 00000028 fe000ce3 00000000 00000000 00000000 00000000 00000000 
  ---
  Reference: 0007 00000014 fe000ce3 00000000 00000000 00000000 00000000 00000000 
  Student:   0007 00000020 fe904ce3 00000000 00000000 00000000 00000000 00000000 
  ---
```

We know that the instruction at time step 4 failed, but which instruction is that? Let's check the RISC-V code in `tests/integration-branch/in/branch-basic.s`:

```
         addi s1, x0, 1
         addi s0, x0, -1
label5:  beq  x0, x0, label1
label6:  bltu x0, s0, label8
label4:  blt  s1, x0, label5
         beq  x0, x0, label6
label3:  beq  s1, s1, label4
label10: beq  s0, s0, end
label2:  blt  x0, s1, label3
label9:  blt  s0, s1, label10
label1:  beq  x0, x0, label2
label8:  bltu s1, s1, label9
         beq  s1, s1, label9
end:     addi s0, x0, 2
```

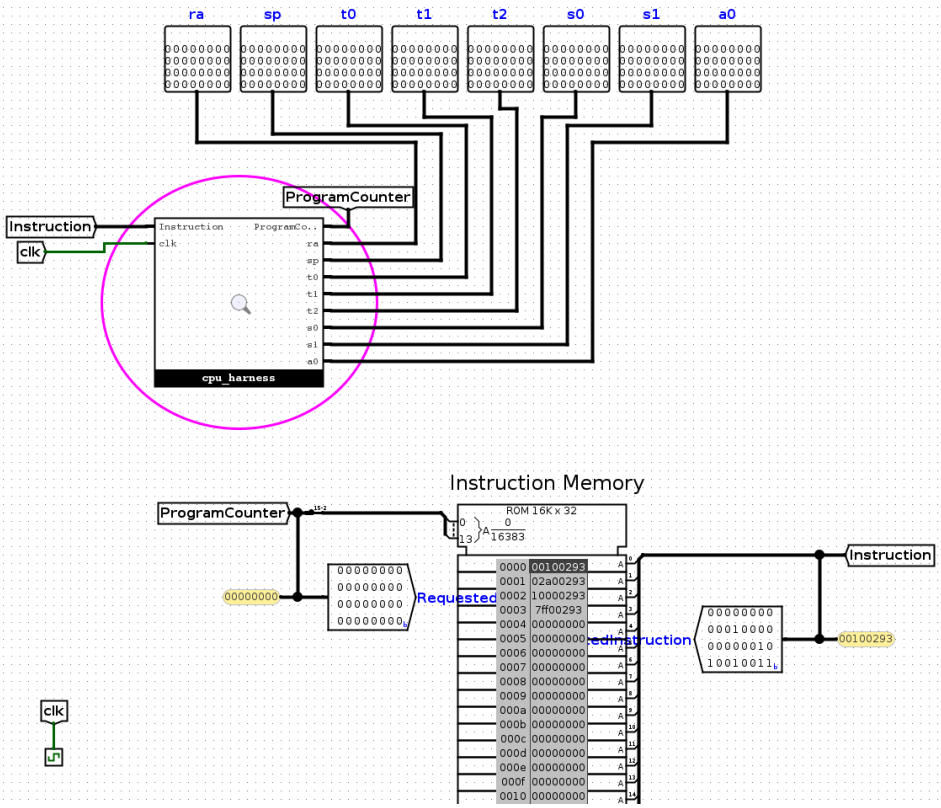
- At time step 0, `addi s1 x0 1` executes.
- At time step 1, `addi s0 x0 -1` executes.
- At time step 2, `beq x0 x0 label1` executes, causing a branch to be taken.
- At time step 3, `beq x0 x0 label2` (the line at `label1`) executes, causing a branch to be taken.
- At time step 4, `blt x0 s1 label3` (the line at `label2`) executes.

At time step 4, s1 holds 1, so the branch at time step 4 should be taken. This means at time step 5, we should be executing the instruction at `label3`. This is the 6th instruction (zero-indexed), which corresponds to PC value 24 (each instruction is 4 bytes), which is 0x18 (the value under PC in the reference output).

