Lab 4: The Complete Microprocessor

Prerequisites: Before beginning this lab, you must:

- Understand how to use the tool flow (See the installation guide and Lab 0)
- Have completed Lab 1: Half Adder, Full Adder, 4-bit Incrementer and Adder.
- Have completed Lab 2: Multiplexers, Decoders, and the Arithmetic Logic Unit.
- Have completed Lab 3: Registers, Counters, and the "Brainless CPU".

Equipment: Personal computer with the required software installed.

Files to copy from Lab 3 (do NOT copy from Lab 1 or Lab 2!):

alu.dig
and_add.dig
brainless.dig
four_bit_adder.dig
four_bit_mux.dig
four_bit_reg.dig
full_adder.dig
half_adder.dig
incrementer.dig
not_neg.dig
program_ctr.dig
program_ram.dig
two_bit_mux.dig

Files to download:

rom_vals.hex
ram_vals.hex
micro_top.v
micro_stim.v

Objectives: When you have completed this lab, you will be able to:

- Build a Memory Address Generator for a microprocessor.
- Design a ROM-based controller for a microprocessor.
- Create an instruction set for a microprocessor.
- Use the instruction set to write a program and enter it into a RAM.
- Execute programs in Digital and in Verilog.

Introduction

In this lab we will complete the microprocessor design by building a memory address generator and a controller and adding them to your brainless microprocessor. We'll also define an instruction set for the controller. Finally, you will use the language inherent in your instruction set to create a simple program, enter the program in your microprocessor's memory and execute the program. You will test the program first in Digital and then in Verilog.

Warning: Use the signal and circuit names provided! Verilog does not allow names to start with a number or names that have dashes!

Create a folder named Lab4. Into that folder, copy the files listed above from Lab 3. Be sure to only copy the required files. While it is tempting to do Lab 4 in the same directory where you did a prior lab, it is advisable to start in a fresh directory in case something goes wrong. That way, you still have the pristine prior lab results to copy again. In addition, this means that the Lab4 folder will only have the files necessary for Lab 4.

Once you've copied the files from Lab 3, download the files provided for Lab 4 and place them in the Lab4 folder. Now, you're ready to start!

NOTE: You are required to design the circuits as presented in this document. Even though Digital supplies many of the functions we will design, you are not to use them.

Task 4-1: Build and Test the Memory-Address-Generation Circuit

Now that your brainless microprocessor is working, we will add additional pieces of circuitry to it and assemble the complete microprocessor. The first step is to modify the 4-bit incrementer you built in Task 3-2 so that you can use it to generate the memory addresses for your CPU. The memory address generation circuit is shown in Figure 1. The memory address generation circuit is comprised of two independent pieces. One piece consists of a register and increment circuit, which is your 4-bit incrementer, and is known as the Program Counter (PC). The other piece, which accepts addresses from the data bus, is known as the Memory Address Register (MAR). This register can be loaded from the data bus and holds the address of a storage location that we want to access. This address does not have to be in sequence with the other addresses. Thus, the MAR allows us to access a specific address and that address is usually obtained from our RAM. In order to load the memory address from the **data_bus** to the MAR, we have an enable signal that is called **load_mar**. The output of the address generation circuit is generated by a mux, which controls whether the PC or MAR drives the address bus, **addr_bus**. The control input to the mux, **use_pc** (short for Use Program Counter), is high when the PC values are driving the address bus and low when the MAR contents are to be supplied to the address bus.

The memory address generation circuit will allow our processor to access memory locations both in and out of sequential order. The circuit can:

- Automatically increment memory addresses after each instruction or operand is retrieved by the CPU.
- Change the memory reference address so that we can store data to arbitrary memory locations or read data from arbitrary memory locations.

Open Digital and select File->New and File->Save As and name the circuit addr_gen, making sure it is saved in the Lab4 folder. Then build the circuit shown in Figure 1 using the circuits you built in prior labs plus an inverter. Note that the **data_bus** input and the **addr_bus** output are 4-bits wide. And remember to use the \ in front of underscores when entering the labels. For

example, data_bus. You'll also want to change the width of the program_ctr to 4. (Open the program_ctr properties, select Open Circuit, then Edit->Circuit specific settings.) You may want to label the MAR as "MAR" to remind you what this four-bit register is holding.

