ELEC3441: COMPUTER ARCHITECTURE
SECOND SEMESTER, 2023–24

Homework 1 (r1.0)

Due:

- `Part (A) -- 13 Mar, 2024, 11:59pm`
- `Part (B) -- 13 Mar, 2024, 11:59pm`

Instruction: Submit your answers electronically through Moodle.

There are 2 parts in this homework. Part A includes questions that aim to help you with understanding the lecture materials. They resemble the kind of questions you will encounter in quizzes and the final exam. Part A is an individual portion of this homework. You should be completing and turning in *your own work*.

Part B of this homework are hands-on exercises that require you to design and evaluate processor systems using various software and hardware tools, including the RISC-V compilation tool chains. They are designed to help you understand real-world processor design and the use of various tools to help you along the way. Part B is a group work. Each group should turn in only 1 set of project file and 1 project report.

The following summarizes the 2 parts.

Part	Type	Indv/Grp
A	Basic problem set	Individual
B	Hands-on	Individual or Group of 2

In all cases, you are encouraged to discuss the homework problems offline or online using Piazza. However, *you should not ask for* and *you should not give out solution directly* as that defeat the idea of having homework exercise. Giving out answers or copying answers directly will likely constitute an act of plagiarism.

Part A: Problem Set

A.1 Iron Law

Consider the 3 programs P, Q and R running in processor A and B with the following number of instructions:

Processor	Class	ALU	Branch/Jumps	Load/Store
A	Cycles per instruction	1	4	20
B	Cycles per instruction	1	20	12
A&B	# instructions in P	1 000 000	300 000	500 000
A&B	# instructions in Q	500 000	250 000	125 000
A&B	# instructions in R	1000	3000	2000

HINT: use a spreadsheet program to help with answering this question.

A.1.1 What is the average CPI of program P, Q, R, when executed on processor A and B?

A.1.2 Processor A runs at 1.5 GHz while processor B runs at 1.2 GHz, double the clock rate. Consider only the 3 programs P, Q and R individually, which processor results in higher performance, and by how much?

A.1.3 When considering all three programs as a whole, which processor is faster? On average (geometric mean), how much faster is the faster processor?

A.1.4 You are allowed to improve the speed of only 1 of the three instruction types (ALU, Branch/Jump, Load/Store) of the slower processor. Assuming the clock frequency remains unchanged, what is the maximum speedup possible when using the *total runtime* (not geometric mean) for the three programs?

A.1.5 If the geometric mean of speedup between the processors among the 3 programs were to measure how good a processor is, would that affect your choice of instruction class to improve on? If so, how does it affect your answer? If not, explain why it doesn't matter whether *total runtime* or *geometric mean* is used?

A.2 Benchmark Performance

In class, we mentioned it is necessary to use geometric means when considering normalized performance of multiple benchmark programs. In this question, you will explore further how various ways of performance evaluation can be performed. For those of you who are interested, a lot of the ideas in this question can be traced back to this paper `http://www.eee.hku.hk/~elec3441/sp23/handout/Fleming_Wallace_86.pdf`.

Now, you are given 3 processors A, B, C, and 5 benchmark programs $P1$ to $P5$. The following summarizes the *run time* of these programs in the 3 processors. In all cases, both the raw CPU times (in seconds) and the normalized time relative to processor A are shown. *HINT:* use a spreadsheet program to help with answering this question.

	Proc. A		Proc. B		Proc. C	
Benchmark	raw time	norm. time	raw time	norm. time	raw time	norm. time
P1	20 000	1	40 000	2.00	80 000	4.00
P2	80 000	1	20 000	0.25	40 000	0.50
P3	40 000	1	80 000	2.00	20 000	0.50
P4	1200	1	2400	2.00	600	0.50
P5	12 000 000	1	24 000 000	2.00	6 000 000	0.50

A.2.1 Consider the first 3 benchmark programs ($P1$ to $P3$) only in this part. For each processor, compute the following:

(i) Arithmetic mean of *raw CPU time* of the benchmark programs;
(ii) Geometric mean of *raw CPU time* of the benchmark programs;
(iii) Arithmetic mean of *normalized CPU time* of the benchmark programs;
(iv) Geometric mean of *normalized CPU time* of the benchmark programs;
(v) Total raw CPU time;
(vi) Total normalized CPU time;

A.2.2 Follow up from previous part. Examine the benchmark results closely, and you will notice that processor A, B, and C, is fastest in exactly 1 of the 3 benchmark. As a result, a reasonable conclusion is that all three processors are equally good.