However, in our output, the PC is at 0x24, or 36 in decimal. This is the 9th instruction, which is `label9` (the line directly after `label2`). It looks like at time step 4, the branch instruction should have updated the PC by taking a branch, but in our implementation, the PC was updated incorrectly.

Opening Debugging Circuits

Each integration test has a `.circ` test circuit you can use for debugging. To debug the example test from the previous section, **open tests/integration-addi/addi-basic.circ**.



The top-level harness for each integration test contains a ROM block (bottom half of screenshot) containing the RISC-V instructions for that test, representing IMEM (instruction memory). These instructions are passed into your CPU (the circled `cpu_harness` block at the top). You can also see the 8 debug register outputs; the testing framework will log their values into the `.out` file when running the test.

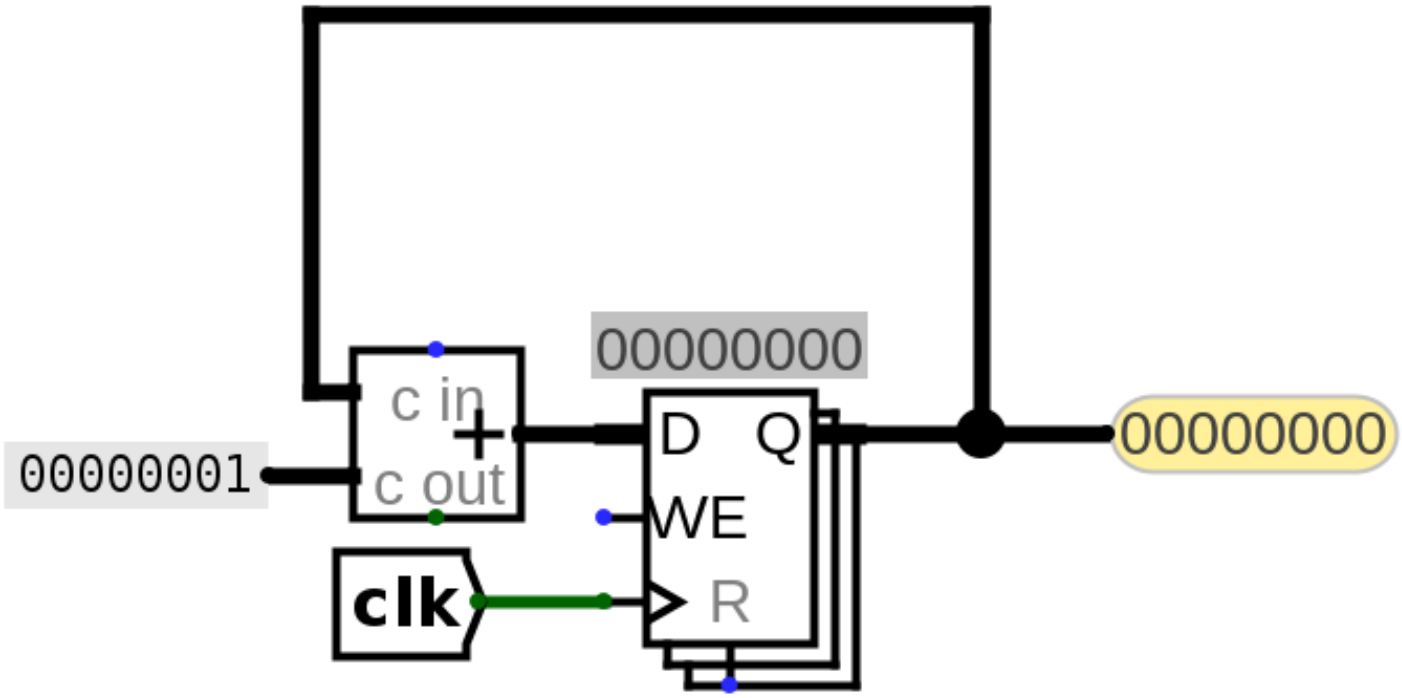
To view your CPU circuit, either **right-click the `cpu_harness` block** and **select "View cpu_harness"**, or **click the `cpu_harness` block** and **click the magnifying glass**. This takes you into the CPU harness, where your CPU interacts with memory. **Click another time into the `cpu` block**, and now you should see the CPU you've been wiring.

