Instructions

Deliverables There are two deliverables for this assignment: an Excel spreadsheet with your completed state transition table (`Control_Unit_FSM.xlsx`), and a Logisim circuit file named `cpu.circ` with your CPU implementation. Please submit commit these files to GitHub and submit via Gradescope. Only one member of each pair needs to do this, though do remember to add the other person's name to the submission.

Pair Programming You are required to work with your assigned partner on this assignment. You must observe the pair programming guidelines outlined in the course syllabus—failure to do so will result in both team members receiving a score of 0 for the assignment. *Collaboration across teams is prohibited and at no point should you be in possession of any work that was completed by a person other than you or your partner.*

CPU Data path

The first step in building your CPU is to ensure that you have the `datapath` and `control` circuits laid out according to the in-class activity. You should also copy an implementation of the ALU into the appropriate sub-circuit. Initially this can be your implementation from the previous homework, but once it is posted, please use the reference design.

Designing a Processor

In this assignment, you will finish building the CPU—at the end, you will have a working processor that is capable of executing simple stored programs.

Creating the State Transition Table

The Control Unit is a finite state machine that is responsible for governing the behavior of the rest of the CPU. It issues the enable and select bits that control how data moves through the ALU and the registers. In class, we started the process of constructing the state transition table for the control unit. Your primary task is to complete table within the provided Excel spreadsheet. The first four states, which perform fetch and decode have been provided for you. The remaining states govern the execution of each of the CPU instructions in our simple language.

Instruction	**State/Opcode**	**Description**
INPUT	00100	Reads the data on the input line and stores it in register A.
OUTPUT	00111	Copies the contents of register A to the OUT register.
JMP *val*	01010	Jumps to the location in memory that is specified by the byte that immediately follows the JMP instruction. In other words, the byte following the JMP instruction in the program is not another instruction, but a *value* that should be loaded into the PC next. Program execution will resume from this new PC value.
LOAD *val*	01110	Loads a value into register A. As with JMP, the value is specified by the byte that follows the LOAD instruction in memory.
INC	10100	Increments the value of A by 1.
MOV	10111	Copies the contents of register A to register B.
ADD	11010	Adds the values in A and B together, and stores the result in A.
HALT	11101	Halts execution—note that this can be accomplished by creating a self loop back to HALT, instead of returning control to the fetch state.
NOP	1111(0|1)	No operation—a do-nothing operation (like `pass` in Python) that is simply a placeholder.

Table 1: The instruction set for the processor.

The semantics of each of the CPU instructions are summarized in table 1; you will also want to reference the *ALU opcodes* table from the previous homework assignment. Work carefully to avoid errors. For each instruction, start by filling in the "Comments" column, describing what sequence of operations need to be performed to execute that instruction. Then devise the appropriate signal settings for the individual states. It might be a good idea for you and your partner to do this independently for each instruction and cross-check your answers, before moving on to the next instruction. Here are a few things to keep in mind as you fill out your spreadsheet:

- Do *not* read and write to a register on the same clock cycle. This can lead to timing bugs. Use one cycle to generate the result you would like—then, while holding all the control signals constant, assert the appropriate register's write enable line on the subsequent cycle.

- When you've filled in the control and next-state bits for a row in the spreadsheet, concatenate these digits into a single 16-bit binary number, and then translate it to a 4-digit hex value. For example, in state `00001`, we wish to produce the binary output `0b 0001 0001 1100 0000` which corresponds to `0x11C0`.
- The instructions have been appropriately spaced apart in the spreadsheet. You should have enough room to implement the necessary transitions within each instruction's allocated space. Don't be alarmed if you don't use every row.

Building the Control Unit

Our Control Unit FSM stores most of the information about its transition table in a block of Read-Only Memory (ROM). On each clock tick, we look up the next state and output signal settings from the ROM.

In the `control` subcircuit, we use a ROM block and two registers. To look up a value in the ROM, the SPC supplies the binary code for the current state, which is a 5-bit value—this is the "address" in ROM that we wish to access. The value that is returned by the ROM is a 16-bit number, that packs together the next state code (another ROM address) and the settings for the register Write Enables, the B Select and the ALU Select lines. To program the ROM, we need to place the hex value for each state in the corresponding location in the memory, so that when we look up that state, the ROM will output the control bits that we determined when filling out the transition table. To edit data in the ROM, right click on the Logisim component and select "Edit contents...". This will launch a new window that shows the current data values stored in your ROM (initially all zeros). Overwrite these with the hex values from your spreadsheet sequentially. The memory addresses increase left-to-right, top-to-bottom, i.e., memory address (state) 00000 is the cell at row 0, column 0, memory address 00001 is the cell at row 0, column 1, and so on. Once you complete this process, your state transition look-up table is complete!

The purpose of the two memory units in this CPU design can cause confusion, so to clarify:

- The *sequencer memory* (i.e., the ROM unit inside your control unit) contains the state transition table for your control unit. It tells your CPU how to execute a given instruction from the instruction set (table 1), and is therefore read-only.

- The *program store* or *user memory* (i.e., the RAM unit in your datapath) contains the user program and data. The contents of this RAM block will be rewritten by the user when they wish to run a new program.