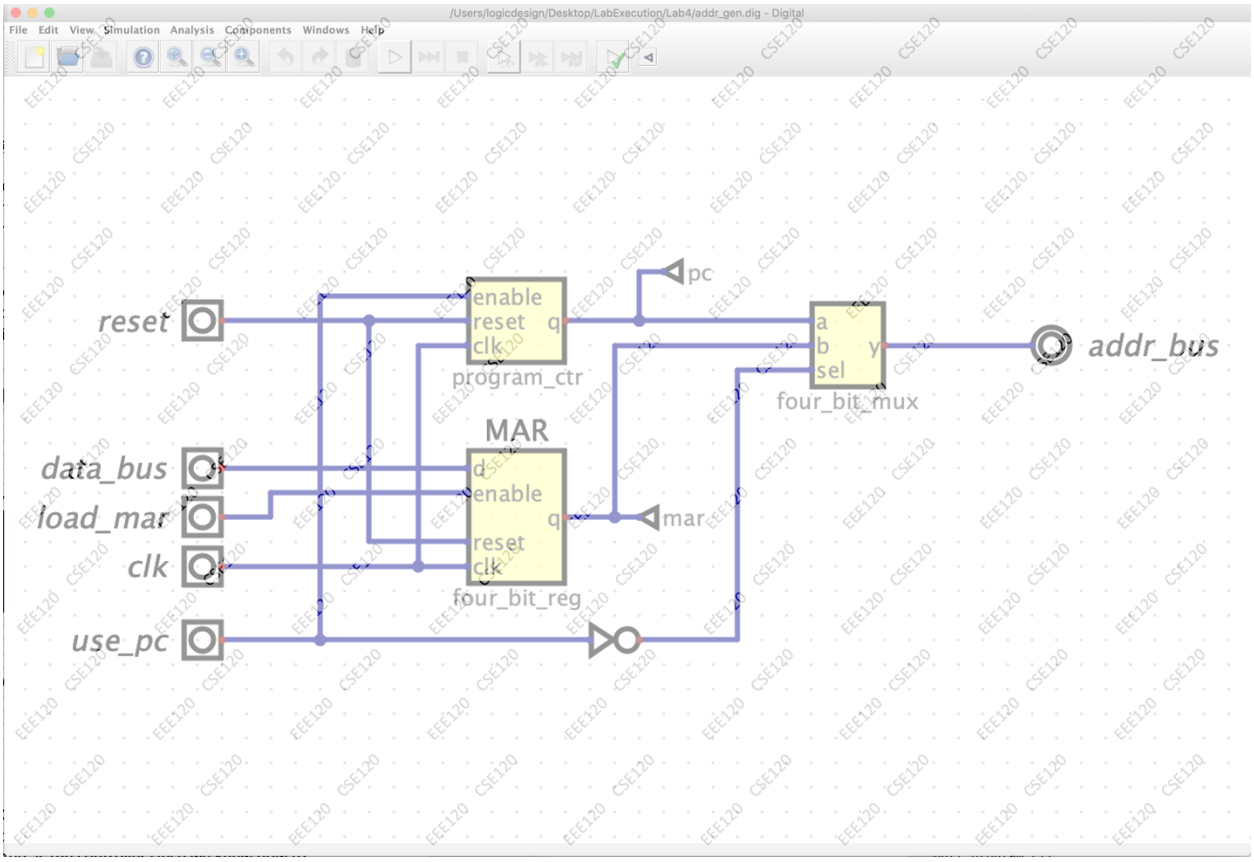

Figure 1. The Memory Address Generation circuit.

Notice the small triangle symbols labeled "pc" and "mar". We're going to add these to make finding these wires easier when we simulate in Verilog. In Digital, this component is called a Tunnel. In other environments, they are also referred to as connectors. The normal use case for a Tunnel is to allow two distant points to be connected without requiring a wire to run between them. That is, the Tunnel establishes a virtual connection. The advantage is that you can connect distant things without your circuit becoming a mess of wires. In our case, we'll use the Tunnel components to give the wires names as there is no other way to do so in Digital.

The best way to add the Tunnel component is to add all the Tunnel instances you need and name them before connecting them to the wires. Select Components->Wires->Tunnel and place them as shown. Open their properties and set the net names as they appear in the diagram. Finally, connect them to their respective wires.

When done, simulate the circuit in Digital. Does it work properly? Can you load values into the Memory Address Register? Make sure that both the PC and the MAR can be used to drive the

addr_bus. Once you are convinced your circuit is working properly, end the simulation and save your design. Take a screenshot of the circuit and paste it into your template.

Task 4-2: Build and Test the Controller Circuit

Remember the brainless microprocessor from Lab 3? How did it know how to execute each of the instructions in our program? The answer, of course, is that we acted as the controller since we knew how to set all of the signals to execute each operation. Next, we will design a controller that will be able to automatically load instructions (in the form of hexadecimal numbers, called operation codes or **"opcodes"**) stored in the Program RAM and then automatically perform the operations needed to carry out each instruction. We will use a ROM to decode the instructions and generate all control signals to operate the CPU. The only remaining signals that we will control ourselves are **clk**, **reset**, and **data_in**.