To step through the RISC-V instructions, click Simulate → Manual Tick Full Cycle. In each clock cycle, your CPU will output a new `ProgramCounter` to the harness, which will use the new PC to select the next instruction for your CPU to execute. In this `addi` test, the instructions execute in sequence, but when testing branches and jumps later, the CPU could output a different `ProgramCounter` value (not always adding 4) and execute the instructions in a different order.

To reset the simulation, click Simulate → Reset Simulator. You can also close and re-open the debugging circuit.

Time Step Counter

When instructions don't execute in sequence, it can be hard to figure out how many cycles to tick before you reach the first failing instruction. To make that easier, we recommend adding a small counter circuit to your `cpu.circ` circuit:



This is the same logic as the PC circuit in the starter code, except the counter circuit adds 1 at each time step instead of adding 4. Also, the `ProgramCounter` tunnel has been replaced with a Probe that lets you view the value of the wire while debugging.

This counter circuit is only here for debugging, not for your CPU implementation. When you modify your PC circuit to handle jumps and branches, this counter circuit will still be adding 1 at every time step, allowing you to keep track of when the first failing instruction occurs.

Now, you can use this counter to stop at the first failing instruction. For example, if your terminal output looks like this:

```
$ bash test.sh test_integration_branch
FAIL: tests/integration-branch/branch-basic.circ (Did not match expected output)
             Time PC       Instruc. ra (x1)  sp (x2)  t0 (x5)  t1 (x6)  t2 (x7)  s
  Reference: 0006 00000010 fe04cce3 00000000 00000000 00000000 00000000 00000000 f
  Student:   0006 00000028 fe000ce3 00000000 00000000 00000000 00000000 00000000 f
```

```
---
Reference: 0007 00000014 fe000ce3 00000000 00000000 00000000 00000000 00000000 
Student:   0007 00000020 fe904ce3 00000000 00000000 00000000 00000000 00000000 
---
```

You know that the instruction at time step 5 failed, so you should go into the debugging circuit and tick the clock until the counter circuit is showing 5. Now, you can start poking around to see what instruction is currently executing, and why it's failing.

Using Debugging Circuits

At this point, you should have identified which RISC-V instruction is failing on your circuit, the expected register values after that instruction, and your register values after that instruction. You should have also opened a debugging circuit and ticked the clock until the failing instruction. Now, it's time to poke at all the wires in your circuit to see why this instruction is failing.

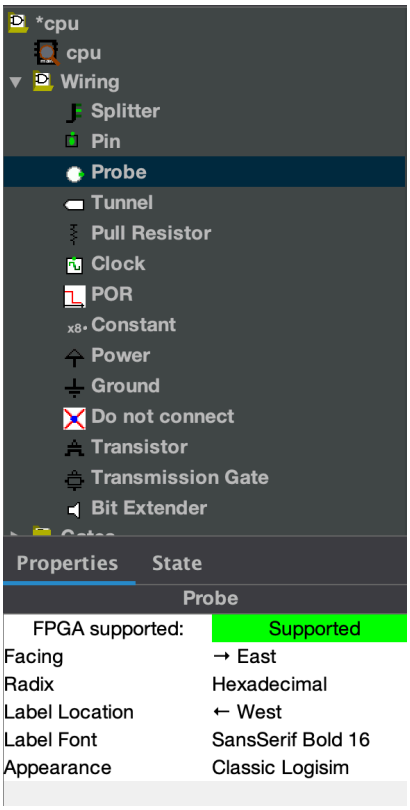
Some useful wires you can poke:

- If the instruction writes back to a register: which RegFile input corresponds to the data we're writing on this cycle?