Now for some final implementation details. Observe that the ROM component does not accept a CLK input: this indicates that this is an *asynchronous* device, i.e., when you change the address line, the data line will respond immediately. This would be a problem for the remainder of our clocked CPU as it could cause the control lines values to fluctuate in undesirable ways. This is the reason for buffering the inputs and outputs of the ROM using registers (the *Sequencer Program Counter* (SPC) and the *Instruction Register* (IR)) that trigger on opposite halves of the clock cycle.

Let's take a step back to understand how all the pieces fit together: when the dispatch signal is 1, on the rising clock edge, the current value of the MBR (i.e., the code for the instruction we wish to execute) will get copied into the SPC. We will perform a ROM look-up using this instruction encoding, and the result will appear on the ROM data line after a slight delay. Then, on the falling clock edge of the *same* cycle, the 16-bit output from the ROM will be copied into our IR. The output of the IR will get split into its components and sent to the appropriate destinations. Thus, we will have read the current state of our FSM, and determined our outputs and the next state value, all within a single clock cycle, exactly as desired.

Finally, you may have noticed that the Control Unit also generates a single bit "Halt" output—this output should go high if and only if the current instruction under execution is the HALT instruction. You will need to add some logic gates that take as input the value in the SPC register, and determine whether that state is the HALT state.

The CPU: Putting It All Together

Once you have built the control unit, you need to verify that it is properly integrated with the rest of your CPU. With the Control Unit as a subcircuit in the datapath circuit, connect it as follows:

- Connect the outputs of your control unit to the appropriate write enable, ALU select, and mux select lines.
- Connect the halt output from the control unit to an output pin in the datapath named *halt* (all lowercase).

- Connect the clock signal shared among all your registers to the control unit.
- Connect the *RST* (reset) signal shared among all your registers to the control unit as well.
- Connect the lower five bits of the *MBR* register (bits 0-4) to the *MBR* input on your control unit (since our instruction opcodes are only 5 bits, we will disregard the upper three bits).

Congratulations—your hardware layout is complete! Now to take your CPU for a spin. First, take a look at the following assembly program and determine what it will do:

```
LOAD 1
OUTPUT
INC
JMP 2
```

To load this program into user memory, you will first have to translate each assembly language instruction into its corresponding *machine code*, i.e., binary code. Let's start with the first instruction. Looking at table **??**, we see that the opcode for `LOAD` is `01110`. As an 8-bit value, this is `00001110` (since our RAM stores data in 8-bit chunks, i.e., bytes) or `0x0e` in hexadecimal. So at location `0x00` in user memory, we place the hex code `0x0e`. According to our instruction set, the byte following a LOAD instruction should be the value to be loaded into register *A*—in this case, the value 1 which is `0x01`. So the value `0x01` goes in location `0x01` in user memory. At location `0x02`, we place the opcode for OUTPUT (`0x07`), and so on. Once you've finished translating the entire program, reset your CPU. Then, simply start the clock to execute your program. Make sure to verify that *all* the instructions in the instruction set behave correctly by writing various short programs. For example, here's another program you might want to try: one that continuously reads the value on the *INPUT* line and echoes it on the hex display.

Using Your Assembler

On the last section, you started with various short programs in assembly language that you then converted by hand into machine code. But you've already written a program that can do this automatically. The `assembler.c` program you wrote for homework 1 works on the same assembly language used by this CPU.

You should try running the test programs you assembled on homework 1, and then writing some additional programs using the assembly language and running your assembler to translate them into machine code. Once the machine code file has been produced, you can load it into the CPUs RAM in a single step by right-clicking on the user memory component and selecting "Load Image...".

Once you're done, take a step back to marvel at what you've just built and the layers of abstraction that made it possible.

Submission

Submit the files `Control_Unit_FSM.xlsx` and `cpu.circ` by committing them to GitHub and uploading your repository to Gradescope. Only one member of your team needs to do this, though you should remember to attach your partner's name to the Gradescope submission. Note that the autograder will only run tests on your Logisim circuit and your assembler; the Excel spreadsheet will only be examined in case of autograder failures, to make partial credit determinations. Please pay attention to the following specifications to avoid significant autograder failures; in particular:

- Ensure that your files are named exactly as given in this handout, including capitalization.
- Ensure that your circuits in the `cpu.circ` file are named exactly as given. In particular, your submission *must* contain a circuit named `datapath` that constitutes your CPU implementation.
- Ensure that the input and output pins in your `datapath` circuit are named exactly as specified in this handout, including capitalization. There are two input pins, *Input* (8-bit) and *RST* (1-bit), and two output pins, *Out* (8-bit) and *halt* (1-bit).
- Finally, to ensure that the autograder can correctly interface with your CPU, ensure that your input and output pins are oriented correctly. Your *Input* and *RST* pins should both face east (check the "Facing" property in the Attributes pane), and the *Input* pin should appear somewhere above the *RST* pin in your schematic (if you scan your circuit from top-to-bottom, you should encounter the *Input* pin before you encounter the *RST* pin). Similarly, the *halt* and *Out* pins should both face west, with the *halt* pin placed somewhere above the *Out* pin in the layout.