The controller we will design and build is a ROM-based finite state machine as shown in Figure 2. Why do we need a finite state machine? It turns out that some of our operations cannot be completed in one clock cycle. Think of the operation that writes the accumulator content to the RAM at a specific address. We might need one or more extra clock cycles to accomplish this. Rather than creating lots of opcodes for each individual step, our finite state machine allows us to break down the opcodes into "micro-operations" or "micro-ops" (µ-ops). Each opcode, now called macro-op, can have up to four micro-ops in our design. This will be enough for all of the operations that we want to perform.

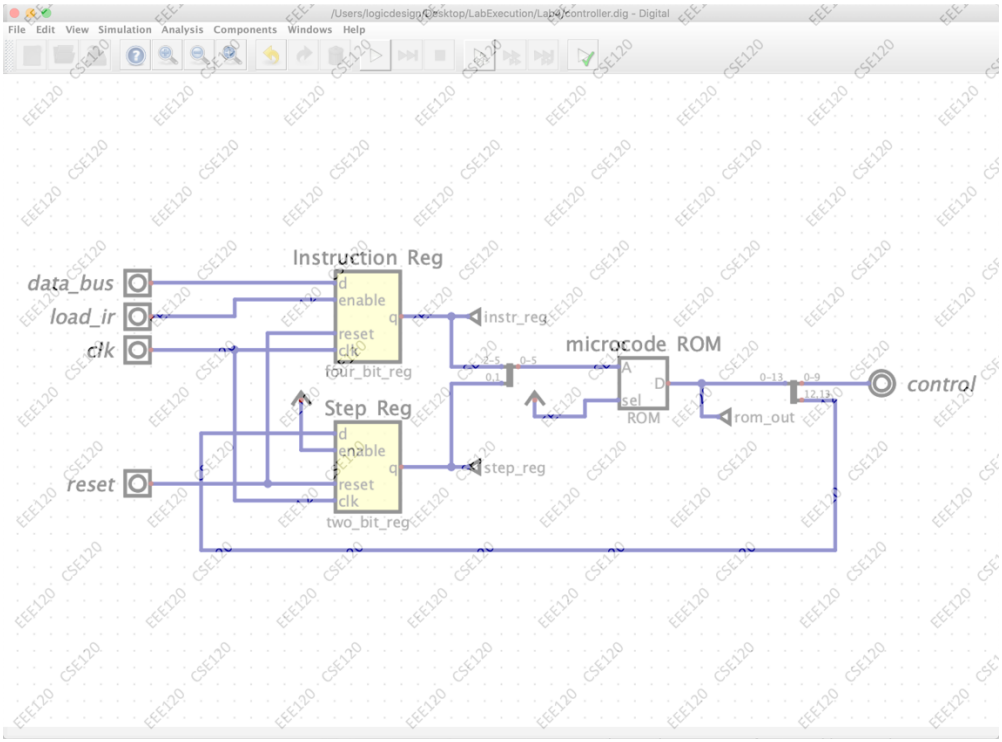

Figure 2. The controller.

The central part of the controller circuit is the “Instruction ROM”. It performs “double duty”, acting as both a decoder for the control signals and as the programmable logic for the finite-state machine. The controller circuit consists of three pieces. One piece is a 4-bit register which holds the current instruction, or opcode, we are currently executing. This is known as the Instruction Register. The second piece is a 2-bit register which holds the number of the micro-op we are executing. (If you recall, each instruction can be broken down into, at most, 4 micro-ops numbered 0 to 3.) This 2-bit register is called the Step Register. The outputs of the two registers form a 6-bit bus which is used as the address into the third piece, the ROM. A ROM is a Read Only Memory into which we will preload values. Our microprocessor will be able to read from the ROM but, as its name implies, the microprocessor will be unable to modify the ROM’s contents.

Why use a ROM? Because it is a straightforward way to implement the logic that we need. Otherwise, we’d have to generate 12 6-variable Karnaugh maps and implement the logic with gates – not an attractive prospect!

As stated above, the output of the Instruction Register and the Step Register are combined to form the address to the ROM. The 4 bits of the Instruction Register are on the left and the 2 bits of the Step Register are on the right. This way, the steps of an instruction are contiguous in the ROM. If we represent the addresses as hexadecimal numbers, that means the first instruction will be contained in addresses 00, 01, 02, and 03. Then the second instruction will use addresses 04, 05, 06. and 07. If the instruction doesn’t need all the steps, that’s ok – we can just load the unused addresses with 0s.

