Carleton University

Department of Systems and Computer Engineering

SYSC 3006 Fall 2016

Computer Organization

Lab #5

Prelab Due: Your Prelab should be done *before* the start of your scheduled lab session. Bring your Prelab with you in hard or soft copy form (e.g. print it or have a word/pdf document ready) for TA inspection during the lab session (no online submission).

In this lab you will:

- Program a given microarchitecture to execute additional instructions.
- Implement, test, and log using Logisim.

Read this document carefully before deciding what to do.

A folder containing 2 Logisim circuits has been included. **You MUST UNZIP the folder** (i.e. extract all files) before trying to load these circuit files into Logisim. If you do not unzip the files, they may load into Logisim, but will not simulate properly!!!

The distributed Lab5 circuit is a working implementation of the microarchitecture discussed in class (wrapping up the TowardsHardware section of the course). In this lab you must program the ROMs to perform some instructions, and demonstrate their execution.

Until this lab, we have assumed that all instructions could be managed by (at most) 3 generic execution states (E0, E1, and E2). The Decode ROM introduced in Lab 4 provided additional flexibility by allowing the NOP instruction to be implemented with no execution states.

Any **new** instruction can be added to the microarchitecture quite easily using the following steps:

1. Design the execution states (E0, E1, etc.) of the instruction. This requires designing how the instruction will be executed as a sequence of steps utilizing the components in the datapath (i.e. the registers and ALU). Recall: we already did this in class for the initial set of operations (NOP, ADD, XOR, etc.). Your design must be concerned with issues such as: how the required behaviour can be realized, what operations (if any) the ALU should preform, and how/when the temporary registers (T1 and T2) should be used to avoid conflicts in the use of the Internal Data Bus.

2. Realize the design as a sequence of Control FSM states using the provided Instruction Execution States Table. The FSM outputs in each state will realize a step in your designed execution of the instruction. In order to fill in the Next State fields of the Table, you must decide which Control FSM Output ROM words will be used to implement each state. Select some unused words in the Control FSM Output ROM to be used to hold the encoding of the execution states. "Unused words" are words that do not currently hold the encodings of any FSM states. The appropriate addresses of the selected words should be entered in the appropriate "Next State" fields of the Table. Program the resulting states in the table into the corresponding Control FSM Output ROM words.

3. All instructions use the same fetch and decode states, but the first execution state for each instruction is selected by the Decode ROM. Using the new instruction's opcode as the Decode ROM address, program the Decode ROM to point to the instruction's first execution state (i.e. the one that you programmed in Step 2).

The goal of this Lab is to implement the instructions shown in Table 1: (Note: the instruction associated with the RY operation is named "MOV" (MOVE).)

Instruction	Effect	OpCode (hex)
NOP	Do nothing	00
ADD	Rd ← Rsx + Rsy	01
SUB	Rd ← Rsx – Rsy	02
MOV	Rd ← Rsy	03
AND	Rd ← Rsx AND Rsy	04
OR	Rd ← Rsx OR Rsy	05
XOR	Rd ← Rsx XOR Rsy	06
NOT	Rd ← invert(Rsy)	07
NEG	Rd ← –(Rd)	17

Table 1: Lab 5 instructions

Part 1

Program the microarchitecture for the ADD through NOT instructions in Table 1 above (opcodes 0x01 through 0x07). If you followed the directions in Lab 4, the solution to this part should be identical to your solution in Lab 4 Part 2 (i.e. the supporting design tables are the same). If you did not get Lab 4 Part 2 working, this is a good opportunity to complete that part of Lab 4.

Part 2

Recall that the Lab 4 version of the NOP instruction did not have any execution states. Extend your solution to Part 1 (following the process outlined above) to include a new implementation of the NOP instruction. This version of the NOP instruction should have 2 execution states, and these execution states must not change the internal state of any components on the datapath. This new implementation intentionally "burns" (i.e. wastes) 2 execution states. (Can you think of cases where idling the processor in this way might be useful?)

Part 3

Extend your solution to Part 2 to add the "Negate" (NEG) instruction shown in Table 1. This instruction has only one operand: the destination register. NEG calculates the 2's complement of the initial value in the destination register and loads the result back into the destination register. The instruction might be written as:

```
      NEG          ; R6 is the destination register
   R6 ← – R6       ; set the contents of R6 = 2's complement of the initial value of R6
                   ; this instruction would be encoded as: 0x 1760 0000
```

Hint: The opcode of the NEG instruction (0x17) has been carefully chosen to support the instruction's implementation. Think about the ALU operation represented in the least significant nybble of the opcode … could this be used to help achieve the "negate" functionality?