Now, consider each of 6 results you obtained in A.2.1 and answer the following:

1. What is the conclusion that each metric is suggesting? For example, consider the *total run time* of the 3 programs, which processor is fastest?
2. Which metric gives you the answer that the 3 processors perform equally well?
3. Which metric would you choose to use for a fair comparison of the performance of the processors?

A.2.3 Now, repeat the previous 2 questions but with benchmark programs $P1$ to $P4$. That is, you add benchmark program $P4$ into the mix and recompute the results in various metrics.

You would notice that Processor C is fastest in program $P4$. As a result, considering all 4 programs, a reasonable conclusion is that since C is fastest in 2 programs, while A and B are fastest in only 1, then C should be the fastest processor. Which of the metrics gives you this conclusion?

However, if you consider all 6 different ways of comparison, some of them may lead to a different conclusion that *may* also be reasonable. What other processor can you argue is having the best performance when considering $P1$ to $P4$?

A.2.4 Now, if you consider only $P1$ to $P3$ and $P5$ (i.e., skipping $P4$), how would that affects your above conclusions? Note the runtime values of $P5$ are exactly 10 000 times of $P4$.

A.2.5 Considering using benchmark $P1$ to $P5$, plus ONE ADDITIONAL benchmark of your choice ($P6$), is there a way to make Processor B stands out as being the fastest? If yes, let the run time of $P6$ in processor A, B, C, be t_A, t_B, t_C respectively, suggest values of t_A, t_B, t_C and the performance metric to use that will make Processor B become fastest. If it is impossible, explain why is it the case.

A.3 RISC-V Assembly Programming

In this question, you will practice writing assembly programs using the RISC-V assembly code. In all cases, assume the C variables are stored in the following registers:

C Declaration	Variable	Register
`unsigned x`	`x`	`a0`
`unsigned y`	`y`	`a1`
`unsigned z`	`z`	`a2`
`unsigned a[16]`	`a`	`a3`

A.3.1 Arithmetic and logic operations Implement the following C code segment as RISC-V assembly code. When in doubt, check with the RISC-V gcc toolchains you learned in Lab 1.

- `z = z + (x - (y << 1))`
- `y = x + 0x757`
- `y = x + 0x7800A757`
- `a[3] = a[2] + 1`
- `a[x & 0xF] = y % 16` *Hint:* you don't need to use modulo operator.
- `z = 4*y + 2*x` *Hint:* you don't need to use multiplication.

A.3.2 Branches and Jumps Implement the following C code segment as RISC-V assembly code.

-
  ```
  if (x > y) {
      z = x - y;
  } else {
      z = y - x;
  }
  ```

-
  ```
  if (x < y && x > z) {
      x = x + a[x];
  }
  x = x + y + z;
  ```

-
  ```
  for (x = 0; x < 10; x++) {
      a[x] = x;
  }
  ```

-
  ```
  x = 0;
  while (x <= 10) {
    a[x] = y;
    y++;
    x++;
  }
  ```

A.4 Fibonacci Numbers

The Fibonacci series is a series of numbers with many interesting Mathematical properties (See `http://en.wikipedia.org/wiki/Fibonacci_number` if you are curious.) In general, the n-th Fibonacci number F_n can be obtained by the following formular:

$$\begin{cases} F_n = F_{n-1} + F_{n-2} & n = 2, 3, \ldots \\ F_0 = 0 \\ F_1 = 1 \end{cases}$$

A.4.1 The following C code computes the n-th Fibonacci number by computing the entire series and returning the n-th Fibonacci number. You can assume that $1 < n < 100$ in this question.

```
int F[100];
int fib(int n, int F[]) {
  int i;
  F[0] = 0;  F[1] = 1;
  for (i = 2; i < n+1; i++) {
    F[i] = F[i-1] + F[i-2];
  }
  return F[n];
}
```

Implement the above `fib` function using RISC-V assembly instructions. You can assume `n` is stored in `a0` and `F` is stored in `a1`. Your return value should be stored in `a0`.