There are two types of values which the ROM produces. Bits 1:0 are used to determine the next value that will be in the Step Register. Bits 11:2 are all control bits. We used all but one of these in the brainless CPU. The only new control signal is the one used to load a new instruction into the Instruction Register.

The IR is loaded with an opcode that is stored in the Program RAM and appears on the Data Bus. In order to load that opcode into the Instruction Register, the controller has to set its **load_ir** output to 1. We call this operation the **“Instruction fetch”**. The “fetch state” will be the first step executed for each of our instructions. The next three steps will be “execute states” to achieve whatever functionality that instruction requires.

To build the controller, we’ll have to create a 2-bit register. To build this register, we’ll first have to create a mux for 2-bit buses as shown in Figure 3. In Digital, select File->Open and select four_bit_mux.dig. **Immediately**, select File->Save As and name the circuit two_bit_bus_mux.

IMORTANT: You must call this two_bit_bus_mux! If you leave out the “bus”, you’ll overwrite the two_bit_mux circuit you created previously and your four_bit_mux will no longer work. (If you already forgot the bus, that’s ok. Just copy the two_bit_mux circuit over from Lab 3 and start this step over.)

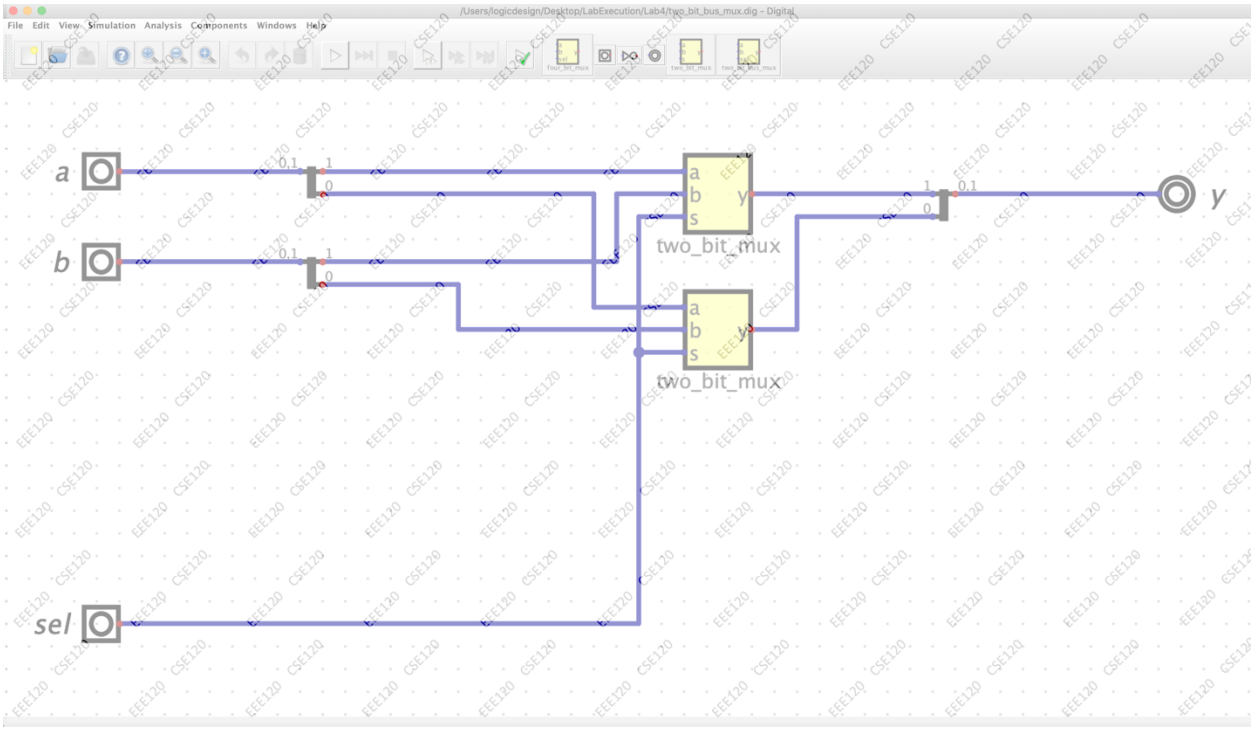

Figure 3. The mux for 2-bit buses.

Open the properties of the splitter/merger components and delete the “3-3,2-2,” leaving just “1-1,0-0” and change the 4 to 2. This way, we’ll just have 2-bit buses rather than 4-bit buses. Now delete the flip-flops no longer connected to the splitter/merger components. Now delete the two_bit_mux components that are no longer connected as well as all the wires that are no longer connected on both ends. Finally, change **a**, **b**, and **y** so that are 2 bits wide. Simulate this circuit in Digital and, when you are satisfied that it works, save it. Take a screen shot and paste it into your template.

