Lab 10: Datapath and Control

CS/CoE 0447(B) Fall 2015 — Dr Kosiyatrakul (Tan)

Released: Wednesday, November 18, 2015
Due: Tuesday, November 24, 2015

Submission timestamps will be checked and enforced strictly by CourseWeb; **late submissions will not be accepted.** Remember that, per the course syllabus, if you are not marked by your recitation instructor as having attended a recitation, your score will be cut in half. **Follow all instructions.**

In this lab, we will build a datapath and control for a simplistic processor. Since this can be a complex task for a MIPS architecture, we will build for a toy architecture — MiniMIPS.

MiniMIPS has only a handful of instructions, two registers (*A* and *B*), no memory instructions like `lw` and `sw`, and no control transfer instructions like `beq` or `j`. By following this guide, you'll build MiniMIPS. You can take the same concepts that you learn and apply them to your MIPS processor.

A. The MiniMIPS instruction set

The instruction set is very simple – there are only five opcodes. There are only two registers, *A* and *B*. The registers are 8 bits. Every instruction has a destination register (either *A* or *B*). The source registers are always *A* and *B*, so there is no need to specify them. The instructions are:

Opcode	Assembly	Definition
0000	`set` *$r, imm8*	*$r* ← *imm8*
0010	`add` *$r*	*$r* ← *A* + *B*
0011	`sub` *$r*	*$r* ← *A* – *B*
0100	`or` *$r*	*$r* ← *A* | *B*
0101	`and` *$r*	*$r* ← *A* & *B*

In these instructions, *$r* refers to either register *A* or *B*. Register *A* is *$r*=0, and register *B* is *$r*=1.

Unlike MIPS, which has R, I, and J format instructions, MiniMIPS has only one instruction format, a 16-bit instruction containing a 4-bit opcode, a 1-bit destination register, and an 8-bit immediate:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode				*$r*	*Unused*			Immediate							

Instructions are addressed by these 16-bit words, and their addresses are 8 bits; therefore, a program can contain up to 256 of these 16-bit instructions.

B. Components and design

To implement MiniMIPS, requires only a few components, many of which are available in Logisim's library. The components include the following:

- **Registers** – hold *A* and *B*, 8 bits each
- **ROM** – 8-bit address, 16-bit data width (instruction)
- **Program Counter** – 8-bit register
- **Fetch adder** – 8-bit adder incremented by 1 on each cycle
- **ALU** – four functions: add, sub, AND, OR
- **Control unit** – decodes opcode for MUXes and ALU
- **MUX** – select between *imm8* for set and output of ALU
- **Fields** – extract opcode, register and immediate from instruction
- **Decoder** – generates control signals from opcode

Start by opening Logisim to a blank circuit. In the following steps, we will add or create the necessary components:

1. **First, add a memory to hold instructions.** Use a "ROM" from the Memory library. By default, Logisim creates a ROM with an 8-bit address and an 8-bit data width. The ROM will hold the program, composed of 16-bit MiniMIPS instructions. Thus, we need to change the data width to 16 bits. Click on the ROM in the main circuit grid. This will open a window pane on the left with the ROM's attributes. One of the attributes is "Data Bit Width". Configure this to 16 bits by clicking on it, and scrolling down to 16.

2. **Memory typically has a select signal that controls whether the memory will do any operation.** The ROM has a select signal labeled *Sel*. We need to set this to logic-1 using a constant from the Wiring library. Draw a line, using the pointer, between the 1 and the *Sel* of the ROM.

3. **We need a program counter.** The PC is just a register. Select the Memory library components, and then grab a Register. Add it to the circuit. By default, the Register is 8 bits, which matches the address size for MiniMIPS (8 bit address for instructions). Connect the *Q* output of the register to the *A* input of the ROM. This connects the PC to the address input of the ROM.

4. **Each clock cycle, we need to increment the PC by 1 instruction.** Thus, we need to add an adder to the circuit and wire it to the PC. Use the built-in adder from the Arithmetic library. By default, the adder is 8 bits. Connect the output of the Adder to the *D* input of the PC. We need to add a Constant 1 for addition. Then connect the *Q* output of the PC to the first input of the Adder. Lastly, the PC register has an *En*, or "enable", signal. When set to 1, the register can be written on a clock cycle. Since we update the PC every cycle, connect the enable to the 1-bit constant 1 you connected to the ROM's *Sel*. The Fetch portion of the data path should now be completed.

5. **We need two registers, *A* and *B*.** Add two registers to the circuit diagram. We will connect these registers to the output of the ROM, so you will probably want to put the registers to the right of the ROM in the diagram.

6. **Now, build a simple four-function ALU.** We will do this as a subcircuit called "ALU". Include 8-bit AND, OR, adder and subtractor components. Go ahead and try this! (**Hint:** AND and OR gates can be configured to have two 8-bit inputs.) Use a 4:1 MUX to select one of the four ALU functions. Connect input 0 to the output of the adder, input 1 to the subtractor, input 2 to the AND, and input 3 to the OR. Add a 2-bit input pin, labeled "Operation". Finally, add two 8-bit input pins and an 8-bit output pin.

