Name:
CSE 20221 Logic Design
HW07: Programmable Calculator

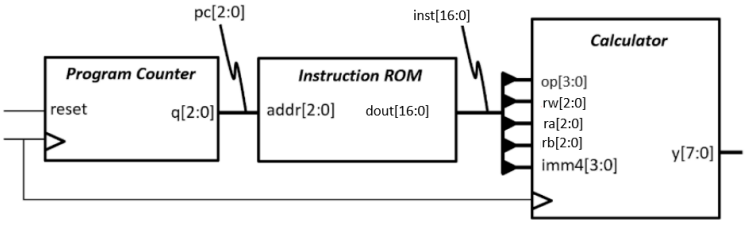
The purpose of this assignment is to design a programmable calculator that demonstrates many of the features, in simplified form, of the albaCore instruction set architecture (ISA). As a smaller and less powerful cousin of albaCore, we will call it *Guppy*. The Guppy ISA supports the albaCore arithmetic/logic and load-immediate instructions, but not loads, stores, branches, or jumps. With a 4-bit opcode, leaving out these instructions affords us "room" for additional operations, so we will also include xor (^) and a "signed ldi" instruction, and an additional instruction of your choice. Unlike albaCore, it has only 8 registers, and they are each only 8 bits wide. It has a small read-only memory for programs up to 8 instructions long. Finally, for simplicity, Guppy uses a common instruction encoding for all instructions rather than repurposing fields based on instruction type like albaCore does, so Guppy instructions are 17 bits wide while albaCore instructions are only 16 bits wide. The Guppy instruction set is as follows:

Instruction Name	operation	17-bit encoding				
		16:13	12:10	9:7	6:4	3:0
`add`	`rw ← ra + rb`	`0000`	`rw`	`ra`	`rb`	`0`
`sub`	`rw ← ra - rb`	`0001`	`rw`	`ra`	`rb`	`0`
`and`	`rw ← ra & rb`	`0010`	`rw`	`ra`	`rb`	`0`
`or`	`rw ← ra | rb`	`0011`	`rw`	`ra`	`rb`	`0`
`not`	`rw ← ~ra`	`0100`	`rw`	`ra`	`0`	`0`
`shl`	`rw ← ra << rb`	`0101`	`rw`	`ra`	`rb`	`0`
`shr`	`rw ← ra >> rb`	`0110`	`rw`	`ra`	`rb`	`0`
`ldiu` (unsigned ldi, like albaCore)	`rw ← {4'b0, imm4}`	`0111`	`rw`	`0`	`0`	`imm4`
`xor`	`rw ← ra ^ rb`	`1000`	`rw`	`ra`	`rb`	`0`
`ldis` (signed ldi)	`rw ← SignEx(imm4)`	`1001`	`rw`	`0`	`0`	`imm4`
`multiply`	`rw ← ra * rb` (your choice)	`1010`	`rw`	`ra`	`rb`	`0`

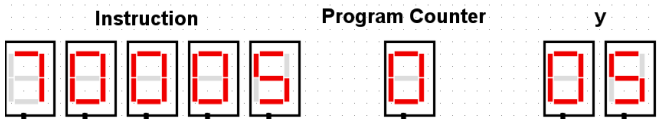
Note: the `ldis` instruction allows us to initialize a register to a negative number, e.g., `ldi r10, -2` sets r10 to 0xFFFFFFF0. The meaning of `SignEx(imm4)` is to *sign-extend* the imm4 by taking the most-significant bit (bit 3) of the 4-bit immediate and copying it into each of the upper (leftmost) 4 bits of the destination register. The lower (rightmost) 4 bits in the destination register are a copy of the imm4 itself. There is a Logisim-evolution built-in component to do this sign extension, which you are encouraged to use.

The "`prog_calculator`" circuit below implements the Guppy ISA. `prog_calculator` consists of the calculator, a ROM containing a series of instructions, and a program counter that generates the sequence of ROM addresses. The calculator inputs come from fields of the 17-bit instruction, and the output of the calculator, `y`, is the result of the current instruction. The program counter (`pc`), instruction (`inst`), and result (`y`) are shown on hex digit displays (shown below).

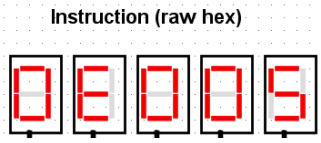
`prog_calculator`:



Example hex digit displays for instruction `ldiu r0, 5` which is at address 0 (`pc`):



Since the instruction encoding does not nicely align to multiples of 4, additional hex displays should show the *raw hex* of the encoded instruction. For example, the instruction `ldiu r0, 5` is encoded as `0x0E005`:



Overview

For this assignment and others remaining in the course, the circuit complexity will increase, both in functionality and in how many subcircuits comprise the entire circuit. Going forward, some circuits will be provided, based on in-class examples, from previous Homeworks, Logisim-evolution built-ins, etc. Here is a list of each circuit you will work with in this assignment, its description, and its state of completion.