Now we’ll finish the two-bit register as shown in Figure 4. Select File->Open and select four_bit_reg.dig. **Immediately**, select File->Save As and name the circuit two_bit_reg. Open the properties of the splitter/merger components and delete the “3-3,2-2,” leaving just “1-1,0-0” and change the 4 to 2. This way, we’ll just have 2-bit buses rather than 4-bit buses. Now delete the flip-flops that are no longer connected as well as all the wires that are no longer connected on both ends. Finally, change **d** and **y** so that are 2 bits wide. Simulate this circuit in Digital and, when you are satisfied that it works, save it and paste a screenshot of it into your template.

We now have the pieces we need to create the controller. In Digital, select File->New. Then select File->Save As and save the new file as controller, again making sure it is saved in the Lab4 folder. Let’s start by placing the ROM. Select Components->Memory->ROM. You’ll notice the ROM has two inputs. **A** is the address and **sel** is the output enable. If the output enable is 0, the ROM outputs are not driven and, therefore, are at the high impedance, or Z state. To avoid that, we’ll tie **sel** to the supply voltage.

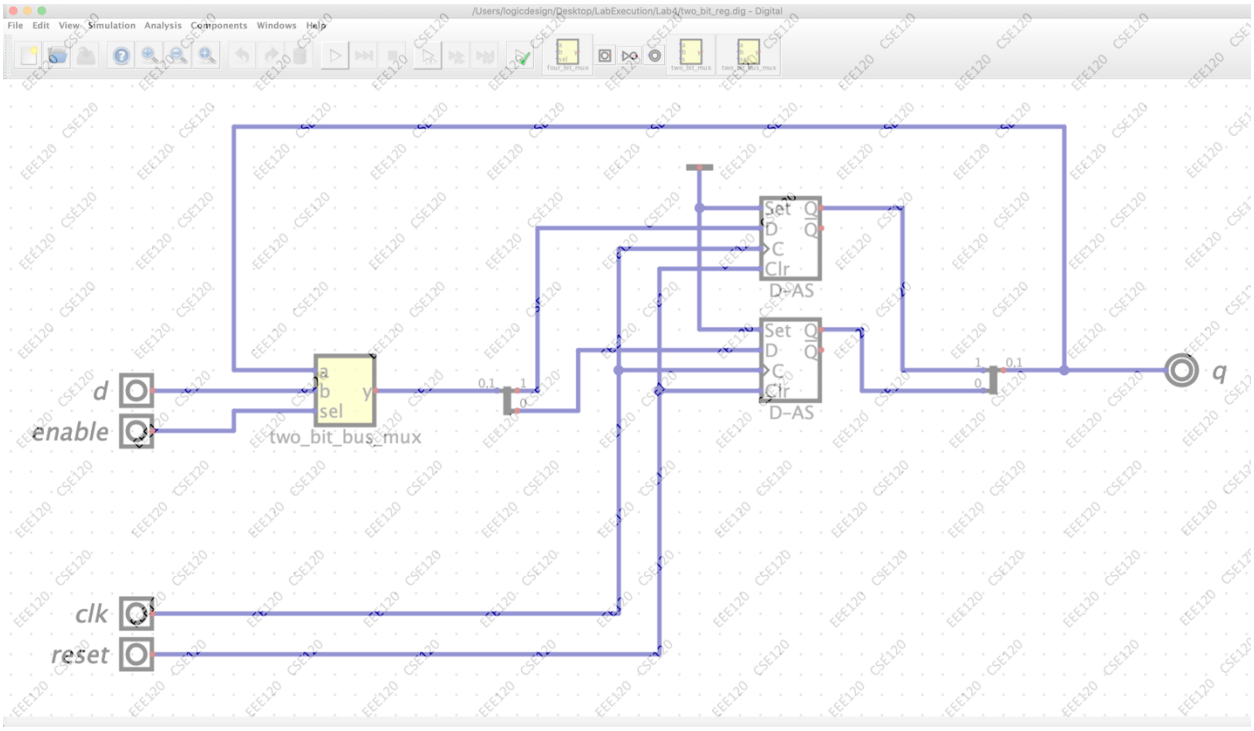

Figure 4. The two-bit register.

Open the ROM properties and fill it out as shown in Figure 5. Then click on the Edit button in the properties window and you'll see a window as shown in Figure 6. At the top of that window is the word File. Select File->Load and another window, shown in Figure 7 will appear. Select rom_vals.hex, which you should have downloaded and placed in the Lab4 folder. Then click Open. The ROM data window should now appear as shown in Figure 8. Click OK and OK to close the ROM windows.