7. **Add the ALU subcircuit to the main diagram.** For our simple instruction set, the ALU's inputs come only from the *A* and *B* registers. So, connect the *Q* signal of registers *A* and *B* directly to the *A* and *B* inputs of the ALU. We also need to connect the ALU's output to the registers. Notice that the registers can be written from either the immediate of the `set` instruction or the result of the ALU instructions. So, we need a 2:1 MUX between the output of the ALU and the inputs of registers *A* and *B*; call it "SetMux". Connect input 0 of SetMux to the output of the ALU. For now, leave input 1 unconnected. Next, connect the MUX's output to the *D* input of *A* and *B*.

8. **Add a subcircuit called "Fields"** to get access to individual fields in an instruction. The Fields subcircuit will contain a splitter which separates its 16-bit input (one instruction) into three outputs: a 4-bit opcode, a 1-bit register, and an 8-bit immediate. Be sure to label your outputs appropriately!

9. **Now, we need to build the control unit, a.k.a, Decoder.** The decoder is nothing more than a combinational circuit. It takes as an input the opcode from the instruction to output the control signals. The control signals we need are the operation signals for the ALU and the select signals for SetMux. You can draw a truth table and then minimize the table with pencil and paper. When you are done with the Decoder, it should appear similar to Figure 1, below:

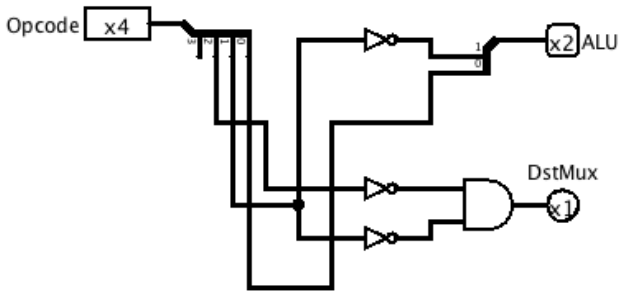

Figure 1. *A simple control circuit for a MiniMIPS processor.*

10. **We're now ready to finish wiring everything on the main circuit.** Make sure you have the following components on the main circuit: ROM, Fields, Decoder, ALU, *A* and *B* registers, PC register, an adder for PC, and a MUX SetMux. **C**onnect the Immediate output of Fields to the second input of SetMux. This is the second possible location for a value to write to the registers. Connect the Opcode output of Fields to the decoder. Connect the Operation output of the Decoder to the ALU. Connect the Select output of the Decoder to the SetMux. At this point, everything should be connected, except the clock and the *Enable* signals for registers *A* and *B*.

11. **Notice that every instruction writes a register.** So, we need to set *Enable* for either *A* or *B* on every clock cycle. When the register field is 0, we should enable the *A* register and disable the *B* register. When the register field is 1, we should enable the *B* register and disable the *A* register. Thus, we have two Boolean functions: $A \leftarrow \neg\$r$ and $B \leftarrow \$r$, assuming *$r* is the register from the instruction (i.e., the Register output of Fields). Add the necessary AND and NOT gates to your circuit and wire them.

12. **Lastly, add a clock.** Connect the clock to all registers: PC, *A* and *B*. Assuming you've done everything correctly, you should be ready to test the processor!

13. **Test the processor by poking at the ROM.** Look at the attributes and select "(click to edit)" for the Data Contents. Let's try the instruction "`set A,1`". This instruction has the value 0x0001. Put the value 0x0001 in the first location of the ROM (address 00, upper left corner). Try putting the value 0x0802 in the second location of the ROM; this corresponds to "`set B,2`". Close the data window and reset the simulation with Simulate > Reset. Now, simulate the circuit by ticking the clock. You should see the values of the *A* and *B* registers change to 1 and 2, respectively.

14. **Congratulations! You've got a working processor!** Write a few more instructions and add them to the ROM to test the functionality of your add, subtract, OR, and AND instructions as well.

▶ **Save your completed Logisim file as `lab10.circ` and submit it via CourseWeb.**

Notes

If you feel confident, you can build the circuit directly without following the steps above. Here are some notes that may help.

The *Enable* control signals for *A* and *B* can be tied directly to the *$r* bit in the instruction. The *Enable* control signals are set as $A \leftarrow \neg\$r$ and $B \leftarrow \$r$. Either *A* or *B* are enabled on every clock cycle since each instruction has a destination register.

Fields can be implemented simply as a wire splitter. Some like to put it inside a subcircuit to hide the splitter, but that is not really necessary for this simple case.

The ROM is configured with 8-bit address and 16-bit data width for the 16-bit instructions. The Select signal should be set.

ALU is implemented as components from Logisim with a 4:1 MUX. Some like to hide the ALU in subcircuit. For my example, we used 00 for add, 01 for sub, 10 for AND, and 11 for OR.

There is a MUX called SetMux on the input to the registers. This MUX will need to select either the imm8 field (for `set`), or the ALU output for all other instructions.