- If the instruction uses the ALU for computation: what are the three inputs (A, B, and ALUSel) to the ALU, and are they what you expect? What is the output from the ALU, and is it what you expect?
- If the instruction is a branch/jump: what value is getting written to the PC register on the next clock cycle? Is that value what you expect? Which control logic signal determines the value written to the PC register? Does that signal have the right value?
- If the instruction is a store/load instruction: what are the three inputs to DMEM, and are they what you expect? What is the output from DMEM, and is it what you expect?

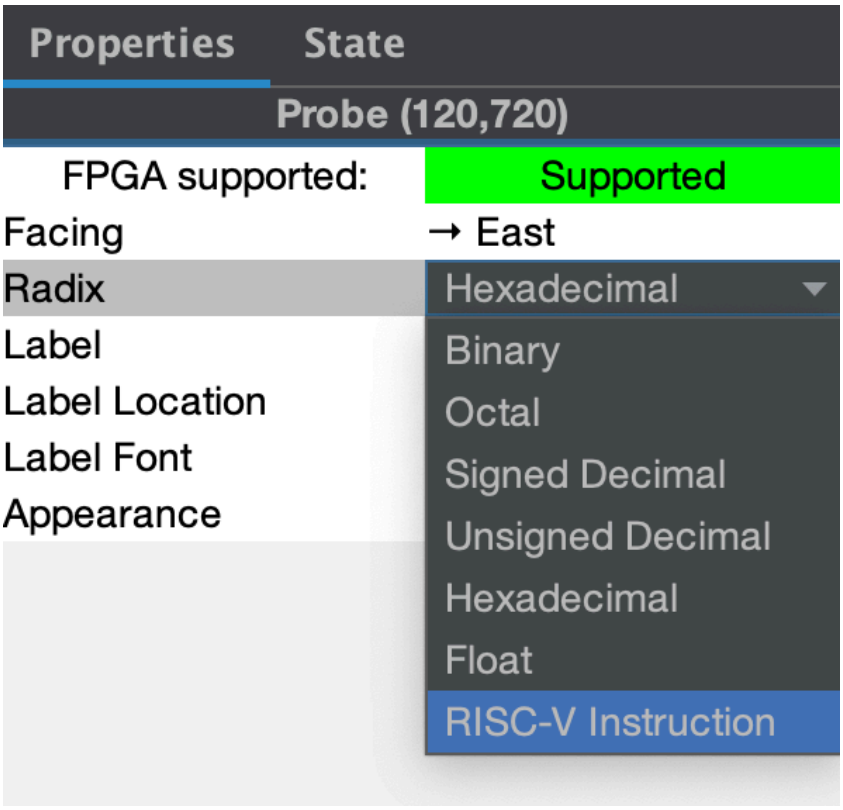
Note: If the failing instruction is a load instruction, it could be the case that your store instruction failed to write to DMEM first, and now the load instruction can't read a value that was never written to DMEM. In this case, you might want to stop at store instructions too and check that they're working as you expect.

