CS447 Lab #11 (Week 12)

Introduction

For this lab, you will build a circuit that runs a series of MIPS instructions. You will read hexadecimal-formatted *addi* instructions from instruction memory, execute the requested addition, and then update the appropriate register to hold the computed result.

Note: This Lab must be submitted online by 11:59 pm on Tuesday, November 22nd. Please see the instructions on the final page.

Objective: Execute a series of MIPS instructions in Logisim

Program to implement

We will use Logisim to implement a MIPS program that only uses four registers and **addi** instructions:

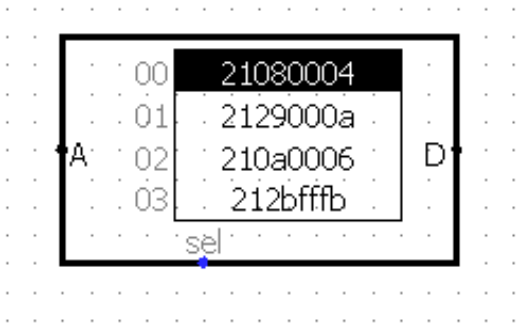
```
.text
addi $t0, $t0, 4
addi $t1, $t1, 10
addi $t2, $t0, 6
addi $t3, $t1, -5
```

The program ends with the registers in the following state:

$t0	8	0x00000004
$t1	9	0x0000000a
$t2	10	0x0000000a
$t3	11	0x00000005

This program has been converted into hexadecimal format and saved in the file **lab11code.txt**. This file is available from **Courseweb**, under **Course Documents/Week 12 Lab**. It is also available on Github. The first line of the file is a format specifier required by Logisim, and the remaining lines are the hexadecimal representations of the above code.

As in Project 2, you can store this code in a **ROM** component. In a new Logisim file, add a ROM component to the main circuit (see the Memory folder). Change the ROM's **Bit Width** to match the bit width of the MIPS instructions (32). Subsequently, load the instructions by right-clicking on the ROM component, choosing **Load Image**, and selecting the **lab11code.txt** file. You should then be able to see the hexadecimal instructions within the ROM component:



Note: If you are having trouble getting the correct values to show up in the ROM, make sure that you have set the correct bit width before attempting to load the instructions. The Address Bit Width can remain at 8. The "sel" input site does not need to be used for this lab.

Set up the Program Counter to fetch instructions

As in MIPS, we will store the current value of the program counter in a $PC register. **Create this register** in your main file, and adjust the **Data Bits** so that it matches the **Address Bit Width** of your ROM.

Remember that, in the processor's Instruction Fetch stage, an instruction is read from memory at the address specified by $PC. This implies that you should **send the current contents of the $PC register** to the **A** input site on the ROM (which allows you to specify the address of the contents you would like to read).

Finally, the **$PC** will need to be **updated on each cycle** so that the subsequent instruction can be fetched. For this simulation, the $PC will only need to be updated by 1 on each cycle, because our addresses correspond to 32-bit words, rather than individual bytes. To implement the $PC updating:

1. Attach a clock to your $PC register.
2. Create a loop, which takes the current contents of $PC, adds a constant of 1 to the current value, and then updates $PC to the new, incremented value.
   ➔ You may use the built-in Adder for this, and a Constant from the Wiring library.

You can test the $PC by selecting **Ticks Enabled**. You should see a new instruction highlighted with each cycle. You can use **Reset Simulation**, in the Simulation menu, to revert to your circuit's initial state.

Set up the register file

Remember that we'll be running code that uses four registers:

```
.text
addi $t0, $t0, 4
addi $t1, $t1, 10
addi $t2, $t0, 6
addi $t3, $t1, -5
```

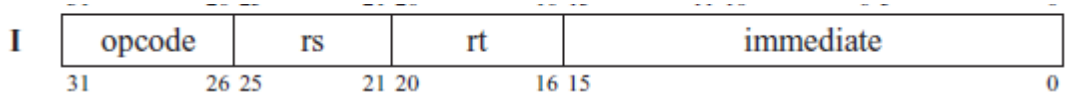
Create a new canvas, and within it, create a new **4-register register file**. For this purpose, you may find it helpful to use your work for **Lab #9.** Note that it is possible to copy and paste circuits between files (on Windows, use Ctrl-A to highlight all components on a canvas). From that point, you will need to change the registers to **accommodate the 32-bit width** of typical MIPS registers. You will also need to update any components that send inputs to, or receive outputs from those registers.