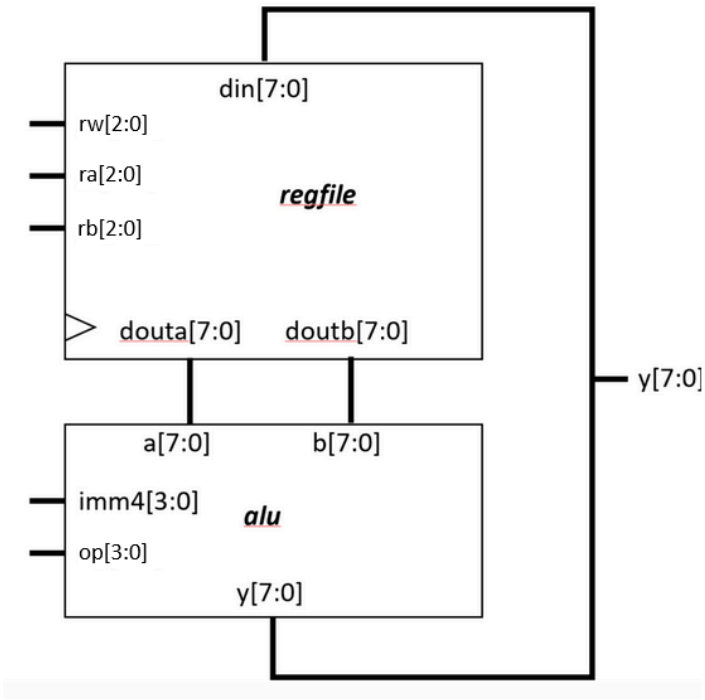
Every circuit should be implemented as a sub-circuit in your HW07 .circ file using the bottom-up approach, like in HW05 and HW06.

Circuit	Description	State of Completion
`alu`	The Arithmetic and Logic Unit (ALU)	Based on HW06, needs to be expanded
`calculator`	Calculator circuit, includes instances of register file and ALU circuits	Create from scratch

`inst_rom`	Instruction ROM, 8 words deep by 17 bits wide	Create from scratch
`prog_calculator`	Programmable calculator, includes instances of calculator, program counter, and instruction ROM circuits	Create from scratch
`prog_counter`	Program counter, a 3-bit (incrementing) saturating counter	Using the Logisim-evolution built-in counter.
`regfile`	Register file with 2 read ports, 1 write port	Based on register file example from class (regfile_1w2r); needs to be expanded/modified

Design and Test of the Calculator Circuit

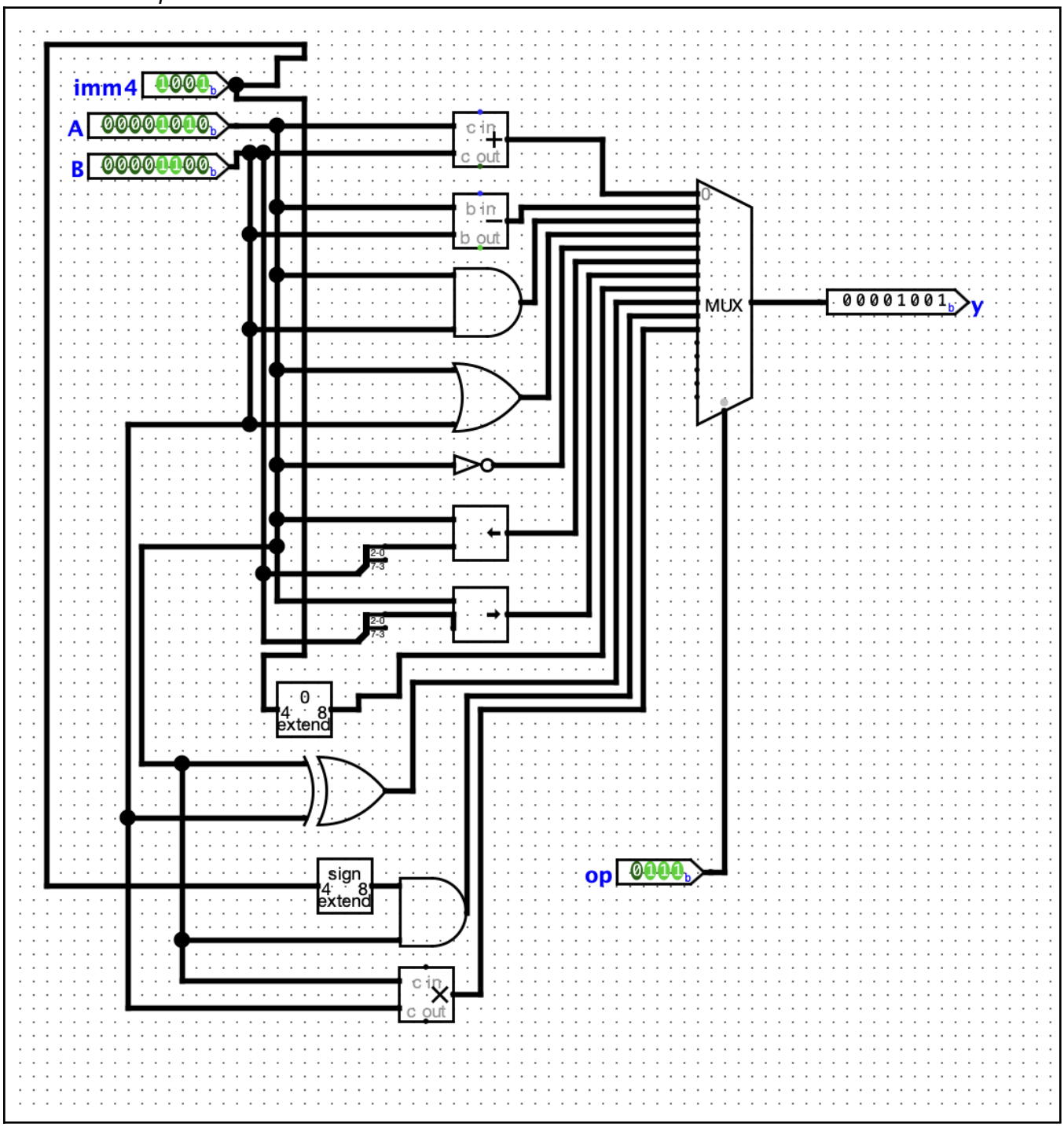
The `calculator` circuit consists of a register file (`regfile`) connected to an ALU (`alu`). The `regfile` and `alu` circuits are based on previous work, but each needs to be modified/expanded and then tested together. Once you have validated the ALU and register file, you will create and test the `prog_calculator` circuit.