A.4.2 Alternatively, the n-th Fibonacci number can also be computed recursively as follows:

```
int fib(int n) {
  int r;
  if (n < 2) {
    return n;
  }
  r = fib(n-1) + fib(n-2);
  return r;
}
```

Implement the above `fib` function using RISC-V assembly instructions. You can assume `n` is stored in `a0`. Store your return value (`r`) in `a0`. Use the `jal` and `jalr` instructions explained in class to implement function calls.

A.5 One Instruction Set Computer

An one instruction set computer (OISC) is an extreme form of a RISC ISA — it has only 1 instruction. In this question, consider one such machine called SUBLEQ. A SUBLEQ machine has exactly one instruction that "subtracts and branches if less than or equal to zero" (called `subleq`):

```
subleq a, b, c
```

A `subleq` instruction has 3 operands, `a`, `b` and `c`, which are all *memory addresses*. It has the following semantic:

```
memory[b] = memory[b] - memory[a]
if (memory[b] <= 0) goto c
```

A.5.1 Based on the 3 terms of the Iron Law: (i) instruction per program (ii) cycle per instruction (iii) cycle time, compare the relative performance between the SUBLEQ processor and the RISC-V processor (both pipelined and non-pipelined) shown in class. For example, in terms of instructions per program, which processor is likely be higher/lower? Make any reasonable assumptions as needed, but please state in your answer. *Hint:* Pay attention to the main memory and how it will affect the various aspects of the Iron Law. You may use the magic memory of A.5.2 as a reference, or you can imagine your own.

A.5.2 By following a similar process of constructing the *single-cycle* RISC-V processor as shown in class, design a hardware implementation of the SUBLEQ processor. Assume you have the following magical memory for use in your SUBLEQ imlementation: The memory space is 64 KiB, i.e., memory address is 16 bit wide. Memory is *byte-addressable*. The memory has 3 *read* ports and 1 *write* port that can be operated on each cycle. 2 of the read ports (`ra` and `rb`) return 1 word. The third read port `ri` returns 3 words in 1 cycle. Write port writes 1 word at a time. Each word is 16 bit wide.

A.5.3 Implement the operation of the SUBLEQ processor using RISC-V instructions. You can assume `a`, `b`, `c` are all 32-bit unsigned numbers stored in registers `a0`, `a1` and `a2` respectively.

Part B: Hands-on Exercise

B.1 RISC-V Processor in Logisim

In this question, your task is to build a complete functional single cycle RISC-V processor similar to that shown in class. Most of the datapath for the single cycle processor design is already provided to you, and we have implemented support for R-type instructions for you as an example. You will have to add support for other types of instructions. Your main task will be to complete the instruction decoder, but some additional data connections will also be needed by modifying the existing design.

For this homework, you will be using a program called Logisim-Evolution. Logisim-Evolution is a fork of the original Logisim software from Carl Burch, which is an educational tool for designing and simulating digital circuit. It is free, and you can download the latest version from its homepage `https://github.com/reds-heig/logisim-evolution`. On its homepage you can also find online documentation about the tool.

For a brief tutorial on using Logisim-Evolution, please refer to Lab 3.

For this hands-on exercise, we will provide you with precompiled memory images to load into the instruction and data memory of the processor that you will be building. Translating from assembly to memory image is done with the RISC-V compilation tool-chain. Any tools you need have been installed on a departmental server called `tux-1`. Since they are all freely available open-source software, so you may also choose to install them on your own machine.

If you wish to generate your own test code for the processor, you will need a Unix system with the RISC-V cross-compilation GCC toolchain configured to compile to the RV32I subset only. For this optional part, we recommend that you use the toolchain already installed on `tux-1.eee.hku.hk`.