Note that, since we are using MIPS instructions, registers $t0-$t3 will be indexed with values 8-11 in the hexadecimal code, which introduces a discrepancy with our 0-3 numbering in the register file from lab #9. A later section of this handout will describe how to account for this mismatch in the main circuit, although you are also welcome to try to handle this within the register file itself.

Fetch and decode each instruction

Now that you can read the instructions, you will have to decipher what each *addi* instruction is asking the processor to do. This corresponds to the Instruction Decode stage of the MIPS processor.

Recall that **addi** is an **I-type** instruction, which is formatted as follows (image credit: MIPS green sheet, from Patterson and Hennessy text):



Since all of the instructions use the same operation, we will not need to use the opcode. We will need to use the other elements, as specified for addi:

Add Immediate	`addi`	I	R[rt] = R[rs] + SignExtImm

In other words:

- **rt = index of destination register,**
- **rs = index of source register, and**
- **immediate = 16-bit immediate, to be sign-extended into a 32-bit immediate**

To decode these necessary instruction elements, you may wish to proceed as follows:

1. Create a new "decode" circuit, and start with an input pin to hold the current instruction.
2. Use a **splitter** to divide the instruction into its fields. The I-type format above provides you all the information you need to determine the Fan Out of the splitter, and how the input bits should be divided amongst the output branches.
    - Note: You can either decode or ignore the opcode field. Note that the Splitter gives you the option of sending individual bits to none of the branches (choose "none").
3. Send the outputs of the splitter to output pins, with appropriate labels for the bit fields.

4. Return to the **main** circuit, and create an instance of your new decoder circuit. It will need an instruction input, which you can retrieve from the **D** output pin of the ROM.

Get the source operand and sign-extended immediate values

Your decode component should send three outputs to main: **rt** (the destination register), **rs** (the source register), and the 16-bit immediate. The immediate value should be sign-extended to **32-bits**. You may use the built-in sign-extender for this.

You will need to use **rs** to retrieve the contents of the appropriate register from the register file. Here we will need to adjust for the different register numbering used by MIPS and our simple Logisim circuit.

- In our hex instructions, registers $t0-$t3 will be indexed with the values 8-11. In our register file, our registers are indexed with the values of 0-3 (the four selection settings recognized by the decoder).
- To account for this, you can send the **rs** value from the instruction to a subtractor, and subtract out a constant, 5-bit-wide value of 8.
- Additionally, you will need to reduce the bit width of **rs** from 5 to 2. You can do this by using the Bit Extender (which can be used to narrow bit representations).

Once you have converted **rs** to a 2-bit value between 0-3, you can send that converted value to a new instance of your register file, at the input pin that determines which register is read. Remember that your register file will need a clock input.

Perform the addition and write the results.

Add the value that you read from the register file to the sign-extended immediate. Send the results to your register file. To select the destination register, you will need to send the information provided by **rt.** As with **rs**, you will first need to **subtract 8** from the current value of **rt**, **reduce the bit width to 2**, and then send this converted value to the register file.

Execute the program

Enable **Ticks Enabled,** and let the program run through the four instructions. After these instructions have run, you should see the appropriate values in your registers:

$t0	8	0x00000004
$t1	9	0x0000000a
$t2	10	0x0000000a
$t3	11	0x00000005

Tips

1. Circuits of this nature are especially prone to creating temporary errors that can be fixed by restarting the program. A restart may be worth a try if you notice odd or unpredictable behavior. The circuit should work eventually.

2. Similar advice holds if the values are never updated in your registers. To determine if your problem is due to some Logisim bug:
    a. First try viewing the register file by using the poke tool, and not by selecting the circuit in the Explorer menu. Often this makes the correct values visible.
    b. Otherwise, try a restart, or add output pins to your register file that show the current contents of $t0-$t3. Read out these values with Probes (see Wiring) in the main circuit. If the probe values are correct, but the register values are all 0s, then the problem may be a Logisim issue. If this issue is persistent, then the correct probe values will be sufficient to receive credit for the assignment.

Submission:

If you finish during recitation, please notify the TA, so that your credit for this assignment can be confirmed right away.

Regardless of whether you finish during recitation or not, you must submit your work online. Save the file as **YourPittUserName_lab11.circ** and submit the file via the appropriate link in Courseweb (see Course Documents/Week 12 Lab). The lab must be submitted by **Tuesday, November 22nd**, by **11:59 pm**.