

CS 61C Fall 2021

Part A: addi

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
<code>A</code>	32	Data to use for Input A in the ALU operation
<code>B</code>	32	Data to use for Input B in the ALU operation
<code>ALUSel</code>	4	Selects which operation the ALU should perform (see the list of operations with corresponding switch values below)

Output Name	Bit Width	Description
<code>Result</code>	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: <code>Result = A + B</code>
1	sll: <code>Result = A << B</code>
2	slt: <code>Result = (A < B (signed)) ? 1 : 0</code>

ALUSel Value	Instruction
3	Unused
4	xor: $\text{Result} = A \wedge B$
5	srl: $\text{Result} = (\text{unsigned}) A \gg B$
6	or: $\text{Result} = A \mid B$
7	and: $\text{Result} = A \& B$
8	mul: $\text{Result} = (\text{signed}) (A * B)[31:0]$
9	mulh: $\text{Result} = (\text{signed}) (A * B)[63:32]$
10	Unused
11	mulhu: $\text{Result} = (A * B)[63:32]$
12	sub: $\text{Result} = A - B$
13	sra: $\text{Result} = (\text{signed}) A \gg B$
14	Unused
15	bsel: $\text{Result} = B$

Some additional tips:

- You might find bit splitters or extenders useful when implementing shift operations.
- 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

Here's a [companion video](#) that goes with this section.

We've provided some tests for each task, located in subdirectories under `tests/`. For example, the ALU tests are in `tests/part-a/alu/`. When these tests are run, the outputs from your circuits are saved in a `tests/part-a/alu/student-output/` subdirectory.

For example, to run the ALU tests:

```
$ python3 test.py tests/part-a/alu/
```


You can also specify a single test circuit, or a grandparent/great-grandparent directory:

```
$ python3 test.py tests/part-a/alu/alu-add.circ  
$ python3 test.py tests/part-a/  
$ python3 test.py tests/
```

After the tests finish running, your ALU circuit's outputs will be saved under `tests/part-a/alu/student-output/` with a `-student.out` suffix (e.g. `alu-add-student.out`). The corresponding reference outputs can be found at `tests/part-a/alu/reference-output/` with a `-ref.out` suffix (e.g. `alu-add-ref.out`).

We've also provided `format-output.py`, which accepts a path to an output file and displays the output in a more readable format (left-aligned **hexadecimal** numbers). For example, to get the reference output of the `alu-add` sanity test in readable format, you would do:

```
$ python3 tools/format-output.py tests/part-a/alu/reference-output
```



If you want to see the difference between your output and the reference solution, you can put the readable outputs into temporary files and `diff` them. For example, for the `alu-add` test, you would do:

```
$ python3 tools/format-output.py tests/part-a/alu/reference-output
$ python3 tools/format-output.py tests/part-a/alu/student-output/a
$ diff reference.out student.out
```

Note: If the lines are wrapping, try resizing your terminal window (or try a slightly smaller font). Or see the following note.

Experimental Note

Here's a [companion video](#) that goes with this note.

Each output file is technically a valid CSV file, so you can also import the output in a spreadsheet app if you *really* want to crunch the numbers (or you really hate tables in terminal). If the app requires a `.csv` extension, you can use

```
$ cp tests/part-a/alu/student-output/alu-add-student.out student.c
```

and import the resulting `.csv` file.

Debugging

Here's a [companion video](#) that goes with this section.

Similar to how you can step through your C code in GDB, you can also step through the test circuits in Logisim! For this example we'll be using the `alu-add` test.

Open `tests/part-a/alu/alu-add.circ` in Logisim. Among other things, one ROM (read-only memory) each feeds into the `Input_A`, `Input_B`, and `ALUSel` tunnels. These tunnels then feed into your ALU near the upper right. The ROMs contain corresponding values for the test being considered. Every clock cycle, the adder on the top left increments by one, which advances each ROM by one entry, thus feeding the next set of inputs to your ALU. If you tick the circuit a couple times (`File -> Tick Full Cycle` or the corresponding keyboard shortcut), you can see the test circuit advance through each set of inputs and your ALU's corresponding outputs. If you want to start over, use `Simulate -> Reset Simulation` (`Command/Control + R`).

Now, let's see what your ALU is actually doing with the inputs. Right-click your ALU, and select **View alu**. Your ALU circuit will appear, with the input values for the current test cycle already on the ALU input pins. With this, you can see exactly what your ALU is doing in every line from the output files! The **Poke Tool** will be very useful here.

Note: Edits to the test circuit, including the ALU we just inspected, **will not be saved**. 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
<code>rs1</code>	5	Determines which register's value is sent to the <code>Read_Data_1</code> output
<code>rs2</code>	5	Determines which register's value is sent to the <code>Read_Data_2</code> output
<code>rd</code>	5	The register to write to on the next rising edge of the clock (if <code>RegWEn</code> is 1)
<code>Write_Data</code>	32	The data to write into <code>rd</code> on the next rising edge of the clock (if <code>RegWEn</code> is 1)
<code>RegWEn</code>	1	Determines whether data is written to the register file on the next rising edge of the clock
<code>clk</code>	1	Clock input

Output Name	Bit Width	Description
<code>Read_Data_1</code>	32	The value of the register identified by <code>rs1</code>

Output Name	Bit Width	Description
Read_Data_2	32	The value of the register identified by <code>rs2</code>
<code>ra</code>	32	The value of <code>ra</code> (x1)
<code>sp</code>	32	The value of <code>sp</code> (x2)
<code>t0</code>	32	The value of <code>t0</code> (x5)
<code>t1</code>	32	The value of <code>t1</code> (x6)
<code>t2</code>	32	The value of <code>t2</code> (x7)
<code>s0</code>	32	The value of <code>s0</code> (x8)
<code>s1</code>	32	The value of <code>s1</code> (x9)
<code>a0</code>	32	The value of <code>a0</code> (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

We've provided the autograder RegFile tests in the `tests/part-a/regfile/` directory. You can run these with:

```
$ python3 test.py tests/part-a/regfile/
```

Refer to the testing section of Task 1 for more information on test outputs.