B.1.1 Getting the Files You will need the files located inside the `logisim` subdirectory in your downloaded file `elec3441hw1.tar.gz` in {`HW1ROOT`}`/logisim`. It contains the following files:

- `logisim`
  - `rv32-addi.circ` ← Logisim source
  - `test` ← Test programs
    - `addsub.s` ← Assembly file
    - `addsub.mem` ← Memory content file for use with Logisim
    - ...
    - `s2mem.sh` ← Converts .s to .mem (requires riscv-gcc)

B.1.2 Subset of RISC-V ISA To keep your design simple, you only need to implement a subset of the RISC-V ISA as shown below. Of course, if you are feeling adventurous, you are more than welcome to implement and try out additional instructions.

31 25	24 20	19 15	14 12	11 7	6 0	
0000000	rs2	rs1	000	rd	0110011	`add rd,rs1,rs2`
0100000	rs2	rs1	000	rd	0110011	`sub rd,rs1,rs2`
imm[11:0]		rs1	000	rd	0010011	`addi rd,rs1,imm`
imm[31:12]				rd	0010111	`auipc rd, imm`
imm[11:0]		rs1	010	rd	0000011	`lw rd,imm(rs1)`
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	`sw rs2, imm(rs1)`
imm[12][10:5]	rs2	rs1	000	imm[4:1][11]	1100011	`beq rs1,rs2,imm`
imm[12][10:5]	rs2	rs1	001	imm[4:1][11]	1100011	`bne rs1,rs2,imm`
imm[12][10:5]	rs2	rs1	100	imm[4:1][11]	1100011	`blt rs1,rs2,imm`
imm[12][10:5]	rs2	rs1	101	imm[4:1][11]	1100011	`bge rs1,rs2,imm`
imm[20][10:1][11][19:12]				rd	1101111	`jal rd, imm`
imm[11:0]		rs1	000	rd	1100111	`jalr rd,imm(rs1)`

Figure B.1: Subset of RISC-V instruction to be implemented in Logisim

B.1.3 Basic Data Path – Add You will begin by the examining the design of a minimal processor that supports only the `add, sub, and, or` instructions as shown in Figure B.2. It contains the main components we have seen in class: an instruction fetch unit, an instruction memory, an instruction decoder, a register file, an ALU, and a data memory. Open the file `rv32-addi.circ` in Logisim.

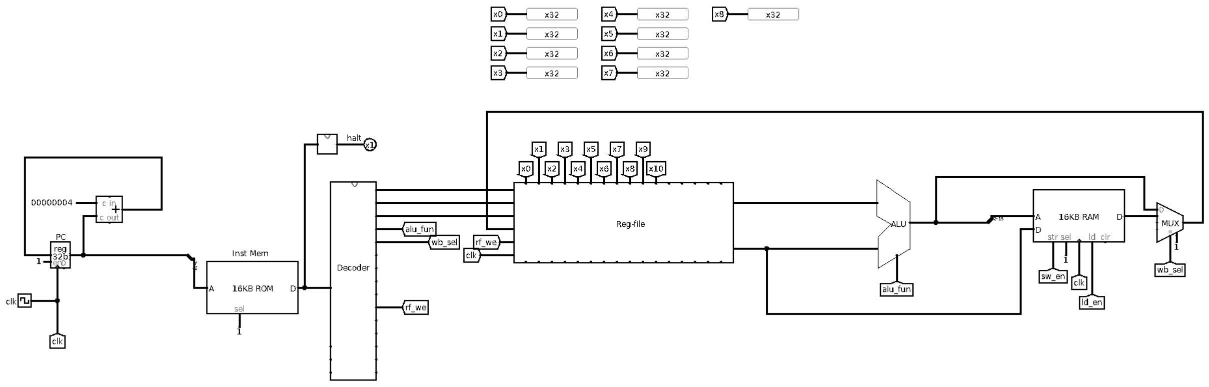

Figure B.2: Data Path

Instruction Fetch Unit

The heart of the instruction fetch is the program counter (PC) and an adder to compute the next instruction address (PC+4). In Logisim, PC can be implemented using a **register**. You can find the library component **register** from the **memory** library that you can see on the left hand side of Logisim windows. As we are implementing a 32-bit architecture, the bit width of the register is set to 32. You can click on the component with the edit tool to view the property "Data Bits" in the lower half of the sidebar on the left.