Prelab: Support Tables are provided (docx file).

Complete the provided Control FSM Output Table for each Part of the lab. The TAs will assume that the Table for Part i + 1 will extend the Table for Part i (so there is no need to duplicate the Table from Part i in the Table for Part i + 1).

Complete the provided FSM Decode ROM Table to show any entries that must be programmed.

Complete the provided Main Memory Table to contain the encodings of the Test Program instructions as indicated, and the expected result of each operation (Results column) .

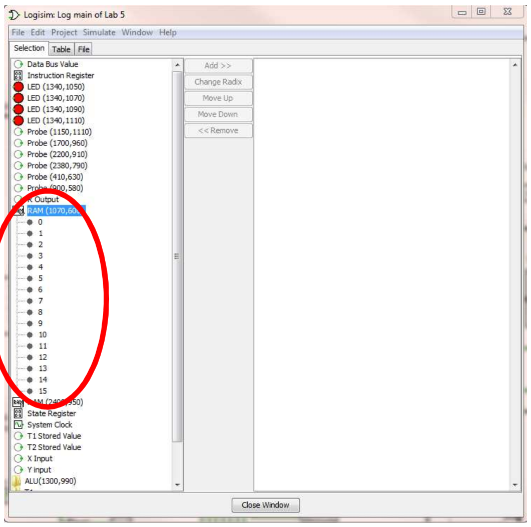
In the Lab: Implement your Control FSM design and execute the Test Program using the following steps:

1. Program the words of the Control FSM Output ROM, Decode ROM, and Main Memory with the values from your completed prelab tables. Be sure to include the Main Memory contents exactly as given in the table (i.e. be sure to include the Illegal instruction!).
2. Make sure the System clock is high (in the 1 state).
3. Make sure all components in the datapath hold the value 0 (use the "Clear Internal State" button in the upper left corner of the circuit).
4. Cycle the System Clock through the execution of your Test program.

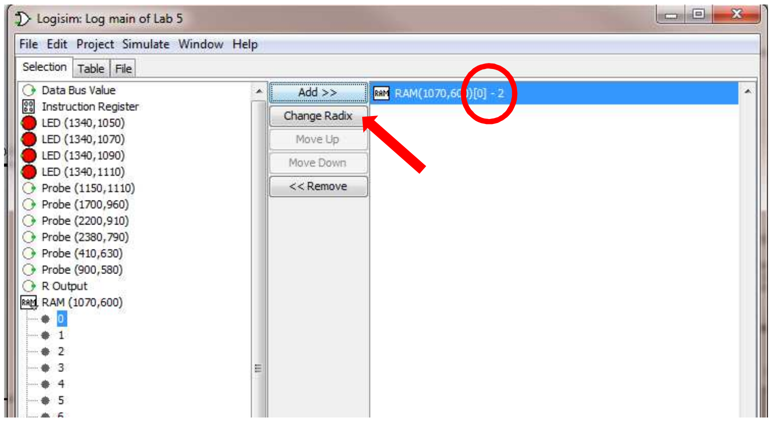
[In-lab: record a Simulation Log that validates the execution of your test program (Main Memory Table). The log should include hexadecimal values for: the PC register, the Instruction Register (IR), and any registers (R0 – R15) modified during the execution of the Test program. See the note below about logging registers and hexadecimal values. Ensure that your log correctly matches the results from your prelab table (Results column under Main Memory Table);

Good Luck! ☺

A note on logging registers and hexadecimal values: Individual register values can be accessed for logging by expanding the appropriate RAM component in the Simulate → Logging → Selection tab. Expanding the component (by double-clicking on the component) should look like:



The individual registers can then be added to the log. For example, adding register 0 would look like this:



Note the (highlight circled) "[0] - 2" in the added register's entry. The "[0]" indicates word 0 (i.e. R0), and the "2" indicates the radix (base) for displaying the value. To change the radix for displaying the value, press the "Change Radix" button (see highlight arrow in figure). Decimal = radix 10. Hexadecimal = radix 16.

The same procedure can be followed to change the display radix for the IR, too.