Expand the `alu` circuit

The ALU from HW06 has nearly all the functionality needed for this assignment. However, `a`, `b` and `y` need to be expanded to 8 bits, and the behavior for `alu_op` = 7 is slightly different – the `b` passthrough now implements the ldiu instruction, i.e., the `rw` ← `{4'b0, imm4}` register transfer operation. You also need to add `xor` and the `ldis` functionality, and an 11th operation of your choosing.

schematic for updated alu circuit



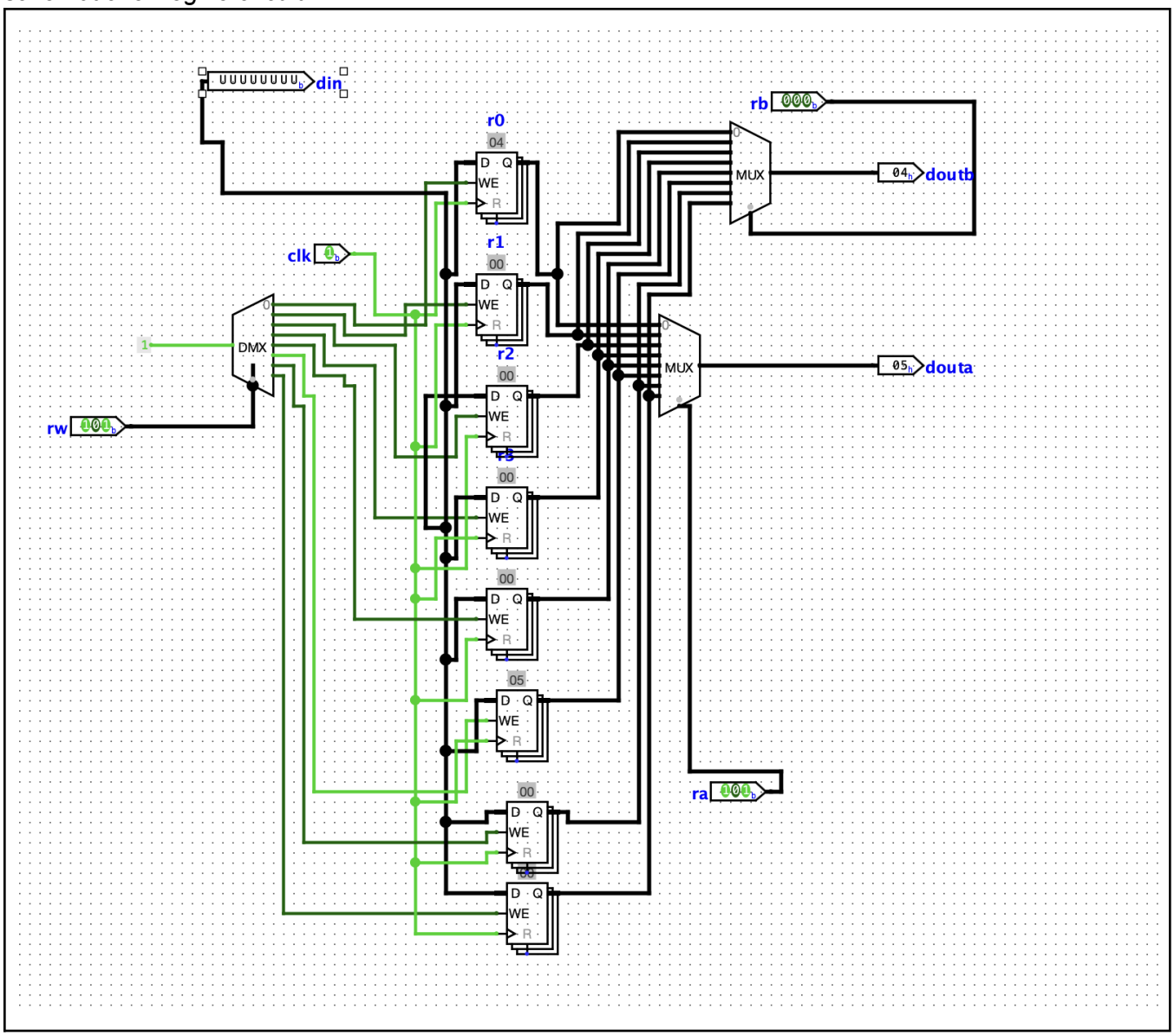
You do **not** need to include a test vector for the ALU, but you are strongly encouraged to make sure the new functionality works correctly before proceeding.

Expand the `regfile` circuit

The register file circuit `regfile` has 2 read ports (`ra/douta, rb/doutb`) and 1 write port (`rw/din`), as in the example we saw in class. Because the register file is written every clock cycle in Guppy, it does not need to have a write enable port (pin) – ensure you are still able to select a single register to write (by setting its WE = 1). You will need to expand the multiplexors to support 8 registers that each

store an 8-bit value. Recall the warning from class that expanding a component like a mux or demux "in place" may lead to bad wire connections. Instead, cut/paste the component off to the side, expand it, and then reorganize as needed before you drag it back into place.