We'll add three Tunnel components to this design. Remember to add the Tunnel components and name them before connecting them to their respective wires. And don't forget the \ in front of the underscores when you set the Net name for each Tunnel.

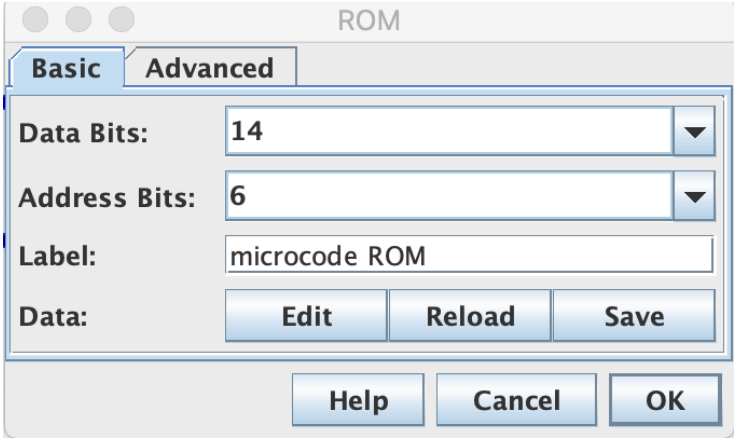

Figure 5. The ROM bit definitions.

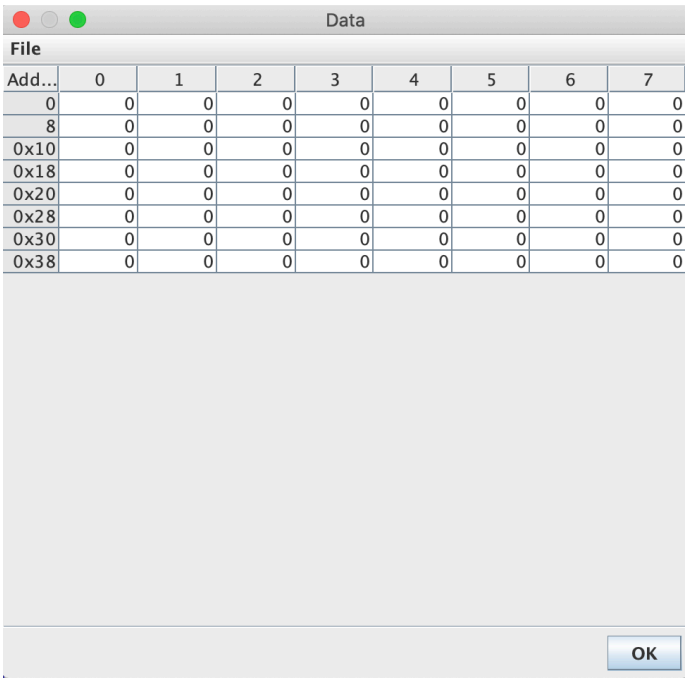

Figure 6. ROM data edit window.

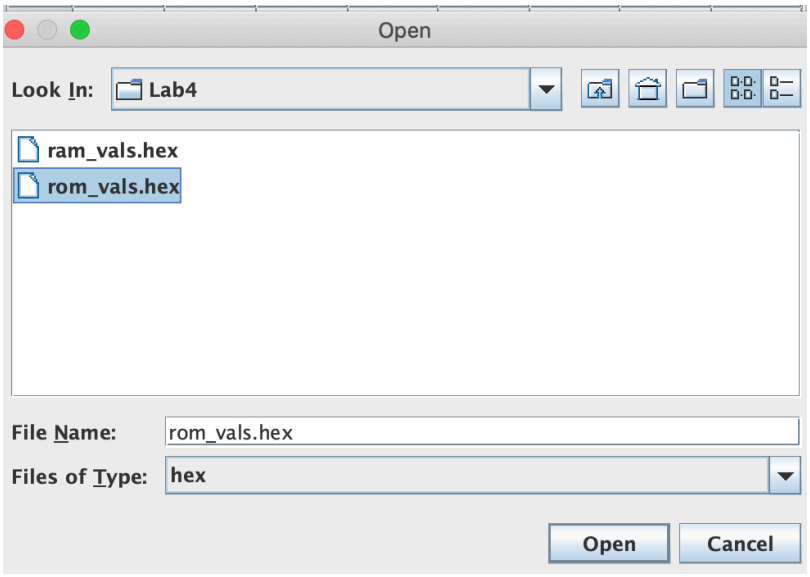

Figure 7. Selecting the file with which to load the ROM.