To compute the next PC, the built-in component **adder** from Logisim is used to add the constant 4 to the current PC value. Note that there is not yet a way to choose a value other than PC+4: you will have to modify this path when implementing jump and branch instructions in part C.

Instruction Memory

To implement the instruction memory of a processor, we use the Logisim built-in **ROM** unit. You can find a ROM unit under the built-in **memory** library on the left.

While we have a 32-bit address space, we do not want to implement 4 GB of memory in this small simulated system. Instead, the ROM component is set to 64 kB of memory. As a result, the instruction memory will have a 16 bit address bus.

However, in the RISC-V processor (as well as most other processors), memory is **byte addressable**. That is, each memory address corresponds to 1 byte. However, in the case of Logisim, as well as in most real processors, the memory returns one **32-bit word** at a time. Therefore, the address signal passed to the instruction ROM should omit the lowest 2 bits, which are always `00` in *aligned* word addresses anyway. If an individual byte is needed (such as in the case of load byte (`lb`)), the lowest 2 bits will then be used to select the correct byte from the output word. In this assignment we only deal with 32-bit words, so we don't need this capacity for now.

That is why you can see:

Address Bit Width	14
Data Bit Width	32

in the parameters for the ROM unit. A **splitter** is used to select bits 2-15 to address the memory component.

Instruction Decoder

This is the component that decodes the fetched instruction from the instruction memory and generates the corresponding control signals. For the sample `add` instruction, it extracts the following:

- Register number of `rs1`, `rs2` and `rd`.
- The corresponding `alu_fun` for the ALU

From the instruction encoding of `add` in Figure B.1, or equivalently the RISC-V ISA from the course website, you see that `rs1`, `rs2` and `rd` are located at `inst<19:15>`, `inst<24:20>`, and `inst<11:7>` respectively. You can also determine that it is an `add` instruction from its opcode at `inst<6:0>`, and in the case of ALU-OP, also `funct3` and `funct7`.

All the decoding logic should be implemented within the `inst_decoder` subcircuit. In Logisim, you can edit a subcircuit by right-clicking the component and selecting "View *component name*", or by double-clicking the component name in the left-hand sidebar. Take a look at the provided instruction decoder (`inst_decoder`).

For each ALU operation `add, sub, and, or`, we create an individual 1-bit signal is high when we encounter an instruction that uses this function. Observe how we use comparators to identify the instruction. The signals are fed into a component called **priority encoder** (lower left) which will output the number of the first port with a signal set to 1. For example, if the `and` signal is 1 and the others are 0, it will output "3", because `and` is connected to port "Input 3". If you look inside the ALU component and compare the mux that selects `alu_out`, you will see that the inputs to the MUX are connected in the same order: the **and32** component is also connected to port 3. So the priority encoder generates the right signal to select the same input port on the mux as is high on its own input ports.

Register File

The register file of a RISC-V processor can essentially be modeled as a RAM with 32 locations. The only difference is that it also has to support 2 read and 1 write operations at the same cycle, and to hardwire the value 0 to register `x0`. A register file has been created for you. Take a good look at the `rf32` subcircuit to understand how it works.

The register file has debug ports `dx0-dx31` that allow you to peek at the contents of the registers so you can see if your circuit is functioning correctly. Above the register file, we have connected probes that display the hexadecimal value of the registers `x0-x8`. This will be your main interface for debugging when you implement the missing instructions.

ALU

The final and arguably most important component in the processor to implement the R-type instructions is the ALU. In the provided design, a simple ALU has been created for you. Its job is to provide the required function (addition, subtraction, etc.) as specified by the `alu_fun` signal.

Data Memory

Since the `lw` and `sw` instructions are not yet implemented, the data memory is not in use yet. We have however already added it to the circuit so that it is ready for use in the next steps.

The data memory is the same size and layout as the instruction memory, however this time we use a **RAM** component as it also has write capability. This means an additional port for write data and a store enable signal that tells the component to write the data to the address given on the address port at the next rising clock edge.