schematic for regfile circuit

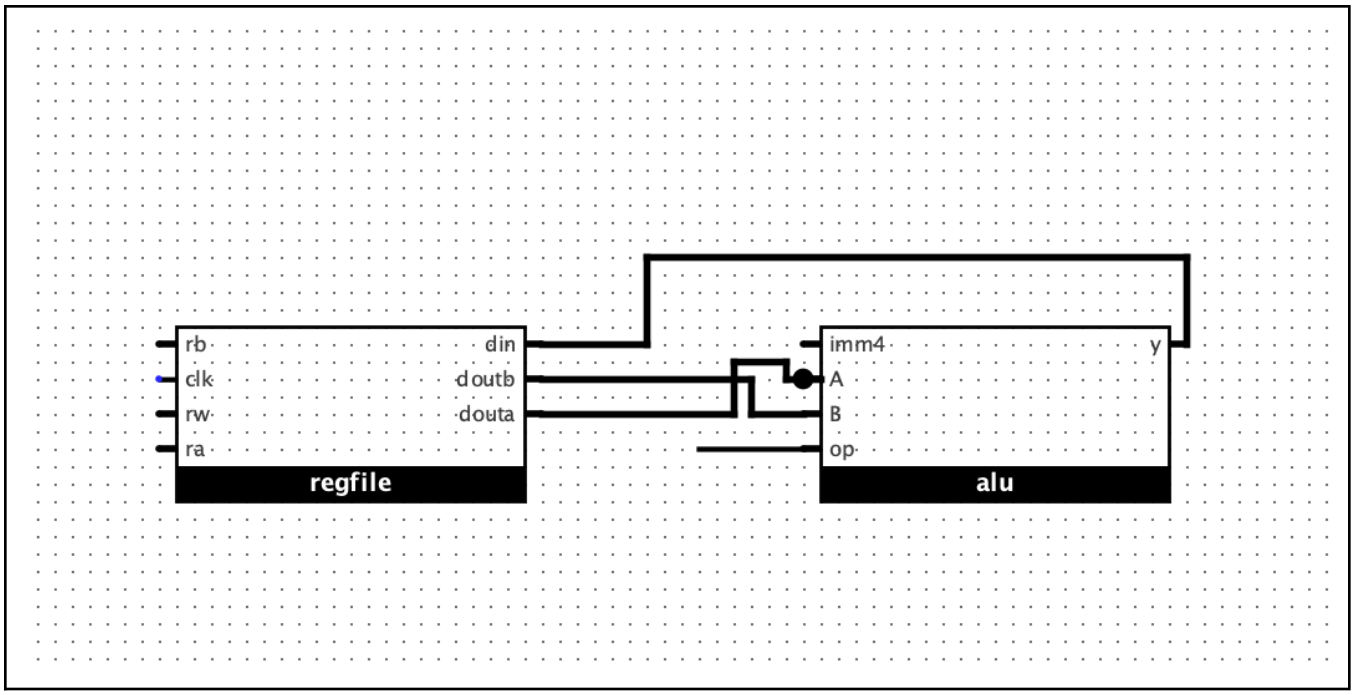


You do **not** need to include a timing diagram test for the register file, but you are strongly encouraged to make sure the new functionality works correctly before proceeding.

Completing the `calculator` circuit

Create the `calculator` schematic and add wires as needed to connect the `alu` and `regfile` as described above. Note the similarities to HW06, where we had a single register (the accumulator) to store the result of ALU computation.

schematic for `calculator` circuit



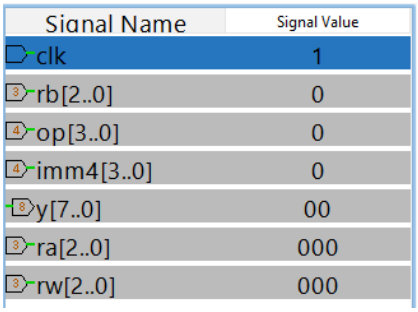
In preparation for testing the `calculator` circuit, complete the following table specifying the values (in hex) for each of the `calculator` inputs needed to perform the specified register transfer operation, along with the expected value of the output `y` resulting from that operation (that would be written to writeback register `rw` on the next rising clock edge). As a guide, the 5th instruction's output is shown. The 6th and 7th instructions (ROM Addresses 101, 110) in the table are for you to test op = 1010, i.e., the one you chose. If you need an additional instruction to help demonstrate your operation's functionality, place that in ROM Address 101 (with r3 as rw) and your custom operation in ROM Address 110 (with r4 as rw); otherwise, place your custom operation in 101 (with r4 as rw) and do not use 110. **In any case, do not use ROM Address 111, it should be all 0's**.

Table with tests for `calculator` circuit

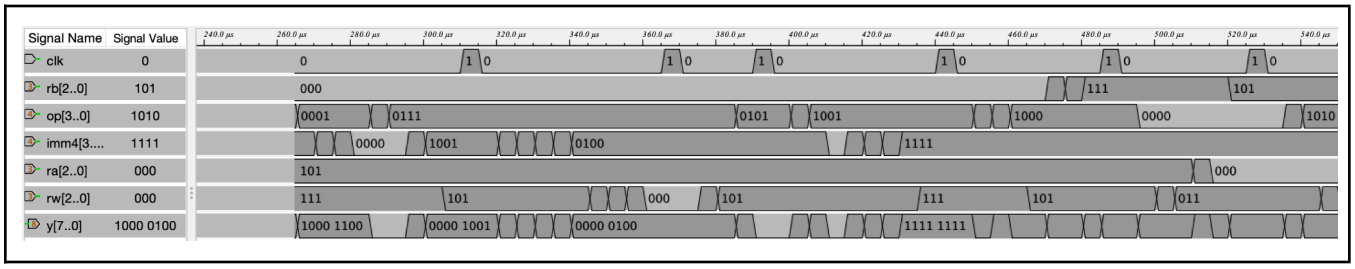
ROM Address	Operation	Input					Output
		op	rw	ra	rb	imm4	y
000	ldiu r5, 9	0111	101	000	000	1001	0x09
001	ldiu r0, 4	0111	000	000	000	0100	0x04
010	shl r5, r5, r0	0101	101	101	000	0000	0x90
011	ldis r7, -1	1001	111	000	000	1111	0xFF
100	xor r5, r5, r7	1000	101	101	111	0000	0x6F
101	add r3, r0, r5	0000	011	000	101	0000	0x73

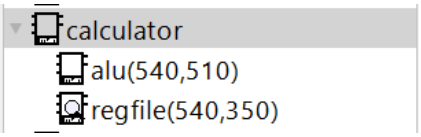
110	multiply r0,r0, r5	1010	000	000	101	0000	0xCC
111	<do not use>	0x0	0x0	0x0	0x0	0x0	0x00

Test your circuit and generate a timing diagram using Logisim-evolution to show the register transfer operations in the table work correctly. All inputs and outputs should be included and their names clearly visible, as shown below.