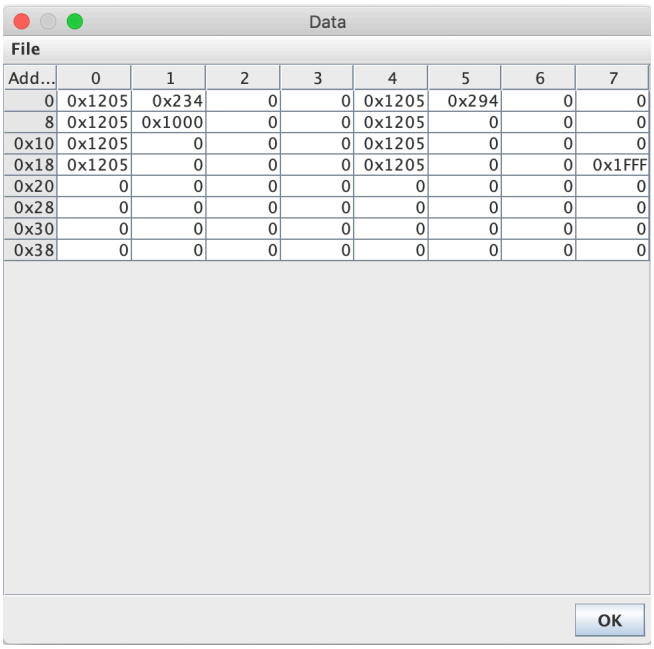

Figure 8. The loaded ROM data.

Now place the four-bit and two-bit registers. Open their properties and select Open circuit. The design will be a bit neater if we reorder the inputs. Select Edit->Order Inputs and match the input order shown in Figure 2. Select Edit->Circuit specific settings and verify the width is 4. Do this for both registers.

The last new work to do is to set up the splitter/mergers. Add the splitter/merger components. Figure 9 shows how to combine the outputs from the registers. Figure 10 shows how to split them back out. Why are we splitting them this way? The upper two bits coming out of the ROM will be the next two bits to be loaded into the Step Register. So we'll pull those off the bus and route them internally. The next two bits are not currently used, so we'll just leave them out. The lower 10 bits will be the various control signals needed to operate the microprocessor. We'll discuss the contents of those bits shortly.

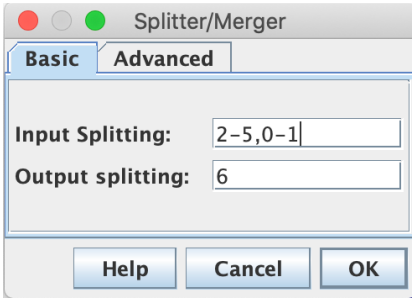

Figure 9. Combining the instr_reg and step_reg buses.

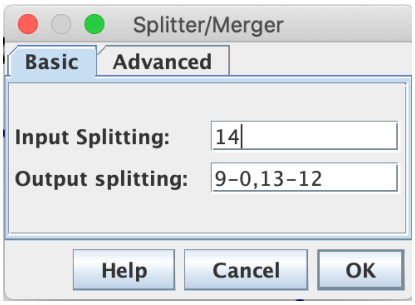

Figure 10. Splitting the ROM output.

Go ahead and add the inputs, the output, and all the wires. Remember that the **data_bus** input is 4 bits wide and the **control** output is 10 bits wide.

Now, let's discuss the contents of the ROM. Figure 11 shows the first 12 entries in the ROM. Remember that the first line, "v2.0 raw" is something that Digital needs to properly read the file. Note that the # character starts a comment so we can annotate the entries.

```
v2.0 raw
1205 # LOAD ACC;    Load IR
0234 #              Load ACC
0000 #              unused
0000 #              unused
1205 # ADD ACC;     Load IR
0294 #              ADD
0000 #              unused
0000 #              unused
1205 # STOP;        Load IR
1000 #              stay here
0000 #              unused
0000 #              unused
```

Figure 11. The first 12 entries of the ROM.

Since this is a hexadecimal file, each digit represents 4 bits. However, the left digit will only represent the numbers 0-3, so there are only 14 bits per row. Bit 13 is on the left and bit 0 is on the right. Table 1 defines what each of the bits represent. Because the first line of each instruction is always 1205, that value has been entered for you. This shows how each instruction will start on a multiple of 4 entries. Note that the very last row in the file, which isn't shown in the figure, is assigned the value 3FFF. This was done to force Digital to write out all the entries of the ROM when exporting to Digital. You'll see why this is needed in a later task.