Viewing Instructions using Probes

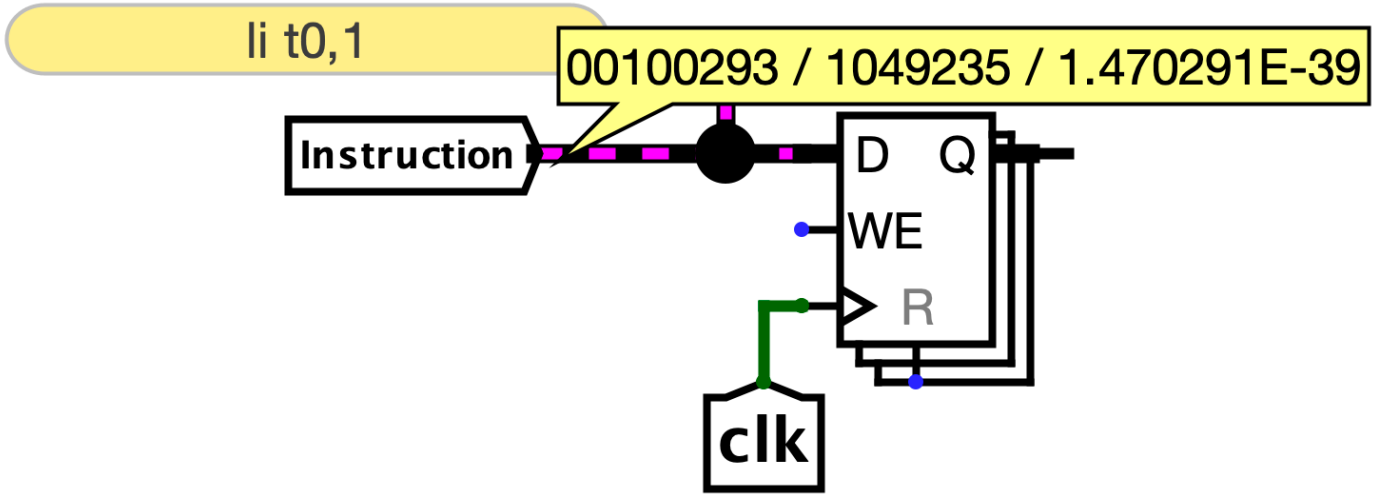
While it is possible to keep track of instructions via manual conversion or counting, it may be useful to see the instruction currently in the datapath. To do this, you can use the probe component in Logisim:



The probe will currently show what value is in the wire it is attached to, and by default, the value will be displayed in hexadecimal format. However, there is also an option to have the probe automatically convert 32-bit hexadecimal instructions into RISC-V format for ease of debugging. To do this, change the radix option for the probe to RISC-V Instruction, as shown below:



Once this is selected, the probe will display the current instruction in a more readable format, as shown below:



Writing Integration Tests

To write an integration test, all you need to do is write some RISC-V instructions for your CPU to

run, and the testing framework will handle the rest.

1. Navigate to `tests/integration-custom/in`.
2. Write a RISC-V test and save it in a filename ending in `.s`.
3. Run `bash test.sh test_custom`.

Some things to keep in mind when writing your RISC-V instructions:

The testing framework only checks the values in the 8 debug registers when comparing your CPU output with the reference output, so when writing your own tests, make sure to only use the 8 debug registers.

This also means the testing framework doesn't check memory (DMEM) when comparing your CPU with the reference. To check values in memory or a non-debug register, you'll need to put the value back into a debug register. For example, to test if a store works, you'll probably have to load the value back from memory into a debug register to see if the value was successfully stored.

IMEM and DMEM are separate in Logisim, but combined in Venus. This means that if you write assembly code that tries to access memory overlapping with instructions, Venus will throw an error. Since counting exactly how many instructions your assembly code requires, and multiplying that by 4 can be annoying, we suggest you load/store using addresses greater than 0x3E8 (leaving space for 1000 bytes/250 instructions), and increase this offset if you have more instructions.

Make sure to write RISC-V instructions that behave differently on a working CPU and a buggy CPU. For example, consider this test:

```
addi t0, x0, 0
addi t1, x0, 0
```

This wouldn't be very useful to check for a working CPU, because the output in the debugging registers could be all zeros even if your CPU doesn't work. As another example:

```
beq t0, t0, 4
addi t1, x0, 10
```

On a working CPU, this would branch to the `addi` instruction. On a buggy CPU where the branch

is incorrectly not taken, this would still execute the `addi` instruction, so this test doesn't do a very good job of distinguishing working circuits from buggy circuits.

Logisim Tips

This section contains some helpful Logisim tips and pitfalls to avoid.

Wiring

- If you want to know more details about each component, go to `Help -> Library Reference` for more information on the component and its inputs and outputs.
- Use tunnels! They will make your wiring cleaner and easier to follow, and will reduce your chances of encountering crossed wires or unexpected errors.