With all the above components correctly connected, you have a complete data path for the R-type instructions.

B.1.4 Testing Your Processor To test that the processor works correctly, you should manually load the program into the instruction memory. To do so, right click the ROM component corresponding to the instruction memory, and select **Load image**. Here is where you can load your test program directly into Logisim using its own memory image format.

Load the memory image file `addsub.mem` in the `tests` directory. It contains the following instructions:

```
add x5, x1, x2
sub x6, x3, x4
and x7, x5, x6
or  x8, x5, x6
ebreak          # indicates end of program
```

Once it is loaded, you can try executing the processor by toggling the clock signal using the **poke** tool. When you execute the `ebreak` instruction, the output `halt` (located above the decoder) will turn high, so you can recognize that you have reached the end of the program.

Of course, with only support for register-register `add/sub/and/or`, it is not going to be very interesting as all the values are zero. Before you have support for immediate instructions in next step, you can manually change the values of `x1`, `x2`, `x3`, and `x4` by going inside the **rf32** component.

B.2 Adding ADDI Support

In the following section, we will guide you through adding support for the *add immediate* instruction. You will need to change the internals of the instruction decoder, as well as add an operand source to the ALU. To test your processor, you can use the test program `addi.mem`.

B.2.1 Modification to the data path First, take a look at the data path. The `addi` instruction is very similar to the `add` instruction. Our ALU already knows how to add two operands. However, you will need to add a path that will allow the sign-extended immediate value to reach the ALU.

Break the connection between the rd2 port of the Reg-file and the lower input of the ALU. Find the **multiplexer** component in the built-in **Plexers** library and add it to the circuit. You are choosing between 32-bit operands, so in the properties sidebar, for "Data Bits", enter 32. Because we are thinking ahead for the future, when there are other possible sources of immediate values, we want a larger select signal. For "Select Bits", enter 2. Change the value of "Include Enable?" to No.

Connect the first port of the mux (Input 0) to the output rs2 of the register file by drawing a direct connection. For the second port of the mux (Input 1) and the select port, the values will be coming from the instruction decoder. To avoid a mess of wiring in the diagram, use the Tunnel component (found under Wiring) to connect these ports. Any Tunnel components with exactly the same name will be considered connected, as if an invisible wire had been drawn between them.

Add a Tunnel facing East with 32 Data Bits. Label it `i_im`. Connect this tunnel to the lower port of the mux (Input 1). Create a second Tunnel, of 2 bits, and name it `op2_sel`. Connect it to the select port of the mux.

Now, locate the port `i_im` of the instruction decoder. Create another tunnel named `i_im` and connect it to this port. Do the same for the port `op2_sel` of the instruction decoder. Now, the instruction decoder can control the second input to the ALU.

B.2.2 Modification to the instruction decoder The data path now has the necessary connections to execute the `addi` instruction, but you still need to add the control signals to make it do so. Right-click on the decoder and select "View inst_decoder" to edit the instruction decoder.

You need to create circuits that set the values of the following outputs to the instruction decoder:

1. `i_im`: First, extract the immediate value `i_im` from the instruction. On the left hand side of the circuit, you can see that the instruction has already been decomposed into the fields `op, rd, fun3, rs1, rs2, fun7`. For I-type instructions, the immediate value is the combination of the `rs2` and `fun7` fields. Find some empty space and create tunnels for `rs2` and `fun7`, facing east, and `i_im`, facing west. Inbetween, place a Splitter component (also found under Wiring). For its parameters, select Facing West, Fan Out 32, Bit Width In 32. This component now has one 32-bit output and 32 1-bit inputs. Connect its output to the `i_im` signal.

   Place another splitter, Facing East, Fan Out 5, Bit Width In 5. Connect its input to the `rs2` signal. Now, connect bits 0-4 of `rs2` to bits 0-4 of `i_im`.

   Using another splitter, connect bits 0-6 of `fun7` to bits 5-11 of `i_im`. Remember that the `i_im` value is sign-extended: connect bits 12-31 of `i_im` appropriately.

