Homework 8

Building the MARIE Architecture: Part 4

Due Date: Before class on Thursday, April 28th.

Setup and Overview

Start with your Logisim circuit from the previous assignment (or you can use my solution). Make a copy and name it `part_4.circ`.

In Part 4, you will build MARIE's control logic. It generates the control signals that drive the datapath you completed in Part 3:

- A 3-bit signal that controls which component is writing to the bus
- A 3-bit signal that controls which component is reading from the bus
- A 2-bit signal that controls ALU function
- A 1-bit signal that controls PC incrementing

The control logic will alternate between generating the micro-ops to fetch an instruction, then generating the ops needed to execute that instruction.

Placing the control logic

Copy the `control.circ` file into the same directory as your `part_4.circ`. From within your solution, select "Project → Load Library → Logisim Library." Select the `control.circ` file. A "control" folder is added to your parts library, with the `control` circuit inside. Note that the `control` circuit is read-only from inside your project, but when you edit it the `control.circ` file, you should see the changes in your project automatically.

The `control` subcircuit has four outputs and three inputs. Add a `control` instance, then connect the `Read from ID`, `Write to ID`, `ALU function`, and `PC+1` outputs to your MARIE CPU. Connect the output of the `IR` to the `instruction` input of `control`.

The other two inputs to `control` are `AC negative` and `AC zero`. Add additional logic to support these inputs (e.g., `AC negative` should be set to *1* if the value in the *AC* register is negative.)

Inside the control logic

The control logic has four main parts:

- A "Fetch Micro-ops" ROM which holds the micro-operations to fetch an instruction. I've already filled it in with the correct data.
- An "Instruction Micro-ops" ROM, which you will fill in to create the CPU microprogram: all the micro-ops for all instructions.
- A "step" counter, which counts which micro-op you're currently executing.
- A "fetch/execute" flip-flop, which is 0 if the CPU is currently fetching an instruction, and 1 if it is currently executing.

The fetch micro-op ROM

The table below shows the micro-ops to fetch an instruction. There are four columns "Read", "Write", "ALU", and "PC+1", which show the control bits needed for each micro-op. There are two additional columns which I'll describe later.

Step	Micro-Op	F/E	If	PC+1	Read	Write	ALU
00	MAR ← PC	0	0	0	010	001	00
01	IR ← M[MAR]	0	0	0	000	111	00
10	PC ← PC + 1	1	0	1	010	010	00

If you reset the simulation, you will see that the `control` is generating the first fetch micro-op: read from the `PC`, write to the `MAR`. Pulsing the clock a few times will execute the rest of the fetch cycle.

If you squint at the table above, it like a memory. If we concatenate all the control bits, we see an array that looks like:

Address	Data
00	000 0100 0100 (hex 044)
01	000 0001 1100 (hex 01C)
10	101 0100 1000 (hex 548)

Inside `control`, there is a ROM with these memory contents. The `step` counter acts as an address register for this ROM, starting at address `0`. The contents of the ROM are split into the individual output signals.

The fetch/execute flip-flop

If you reset the simulation and watch several clock cycles, you can see the `step` counter and the fetch micro-op ROM generating an instruction fetch. On the third op, the bit labeled `F/E` in the table above is high. There are two results:

- The flip-flop labeled `Fetch/Execute` toggles from `0` to `1`. The output of this flip-flop drives a mux; after toggling, the mux now selects micro-ops from the other control ROM, which contains instruction ops.
- The `step` counter is reset to `0`.

The purpose of the `F/E` bit is now clear: set it to `1` on the last micro-op of instruction fetch, and it will begin executing instruction micro-ops. It should also be set to 1 on the last micro-op of every instruction, to start the next instruction fetch.

The instruction micro-op ROM

This ROM has six address bits, rather than the two of the fetch ROM. Take a look at *instruction_ops.pdf* to see why: the address in for this ROM is generated by concatenating the 4-bit opcode with the two-bit `step` counter output.

I filled in a few examples. Take a look at the load instruction, which can be accomplished in two steps (remember, X is just shorthand for IR[11:0]).

- To effect `MAR ← X`, we read from the IR (111) and write to the MAR (001).
- To effect `AC ← M[MAR]`, we read from memory (000) and write to the AC (100). Since this is last last micro-op, the `F/E` bit is set, as well.
- These two micro-ops get filled into memory:

```
000100:  000 1110 0100  (hex 0E4)
000101:  100 0001 0000  (hex 410)
```

- Since we only need a step 00 and step 01, we only fill in locations 000100 and 000101 in memory. We can leave the next two locations uninitialized. Since the `F/E` bit was set in step 01, the CPU will never read these locations.

You should figure out the micro-ops needed for each instruction and write them in the table. Figure out the associated control bits, and write them down as well. Some hints:

- You can do the instructions with fewer micro-ops than the book uses; you never need more than four.
- You can `halt` by simply never fetching another instruction.

You should fill in the instruction micro-op ROM according to your table of micro-operations. Right-click the ROM and click "edit contents..." to enter your data.

Jump instruction

With the setup described above, you can microcode all of the MARIE instrutions except one: the `skipcond` instruction. This is where the `if` bit is used—you should only set it to `1` for the `skipcond` instruction.

This puts an extra condition between the `control` ROMs and output. In pseudocode, it looks like:

```
jumpneg = (IR[11–10] == 00 and AC < 0)
jumpeq  = (IR[11–10] == 01 and AC == 0)
jumppos = (IR[11–10] == 10 and AC > 0)
if (ifbit and (jumpneg or jumpeq or jumppos)):
    pc = pc + 1
else
    do nothing
```

Add additional logic to the `control` circuit to effect this behavior.

Testing

You find two programs in the gdrive, `program1.data` and `program2.data`. You can open them with a text editor to see the machine code (with comments).

Program 1: Double an input

To run the program,

1. Reset your simulation (control/command R).
2. Right-click the RAM, select "load image," and choose `program1.data`.
3. Change the value in the `input` register to any number.
4. Use control/command T to toggle the clock. For the first three clock cycles, you should see the INPUT instruction being fetched. After the fourth clock cycle, you should see the input value copied into the accumulator register.
5. Keep running the simulator. The program should end up finishing with double the input value in the `output` register.

Program 2: Loops

This program multiplies two numbers by using repeated addition. The values in variables *A* and *B* (at locations 0xe and 0xf) are multiplied and the result is written to the output register.

Run the program as above. By default, *A* and *B* are 3 and 4, so you should see 12 (0xc) in the output register when the program halts.

Turning it in

Make a folder labeled "Homework 8" in your turn-in folder and copy in your `part_4.circ` and modified `control.circ`. I will test it by:

1. Resetting the simulation.
2. Right-clicking the RAM and loading a program.
3. Running the program and checking the results.