- Ensure you name your tunnels correctly. The labels are case sensitive!
- You can hover your cursor over an input/output on a component to get slightly more information about that input/output.

Wiring Pitfalls

- Your circuits should always fit in the provided harnesses. This means that you should not edit the provided input/output pins or add new ones. To ensure your circuit fits int he harness, you can open the harnesses in the `harnesses` folder and check that there are no errors.
- Don't create new `.circ` files. You can make additional subcircuits if you want, but they must be in existing files.

Subcircuits

- Note that if you modify a subcircuit, and another circuit file uses that subcircuit, you will need to close and re-open the outer circuit to load the changes from the subcircuit. For example, if you modify `imm-gen.circ`, you should close and re-open `cpu.circ` to load your changes.
- When modifying a subcircuit, you should always open up the subcircuit file. For example, you should modify `imm-gen.circ`, not the `imm-gen` subcircuit in `cpu.circ`.

Signal Tips

- The clock input signal (`clk`) can be sent into subcircuits or attached directly to the clock inputs of memory units in Logisim, but should not otherwise be gated (i.e., do not invert it, do not `AND` it with anything, etc.).
- We recommend not using the `Enable` input on your MUXes. In fact, you can turn that attribute off (`Include Enable?`). We also recommend that you disable the `Three-state?` attribute (if the plexer has it).

Banned Circuit Elements

The following circuit elements are not necessary for this project, so please don't use them in your implementation.

- Pull Resistor
- Transistor
- Transmission Gate
- Power
- POR
- Ground
- Divider
- Random
- PLA
- RAM
- Random Generator

Partial Loads and Stores

Remember that every byte in C memory is referred to by a 32-bit address. In the CPU, store and load instructions access C memory (DMEM) by passing a 32-bit address to DMEM to request bytes of memory starting at that address.

For example, if we wanted to load a byte from address `4` = `0b000100`, we can pass in the address `4` (sign-extended to 32 bits) to DMEM. But what if we wanted to load a half-word (16 bits) or a byte (8 bits) from memory? How does DMEM loading and storing a different number of bytes for different types of instructions?

DMEM in Logisim

The DMEM unit in this project is always configured to access 4 bytes at a time, starting at an address that is a multiple of 4. This means that DMEM can access the bytes at memory addresses `0–1–2–3` or the bytes at memory addresses `12–13–14–15`. (Intuitively, DMEM can access a set of 4 bytes in one of the red boxes in the diagram.)

Decimal address	Binary address
0	0b000000
1	0b000001
2	0b000010
3	0b000011
4	0b000100
5	0b000101
6	0b000110
7	0b000111
8	0b001000
9	0b001001
10	0b001010
11	0b001011
12	0b001100
13	0b001101
14	0b001110
15	0b001111
16	0b010000
17	0b010001
18	0b010010
19	0b010011
...	...

However, in a single access, DMEM cannot access the bytes at memory addresses `13–14–15–16`, because it does not start at a multiple of 4. (Intuitively, the `13–14–15–16` access crosses one of the red lines in the diagram, so it is not supported by DMEM.)

DMEM supports this behavior by always zeroing out the bottom 2 bits of any address you provide, and then accessing 4 bytes starting at this modified address. For example, if you give DMEM the address `19` = `0b010011`, DMEM will zero out the bottom 2 bits to get `16` = `0b010000`, and then accessing 4 bytes starting at this modified address (`16–17–18–19`).

In other words, the provided address will be rounded down to the nearest multiple of 4, and 4 bytes will be read starting at this modified address. (Note that zeroing out the bottom 2 bits of a number is mathematically equivalent to rounding down the number to the nearest multiple of 4.)

You don't need to zero out the bottom 2 bits yourself--the provided DMEM implementation will automatically do this for any address you provide.

Note 1: You don't need to think about endianness for this section. Endianness is relevant when you're trying to interpret a set of 4 bytes as a word (e.g. integer, address) to do some calculation on it, but at this level of abstraction, we only need to worry about the bits being loaded and stored, not what they mean.