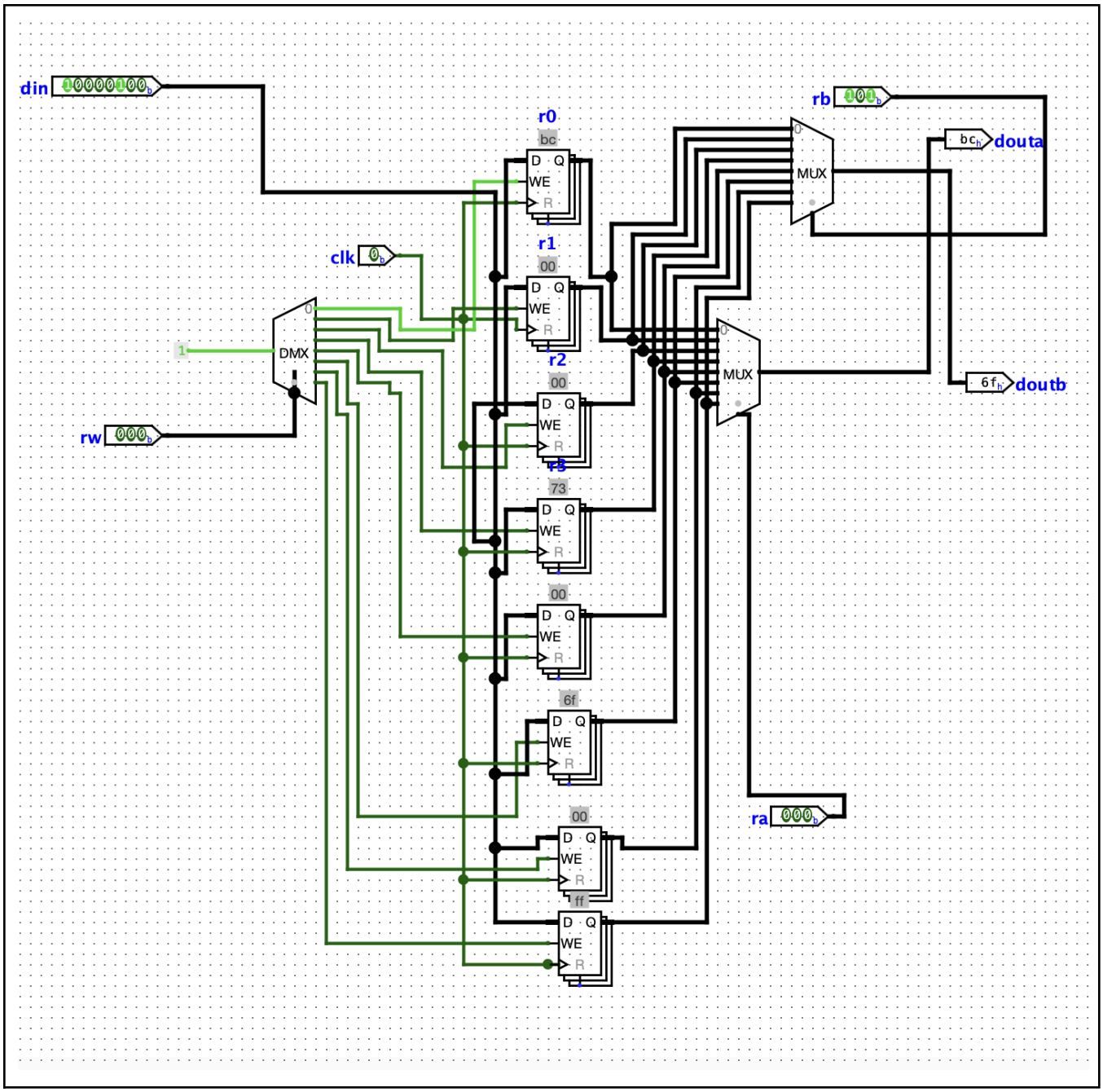
Screenshot of timing diagram for `calculator`



After generating your timing diagram, ensure the simulation is paused and inspect the register file contents to ensure all registers have the expected values. To do this, descend into the instantiated `regfile` circuit by clicking the  on `regfile` in `calculator`. To ascend back, you can double-click the parent in the Simulate tab, i.e., calculator in this case: 

Screenshot[1] of `regfile` circuit showing the 8 registers (so we can see final state)
You can take individual screenshots of each register and paste here

[1] You may note that adding the registers' state to the timing diagram can be more efficient (as in the last part of this assignment). You are asked to use this method just once so you get practice inspecting internal circuit state – this is a useful tool for debugging.