2. `alu_fun`: For the `addi` instruction, the value of the control signal `alu_fun` should be the same as for the `add` instruction. There is an OR gate at the end of the circuit generating the `add` signal, even though it has only one input so far. Create a 1-bit tunnel named `addi` and connect it to another port of this OR.

   To set this signal at the right time, you need to detect that the current instruction is an `addi`. Take a look at the circuits that generate the `add`, `sub`, `and` and `or` signals. Create a similar circuit that will detect an `addi` instruction, and connect its output to a tunnel named `addi`.

3. `op2_sel`: This signal tells the mux added to the datapath in part B.2.1 to use the immediate value for the `addi` instruction, and the register value otherwise. In an empty space, create an east-facing 1-bit label `addi` and a west-facing 2-bit label `op2_sel`. Add a **priority encoder** from the "Plexers" library, and set its "Select Bits" parameter to 2. Connect the output of the priority encoder to `op2_sel`. Create a constant 1 and connect it to the enable port at the bottom of the priority encoder.

   When the instruction is `addi`, we want the output to be 1. Connect the `addi` signal to Input 1 of the priority encoder. When the instruction is one of `add, sub, and, or`, we want the output to be 0. Create a circuit that generates 1 when the instruction is an R-type instruction and 0 otherwise. Connect this circuit to Input 0 of the priority encoder.

B.2.3 Testing ADDI functionality To check that your implementation of `addi` works correctly, load the memory image file `addi.mem` in the `tests` directory into your instruction memory and toggle the clock until the `halt` signal is high.

Read the associated assembly file `addi.s` and trace through the instructions manually. Compare the displayed values of the registers `x1-x5` with what you would expect by tracing through the instructions.

B.2.4 Submission Submit your completed `rv32-addi.circ` online.

B.3 Non-branch/jump Instructions

So far, you have a single cycle processor with basic ALU operation support. In this part, you will add memory operations, another immediate operation, and the load upper immediate instruction to your processor.

Make a copy of the file `rv32-addi.circ` you have completed in the previous section. Rename it `rv32-isu.circ` and open the new file.

Your task is to complete the data path design `rv32-isu.circ`. In particular, you need to add support for:

- S-type instructions `sw, lw`
- U-type instruction `auipc`

In other words, your processor should have support for all instructions except for branches and jumps. Some hints for the implementation:

B.3.1 Add upper immediate to pc (auipc) The `auipc` instruction is utilized to support pc-relative addresses with U-type format. This instruction constructs a 32-bit offset from 20-bit U-immediate filling in the lowest 12 bits with zeros, and adds the offset to the *pc*. You need to think about how to support a U-type immediate value, where to add the offset to PC in the datapath and how to write back the result to register for this instruction.

B.3.2 Memory Operations (lw, sw) The `lw` and `sw` operations interact with the data memory, which was unused so far. The connections in the data path were already made for you, but there are several control signals which were previously defined in an overly simplistic manner that you need to change.

To test the `lw` instruction, use the file `lw.mem`. If you examine the corresponding `lw.s`, you will notice that this file contains a `.data` section. This means that certain values are predefined in memory. You will find them starting at address `0xA00` after assembly. In order for these values to be visible to the data path, you will need to load the file `lw.mem` into both the instruction and the data memory.

This is also how it works in most real processors: instructions and data share a unified address space and are stored in the same DRAM memory, from which the instruction cache and the data cache get their contents.

B.3.3 Control Path In order to support various different instructions, one main task is to make changes to the instruction decoder. The instruction decoder is essentially a very large combinational block that determines the values of various control signals in the processor. In simple terms, it implements a control truth table similar to that shown in Figure B.3. The input to this is an instruction (or precisely, fields of an instruction such as `opcode`, `funct3`, `funct7`, etc).

The only exception is the input from branch conditions. It is the only datapath-generated signal that affects control of the processor. As a result, the control logic has to take into consideration such signals (in the next part).

All such control signals should be implemented inside the `inst-decoder` subcircuit in the design. We have already defined the output ports to this circuit. It may help to fill out the table below in full for all the instructions you need to implement, so that you can always tell what cases you need to cover for each signal.