Note 2: Due to Logisim size limitations, the memory unit only uses the lower 16 bits of the provided address, discarding the upper 16 bits. This means that the memory can only store 2^16 bytes of data. The provided tests will always set the upper 16 bits of addresses to 0, and any tests you write should avoid using the upper 16 bits when interacting with memory.

Alignment

In this project, all memory accesses will be *aligned*. This means that a single load or store instruction will never cross a word boundary in memory. (Intuitively, a single memory access will never cross one of the red lines in the diagram.)

For completeness, this means that: All `lw` and `sw` instructions will use memory addresses that end in `0b00` (i.e. are a multiple of 4). All `lh` and `sh` instructions will use memory addresses that end in `0b00`, `0b01`, or `0b10` (never an address that ends in `0b11`).

You should not implement any unaligned memory accesses in this project.

Conceptual Overview: Loads

The `partial_load.circ` circuit is designed to take data read from DMEM and process it in order to put the relevant data into a register.

Remember that in RISC-V, load instructions read bytes from memory (DMEM) and put those bytes into a 32-bit register. There are three load instructions that each read a different number of bits from memory: `lw` reads 32 bits, `lh` reads 16 bits, and `lb` reads 8 bits. In all three instructions, 32 bits need to be put into the destination register. If fewer than 32 bits are read from memory, they are sign-extended to 32 bits before being put in the destination register.

Regardless of instruction, DMEM will always return 32 bits of memory, but in the `lh` and `lb` instructions, we only want 16 or 8 bits from memory, so we need to extract the correct bits out of the 32 bits returned from DMEM.

How do we know which of the 32 bits from DMEM we care about? We can check whether the instruction is `lw`, `lh`, or `lb` to know how many bits we want to extract. We can then check the bottom 2 bits of the memory address to know where in the 32 bits we want to start extracting bits.

For example, suppose we had a `lb` instruction on address 6 = `0b000110`. We know that DMEM is going to return the 4 bytes at addresses `4–5–6–7`. Intuitively, we want just the byte at address `6`. We can determine this in the subcircuit with this logic: This is a `lb` instruction, so we only want one of these bytes. The bottom 2 bits of the address are `0b10`, so we want the 2nd byte (zero-indexed), which corresponds to bits 16-23.

As another example, suppose we had a `lh` instruction on address 9 = `0b001001`. We know that DMEM is going to return 4 bytes at addresses `8–9–10–11`. Intuitively, we want the bytes at addresses `9–10`. We can determine this in the subcircuit with this logic: This is a `lh` instruction, so we want two of these bytes. The bottom 2 bits of the address are `0b01`, so we want to start extracting at the 1st byte (zero-indexed). In summary, we want to extract the 1st and 2nd bytes (zero-indexed), which corresponds to bits 8-23.

Partial Load

Fill in the `partial_load.circ` subcircuit.

Signal Name	Direction	Bit Width	Description
`Instruction`	Input	32	The load instruction being executed.
`MemAddress`	Input	32	The memory address to read from (bottom two bits are not zeroed).
`DataFromMem`	Input	32	The data read from DMEM.
`DataToReg`	Output	32	The data to put in the register.

A table of all scenarios you need to handle in the partial load subcircuit is provided below:

Instruction	Type	Opcode	Funct3	Bottom 2 bits of `MemAddress`	Value to put in `DataToReg`

lb rd, offset(rs1)	I	0x03	0x0	0b00	`SignExt(DataFromMem[7:0])`
				0b01	`SignExt(DataFromMem[15:8])`
				0b10	`SignExt(DataFromMem[23:16])`
				0b11	`SignExt(DataFromMem[31:24])`
lh rd, offset(rs1)			0x1	0b00	`SignExt(DataFromMem[15:0])`
				0b01	`SignExt(DataFromMem[23:8])`
				0b10	`SignExt(DataFromMem[31:16])`
lw rd, offset(rs1)			0x2	0b00	`DataFromMem`

Conceptual Overview: Stores

The `partial_store.circ` circuit is designed to take data from a register and process it in order to store the relevant data into memory.