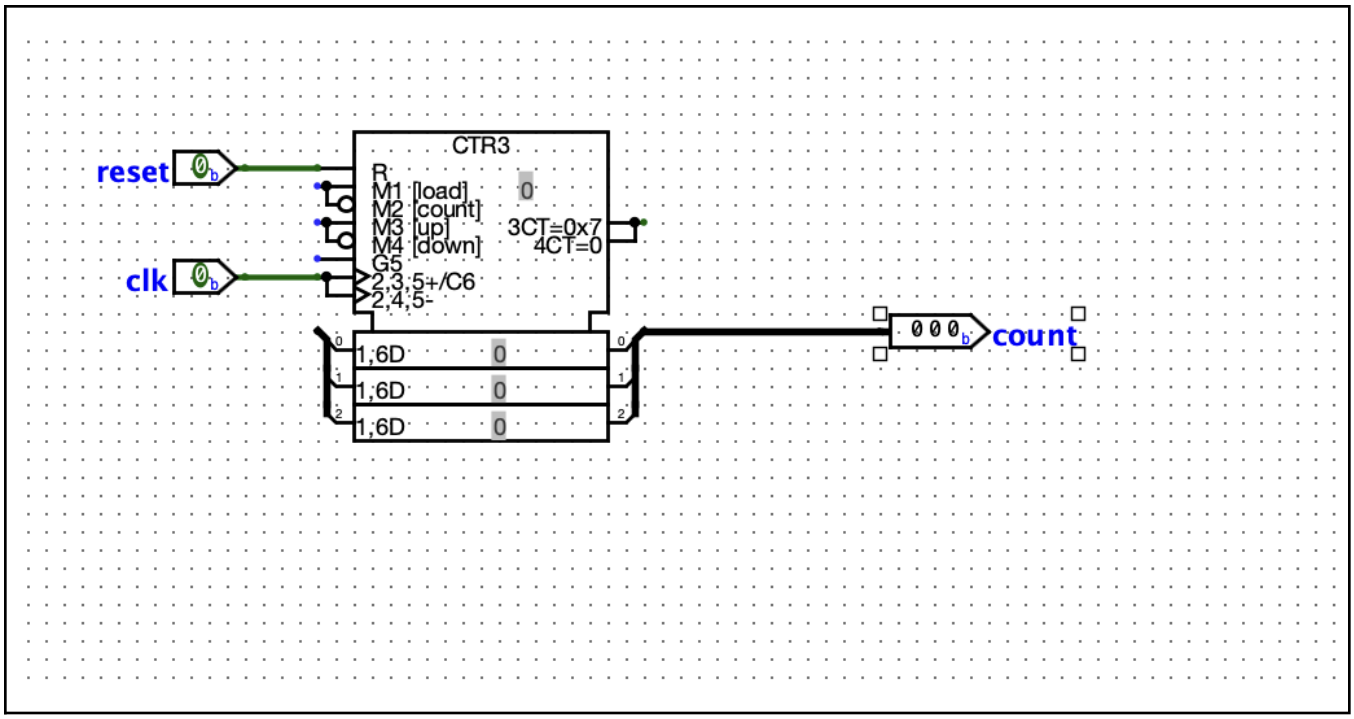
Design and Test of the Programmable Calculator

The Guppy `prog_calculator` circuit consists of the `calculator` connected to the instruction ROM that provides its inputs, which in turn is connected to the program counter that sequences through the ROM addresses.

Completing the program counter `prog_counter` circuit

The program counter circuit `prog_counter` should be a 3-bit incrementing saturating counter that is initialized to 0, counts up to 7, and then stays there until the reset signal is asserted asynchronous with the clock. You may use the built-in *Counter* configured to match this specification. Make sure to create a separate `prog_counter` circuit, i.e., do not just add a *Counter* to `prog_calc`.

schematic for `prog_counter` circuit



You do **not** need to include a timing diagram for `prog_counter`, but you are strongly encouraged to make sure it works correctly before proceeding.

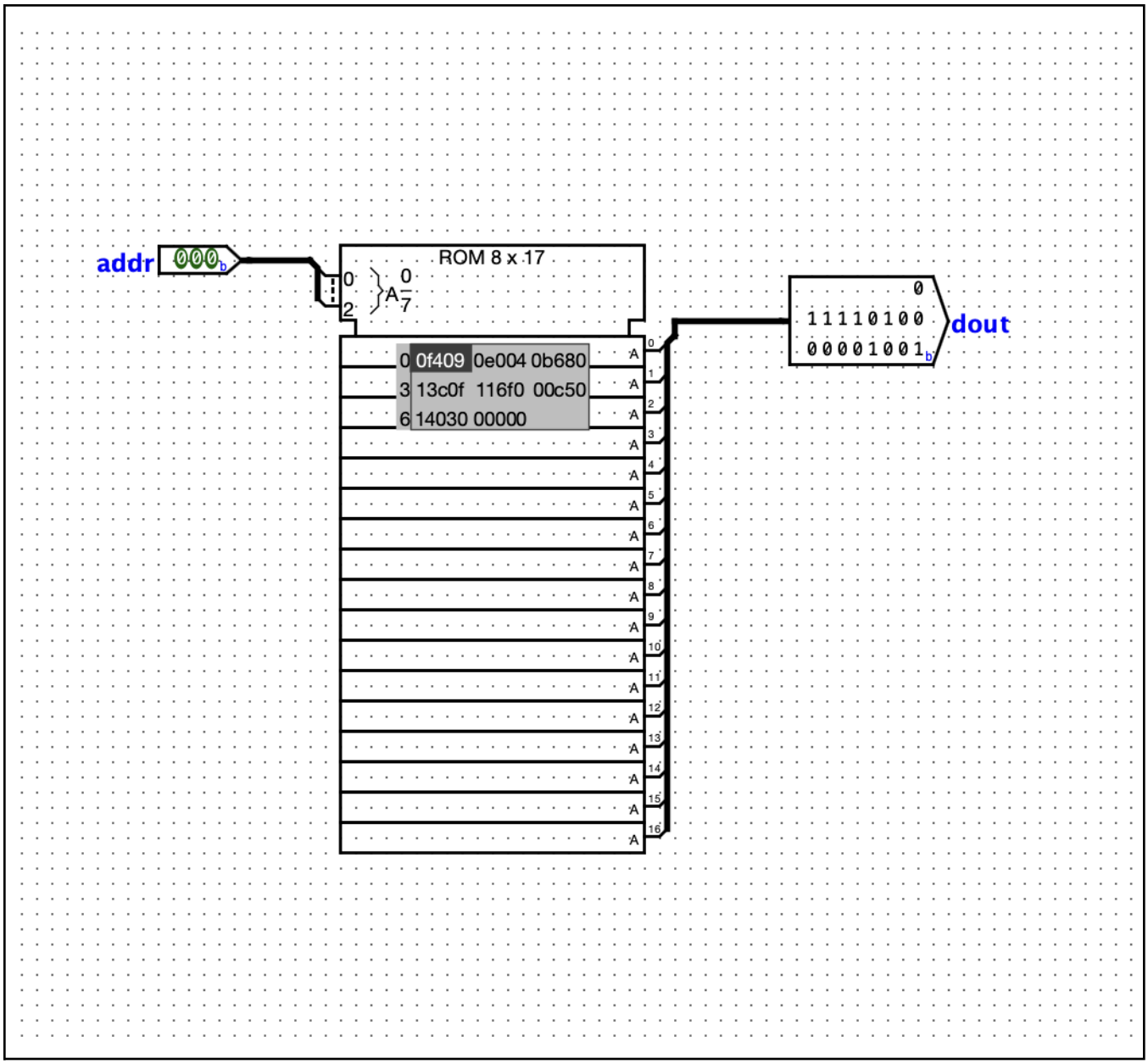
Completing the instruction ROM `inst_rom` circuit