B.3.4 Submission Submit the completed file `rv32-isu.circ` with support for R-type, I-type, S-type and U-type instructions.

Instruction	pc_sel	op2_sel	rf_we	alu_fun	...
add	0	0	1	1	...
sub	0	0	1	0	...
...	-	-	-	-	...

Figure B.3: Control Table

B.4 Complete Processor with Branches and Jumps

Your final task in implementing the single cycle processor is to include branch and jump support. For branches, you need to support all four instruction `beq, bne, blt, bge`. For jump, you need to support the `jal, jalr` instruction.

Make a copy of the file `rv32-isu.circ` you have completed in the previous section. Rename it `rv32-all.circ` and open the new file. Use your processor design from the previous section as a starting point and make the following modifications for branches:

- Add branch comparison logic.
- Add decoder logic to extract correct branch offset.
- Compute branch target.
- Modify instruction fetch unit such that it will select the correct next instruction depending on the branch comparison result.

For `jal` implementation, you need to include additional logic as follows:

- Add a UJ type immediate that extracts and rearranges the immediate offset from the `jal` instruction. Use it to compute jump target.
- Modify instruction fetch unit such that it will select the correct next instruction.
- Write the source address PC+4 to register file.

For `jalr` implementation, you need to include additional logic as follows:

- Use I type immediate that extracts from the `jalr` instruction to compute jump target.
- Modify instruction fetch unit such that it will select the correct next instruction.
- Write the source address PC+4 to register file.

B.4.1 Submission Make your modifications in the new file (`rv32-all.circ`) and submit the completed design online.

B.5 Note on Processor Verification

Throughout the design process, it is very important for you to keep testing the correctness of your design. The basic test procedure involves loading the test image in the instruction memory, execute the code, then compare the result in the regfile and/or the data memory against the expected output.

B.5.1 Test Input You test your processor by loading instructions into the instruction memory. Logisim memory takes a specific format, which we will standardize with a `.mem` extension. A number of test input files are located in `${HW2ROOT}/test`. There is a separate test file for each instruction you should implement, as well as some more complicated programs. Each `filename.mem` memory file is produced by the corresponding source assembly file `filename.s`. Read these files to deduce the expected output. Some files have a `.data` section. If this is the case, you need to load the same memory file into the data memory too for the execution to work correctly.

A script `s2mem.sh` is provided to generate the memory image from a source assembly file. To generate a memory image file `file.mem`, execute the following command:

```
tux-1$ cd ${HW1ROOT}/tests
tux-1$ ./s2mem.sh file.s
```

If you want to create your own test file, follow the format of one of the sample input. In particular, you must define the label `_start` to indicate the beginning of a program.

B.5.2 Loading/Saving Instruction Image To load the memory image, open the Logisim project and switch to the top module, i.e. the `main` circuit. Right click the instruction memory in the main circuit and select load image tab. Browse to the location of your `.mem` file and select OK.

Do the same thing if you have initial data that you want to load into data memory.

To save the memory content, you can **right click** the data memory then choose **edit contents**... tab and view data in data memory. Here you can choose **save image** tab to save data memory image as a file.

B.5.3 Simulating your design Click **Simulate→Reset Simulation** to reset the simulation.

Click the Poke tool ( ) which is right under the File menu. Now that the cursor is changed to an icon of a finger, click the `clk` pin and the simulation will proceed cycle by cycle. Note that the processor does not currently have any facility to stop, you will have to manually stop the simulation as needed.

To visually see the value of a particular signal, you can use a Logisim **probe**. Insert a probe from the **Wiring** library to your design and connect it to any net that you want to see its value. You can select to display the signal in binary, hexadecimal or signed/unsigned decimal numbers. Probes can be very helpful in debugging your circuit when you are in you initial testing phase. A few probes have been added to your design to illustrate how they can be used.

However, if you have a long running simulation, it may be too difficult to poke the `clk` signals repeatedly. For these cases, you may instead set up Logisim to automatically generate clock ticks by selecting **Simulate→Ticks Enabled**. Also, if you set up the correct watch point, you can use the Logisim logging facility to help debug your design. Please refer to the online documentation for more information.