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
<code>inst</code>	32	The instruction being executed
<code>ImmSel</code>	3	Value determining how to reconstruct the immediate (you can ignore this for now)
<code>imm</code>	32	Value of the immediate in the instruction (assume the instruction is <code>addi</code> for now)

Output Name	Bit Width	Description
<code>imm</code>	32	Value of the immediate in the instruction (assume the instruction is <code>addi</code> for now)

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
READ_DATA	32	Data at WRITE_ADDRESS from memory
INSTRUCTION	32	The instruction at memory address PROGRAM_COUNTER
CLOCK	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)
tohost	32	The contents of CSR
WRITE_ADDRESS	32	The address in memory to read from or write to
WRITE_DATA	32	Data to write to memory
WRITE_ENABLE	4	The write enable mask for writing data to memory
PROGRAM_COUNTER	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 `PROGRAM_COUNTER` 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?

`addi` is an I-type instruction. The fields are:

- `imm` [31-20]
- `rs1` [19-15]
- `funct3` [14-12]
- `rd` [11-7]
- `opcode` [6-0]

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

Use the splitter to extract each of the 5 fields from the instruction.

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

The `rs1` bits you split from the instruction should connect to `rs1` on the regfile. The `rd` bits you split from the instruction should connect to `rd` on the regfile. I-type instructions don't have `rs2` so we can ignore `rs2` for now. Remember to connect the clock to the register file!

▼ What needs to be connected to the immediate generator?

Connect the `INSTRUCTION` to the immediate generator. Your immediate generator from the previous task should take the instruction and output the correct immediate for you.

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?

Input `A` should be the `read_data_1` from the regfile.

Input `B` should be the immediate from the immediate generator.

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

`ALUSel` selects which computation the ALU will perform. Since we only care about implementing `addi` for now, we can hard-code ALU to always select the `add` operation (`ALUSel = 0000`).

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?

`addi` takes the result of the addition computation (from the ALU output) and writes it to the register `rd`.

Connect the ALU `Result` to `Write_Data` on the regfile.

Since the `addi` instruction always writes to a register, you can hard-wire `RegWEn` to `1` for now so that register writes are always enabled.

Testing

Here's a [companion video](#) that goes with this section.

Each CPU test is a copy of the `run.circ` file included with the starter code that has instructions loaded into its IMEM. When you run the `test.py` script, it runs Logisim in the background. The clock ticks, the program counter is incremented, and the values in each of the outputs is printed to a `.out` file in the `student-outputs` directory.

Let's take the single-cycle `cpu-addi-basic` sanity test as an example. It has 4 `addi` instructions (see `tests/part-a/addi/inputs/cpu-addi-basic.s`). Open `tests/part-a/addi/cpu-addi-basic.circ` in Logisim, and take a closer look at the various parts of the test file. At the top, you'll see the place where your CPU is connected to the test outputs. With the starter code, you'll see lots of `UUUUs` or `XXXXs`; when your CPU is working, this should not be the case. Your CPU takes in one input (`INSTRUCTION`), and along with the values in each of the registers, it has an additional output: `PROGRAM_COUNTER`, or the address of the instruction to be fetched from IMEM to be executed the next clock cycle.

As you can see, there are many specifically-positioned wires connected to specific input/output pins on your CPU. Make sure that you **do not** edit the provided input/output pins or add new ones, as this will change the shape of the CPU circuit, and as a result the connections in the test files may no longer work properly.

Below the CPU, you'll see instruction memory. The hex for the `addi` instructions has been loaded into instruction memory. Instruction memory takes in one input (called `PROGRAM_COUNTER`) and outputs the instruction at that address. `PROGRAM_COUNTER` is a 32-bit value, but because Logisim caps the size of ROM units at 2^{16} bytes, we have to

use a splitter to get only 14 bits from `PROGRAM_COUNTER` (ignoring the bottommost two bits). Notice that `PROGRAM_COUNTER` is a **byte address**, not a word address.

So what happens when the clock ticks? Each tick of the clock increments an input in the test file called `Time_Step`. The clock will continue to tick until `Time_Step` is equal to the halting constant for that test file (for this particular test file, the halting constant is 6). At that point, the `test.py` script will print the values in each of the outputs to the respective `.out` file. Our tests will compare this output to the expected; if your output is different, you will fail the test.

We've provided the autograder tests for `addi` (Task 3) in the `tests/part-a/addi/` directory. You can run these with:

```
$ python3 test.py tests/part-a/addi/
```

Task 5: Part A README Update

Your last task for Part A is to fill in the `README.md`. Write down how you implemented your circuits and components for this part (including ALU and RegFile, since you used them for `addi`!), and explain the reasoning behind your design choices. There's no specific format or length requirement here, so feel free to get creative!

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

☐ Dark Mode