The instruction ROM circuit `inst_rom` should be created with the built-in *ROM*. Configure the ROM to contain an array of 8 words that are each 17 bits wide. The 17 bits in a words are ordered as follows, corresponding to the fields of a Guppy instruction:

16:13	12:10	9:7	6:4	3:0
op	rw	ra	rb	imm4

Initialize the values in the ROM array with the machine code instructions for the Guppy program that encodes the same sequence of register-transfer operations as used to test the `calculator`. Each instruction should be expressed as a 5-digit hex value – see the example on the first page showing how the `ldiu r0, 5` instruction was encoded in 5 hex digits `0x0E005`.

schematic for `inst_rom` with the encoded program



Is the output from `inst_rom` synchronous or asynchronous, and how can you tell?

The output is asynchronous becaus no clock is needed in the input for synchronization.

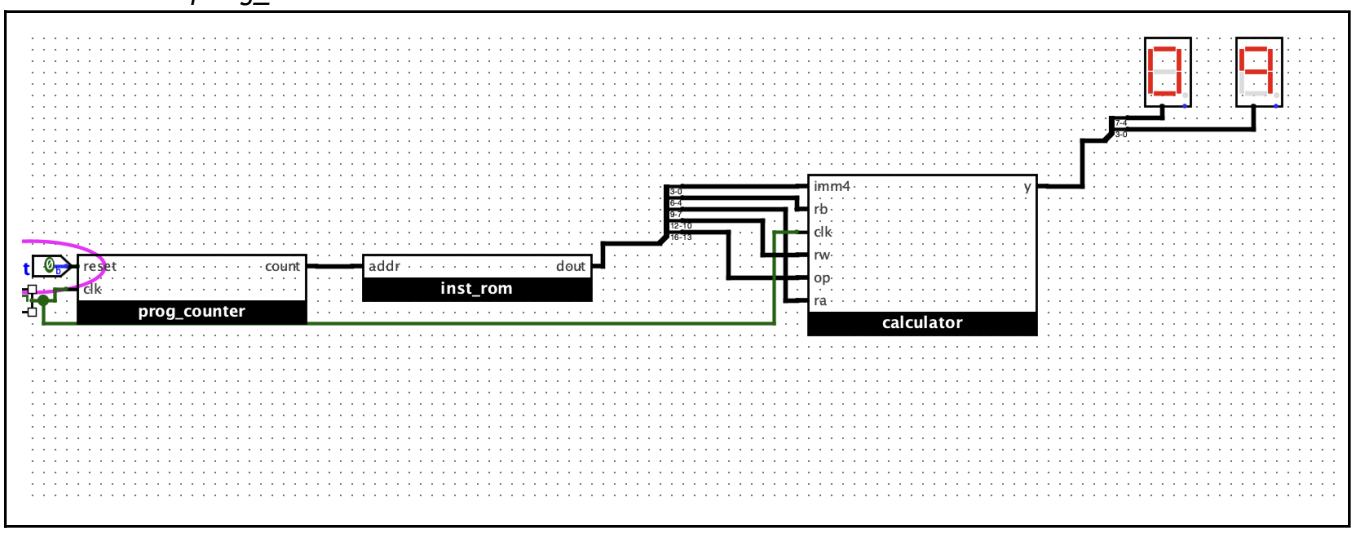
Completing the `prog_calculator` circuit

The final circuit to create is `prog_calculator` wired to match the block diagram on the first page of the assignment. Make sure the program counter has a **Clock** as input (not a named clock) - this should be the *only* **Clock** in your entire .circ file. The port ordering for *Calculator* does not have to be in the order in the block diagram. It may make sense within the *Calculator* circuit to have a different ordering, which will in turn change the ordering in the schematic. If that happens, you can either 0)

carefully wire the circuits as neatly as you can; 1) re-order a splitter's fan-out to match the ordering of the *Calculator* ports – see the "Bit 0, Bit 1, Bit 2… Bit N" options in a splitter's Properties section; or 2) right-click the circuit's name in the list of circuits and click "Edit Circuit Appearance" to define a custom ordering of each pin.