Remember that in RISC-V, store instructions read bytes from a register and put those bytes into memory (DMEM). There are three store instructions that each store a different number of bits from a register to memory: `sw` stores 32 bits, `sh` stores 16 bits, and `sb` stores 8 bits. Unlike loads, these are the exact number of bits that will be stored to memory (no sign-extension needed).

The `partial_store.circ` subcircuit has three jobs:

First, we need to extract the relevant bits from the 32-bit register data. If fewer than 32 bits are stored, they will always be the lowest bits from the register. In `sh` instructions, the bottom 16 bits are stored to memory, and in `sb` instructions, the bottom 8 bits are stored to memory.

Second, we need to put those bits in the right position of the 32-bit data we're passing to DMEM, so that the store to DMEM will store the bits at the right place in memory. We can use the bottom 2 bits of the memory address to know where to place the bits.

For example, suppose we had a `sb` instruction on address `3` = `0b000011`. DMEM will take 32 bits of data we provide and write them to addresses `0`–`1`–`2`–`3`, but we actually want to write 8 bits to address `3`. To do this, we can make a 32-bit value where the bytes at addresses `0`–`1`–`2` (bits 0-23) are all zeros (doesn't matter), and the byte at address `3` (bits 24-31) is the 8 bits we want to store in memory.

We can use this logic to implement this in our circuit: The bottom 2 bits of the address are `0b11` = 3, so we need to place the 8 bits we want to store at the 3rd byte (zero-indexed) of the data we're passing to DMEM.

Finally, we need to create a 4-bit *write mask* which will tell DMEM which bytes of the data we want to store to memory. Each bit of the write mask corresponds to one of the 4 bytes in memory that DMEM might store to. If a bit in the mask is 0, DMEM will not store that byte to memory.

For example, the write mask `0b0001` says to only write the 0th byte (bits 0-8) to memory, leaving the others unchanged. The write mask `0b1100` says to only write the 2nd and 3rd bytes (bits 16-31) to memory, leaving the others unchanged.

Partial Store

Fill in the `partial_store.circ` subcircuit.

Signal Name	Direction	Bit Width	Description
`Instruction`	Input	32	The store instruction being executed.

MemAddress	Input	32	The memory address to store to (bottom two bits are not zeroed).
DataFromReg	Input	32	The data from the register.
MemWEn	Input	1	The control signal indicating whether writing to memory is enabled for this instruction.
DataToMem	Output	32	The data to store to memory.
MemWriteMask	Output	4	The write mask indicating whether each byte of DataToMem will be written to memory.

A table of all scenarios you need to handle in the partial store subcircuit is provided below. All 3 store instructions have opcode 0x23.

Instruction	Funct3	Bottom 2 bits of MemAddress	DataToMem				MemWriteMask
			Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0	
sb rs2, offset(rs1)	0x0	0b00	0	0	0	DataFromReg[7:0]	0b0001
		0b01	0	0	DataFromReg[7:0]	0	0b0010
		0b10	0	DataFromReg[7:0]	0	0	0b0100
		0b11	DataFromReg[7:0]	0	0	0	0b1000
sh rs2, offset(rs1)	0x1	0b00	0		DataFromReg[15:0]		0b0011
		0b10	DataFromReg[15:0]		0		0b1100
sw rs2, offset(rs1)	0x2	0b00	DataFromReg				0b1111

Note that for any non-store instruction (i.e. when your MemWEn control signal is 0), MemWriteMask should be set to 0b0000.

Note: The bytes in DataToMem that aren't being written to the file (i.e. where MemWriteMask is 0) can technically be any value, but we've listed them as 0s in the table. The unit test for this part also assumes that those bytes will be 0.

Testing and Debugging

We've provided some unit tests for the partial load and store subcircuits. These are not comprehensive. You can run these tests with:

```
bash test.sh test_partial_load
bash test.sh test_partial_store
```