Use the poke tool to reset the program counter and take a screenshot of your entire schematic. *Do not make the clock tick yet.*

Schematic for `prog_calculator` circuit



Upload your complete .circ file to one group member's dropbox and note the location in the space below:

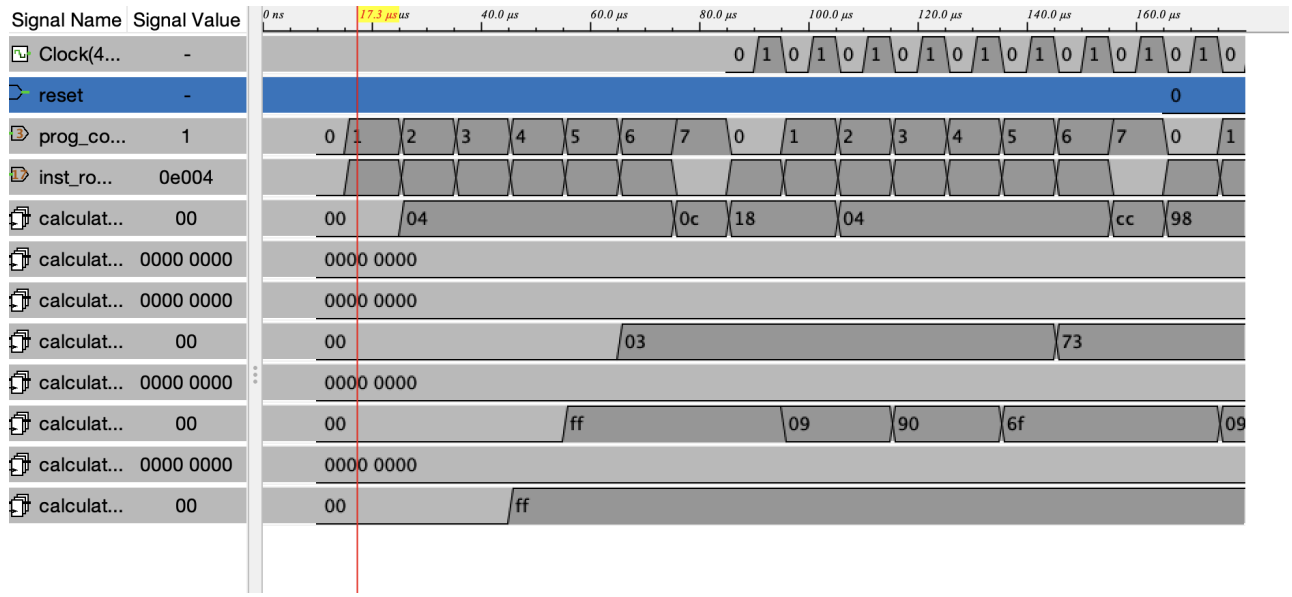
```
/escnfs/courses/sp25-cse-20221.01/dropbox/bmolla/HW07
```

You will generate a timing diagram showing the behavior of `prog_calculator` for your program. Follow these instructions to set up your timing diagram:

1. Add the current instruction, the state of each register, and the PC's value to the timing diagram so you can answer the questions below.
    a. Your signals should appear in the order shown below.
2. Make sure all multi-bit signals are displayed in hex.
3. When taking your screenshots, make sure the signal names and values are completely visible, as shown below.
4. Let the clock run for **two** full clock cycles after PC first becomes 7, and capture that entire range in your timing diagram.
5. Here is an example of all the signals you should see in your timing diagram window. This example shows the initial state of a program with a single instruction, `ldiu r0, 5`.

Signal Name	Signal Value
Clock(200,240)	0
rst	0
y[7..0]	05
pc(460,190)/q[2..0]	0
inst_rom(730,190)/dout[16..0]	0e005
calculator(1120,190)/regfile1w_2r(540,350)/r0[7..0]	00
calculator(1120,190)/regfile1w_2r(540,350)/r1[7..0]	00
calculator(1120,190)/regfile1w_2r(540,350)/r2[7..0]	00
calculator(1120,190)/regfile1w_2r(540,350)/r3[7..0]	00
calculator(1120,190)/regfile1w_2r(540,350)/r4[7..0]	00
calculator(1120,190)/regfile1w_2r(540,350)/r5[7..0]	00
calculator(1120,190)/regfile1w_2r(540,350)/r6[7..0]	00
calculator(1120,190)/regfile1w_2r(540,350)/r7[7..0]	00

Screenshots showing complete timing diagram. Everything must be visible. Break into multiple screenshots if needed.



The hex for the rom isnt showing up but it is as following
0e004
0b680
13c0f
116f0
12c0d
14dd0

After the first clock period, what is in `r5`? Does this match `y`? Explain the difference between `r5` and `y`.

After the first clock period, register 5 is stored with the value 9 and y hold the value 04. This is because y represents the output of the ALU/data path before it is written to the register file.What is in rw is the value in y on the next rising edge. R5 is a reflection of the result of the previous clock

cycle's operation and y shows the result of the current operation.

At the end of the simulation, r0 should no longer be 4. What is its value, and what specifically in the operation of prog_calculator caused it to become that specific value?

The value is 8 because the preogram counter runs the last program in inst_rom. This gives a value of 00000 which adds r0 and r0 (basically doubling it)and save it to r0

Deliverable Checklist

ALU (5 points)

	Deliverable	Points
1.	schematic for updated ALU	5

Register File (5 points)

	Deliverable	Points
2.	schematic for regfile circuit	5

Calculator (22 points)

	Deliverable	Points
3a.	schematic for calculator circuit	10
3b.	table with tests for calculator circuit	5
3c.	screenshot of timing diagram for calculator	5
3d.	screenshot of regfile circuit showing the 8 registers	2

Program Counter and Instruction ROM (12 points)

	Deliverable	Points
4a.	schematic for prog_counter	5
4b.	schematic for inst_rom with the encoded program	5
4c.	Is the output from inst_rom synchronous or asynchronous, and how can you tell?	2

Programmable Calculator (36 points)

	Deliverable	Points
4a.	schematic for prog_calculator circuit (and dropbox path)	18
4b.	Timing diagram	10
4c.	r5's value, y's value, explanation of difference after first clock period	3
4d.	r0's value and explanation for what happened to r0	5