Table 1. Bit definitions for ROM entries

Bit #	Signal Name	Function
13:12	next_step[1:0]	The next step of this instruction to execute.
11:10	-	These are unused. They are included so the next step bits align nicely
9	use_pc	Propagate the Program Counter (PC) to the address bus and increment the PC; otherwise select the Memory Address Register (MAR)
8	load_mar	Load the MAR
7	arith	One of the three controls for the ALU as defined in Lab 3
6	invert	One of the three controls for the ALU as defined in Lab 3
5	pass	One of the three controls for the ALU as defined in Lab 3
4	load_acc	Load the accumulator.
3	acc_to_db	Propagate the accumulator to the data bus; otherwise select the data mux output
2	read	The data mux should select the RAM output; otherwise it selects **data_in**
1	write	Write the value on the data bus into the RAM.
0	load_ir	Load the Instruction Register

So how does this work? When an instruction is in the instruction register, it combines with the value in the step register to select a value from the ROM. This value divides into two functions. The first function is to specify the next step in the instruction to execute. Since each instruction has up to 4 steps associated with it, we need two bits dedicated to this task. These are bits 13 and 12. The second function is to control how the rest of the circuit is operating. These are the 10 bits 9 to 0.

Note that bits 11 and 10 are unused. It is a common practice when designing hardware to leave unused bits in order to line things up nicely. Consider the first data line in Figure 11: 1205. It is easy to see that this means the **control** output from the controller will be 205. On the other hand, if we moved the next_step bits down to eliminate the unused bits, the first data line would be 605. And if you're trying to figure out if the output is correct, you'd have to think about what that number would be without the top two bits. And that can, at times, become painful since one sometimes forgets how many bits to ignore. Another advantage of leaving some unused bits is that there's room for enhancements. (Or bug fixes!)

How do you decide which values to assign in each step? The idea is to have the control signals steer the data to where it needs to go and to get the various components to perform the required logic. Let's analyze the Load ACC command, step by step:

Step 0: This step is the same for all the instructions and is used to get an instruction from the RAM to the instruction register. Table 2 is used to explain how each value was chosen. The bold horizontal lines in the table separate the binary bits that form each hexadecimal digit.

Table 2. Step 0 of the Load ACC instruction.

Signal Name	Value	Discussion
next_step[1:0]	01	This takes us to the next step of the instruction.
unused bits	00	-
use_pc	1	Use the PC as the address to access the RAM and increment the PC
load_mar	0	Not loading the MAR
arith	0	Not doing an ALU function so default to 0
invert	0	Not doing an ALU function so default to 0
pass	0	Not doing an ALU function so default to 0
load_acc	0	Not loading the accumulator
acc_to_db	0	Need to propagate the data mux output to the data bus
read	1	The data mux should select the RAM output
write	0	Not writing to the RAM
load_ir	1	Load the value on the data bus into the instruction register

If you take these bits from top to bottom, you have 01_0010_0000_0101 in binary. Converting that to hexadecimal, we get 1205, the value we see in the rom_vals.hex file. Once this step is executed, we're ready to move on to step 1.

Step 1: This step will be similar for several of the instructions. In this case, all we are doing is loading a value we are reading from the RAM into the accumulator. The values required are shown in Table 3. Since this is the last step needed for this instruction, we'll return to step 0 to fetch the next instruction.

Table 3. Step 1 of the Load ACC instruction.

Signal Name	Value	Discussion
next_step[1:0]	00	This takes us back to step 0 so we can fetch the next instruction.
unused bits	00	-
use_pc	1	Use the PC as the address to access the RAM and increment the PC
load_mar	0	Not loading the MAR
arith	0	Not doing an ALU math function
invert	0	Not inverting
pass	1	Pass the value from the data bus to the ALU output
load_acc	1	Load the accumulator with the ALU output
acc_to_db	0	Need to propagate the data mux output to the data bus
read	1	The data mux should select the RAM output
write	0	Not writing to the RAM
load_ir	0	Not loading the instruction register

If you take these bits from top to bottom, you have 00_0010_0011_0100 in binary. Converting that to hexadecimal, we get 0234, the value we see in the rom_vals.hex file. Once this step is executed, we're ready to move back to step 0.

Note that when each instruction completes, it returns to step 0 and fetches the next instruction. This means that the next instruction actually starts on its step 1. Make sure this is clear before proceeding.

Additional instructions are created in much the same way. At each step, you decide which values to assign to the ROM entries to get the behavior you want.

The one strange instruction, STOP, is a bit different. Its step 1 simply stays in step 1 and all the other bits are 0. That way, your design stays in a final, stable state so you can verify that the circuit worked correctly.

Simulate the controller in Digital and satisfy yourself that it is working correctly. Be sure to save it and take a screenshot of the circuit and paste it into your template.

Task 4-3: Build the Complete Microprocessor Circuit

Now it's time to complete our microprocessor! Open Digital, if it isn't already open, and select File->New and then File->Save As and name the design microprocessor. Make sure it saves into the Lab4 folder! Implement the microprocessor shown in Figure 12.

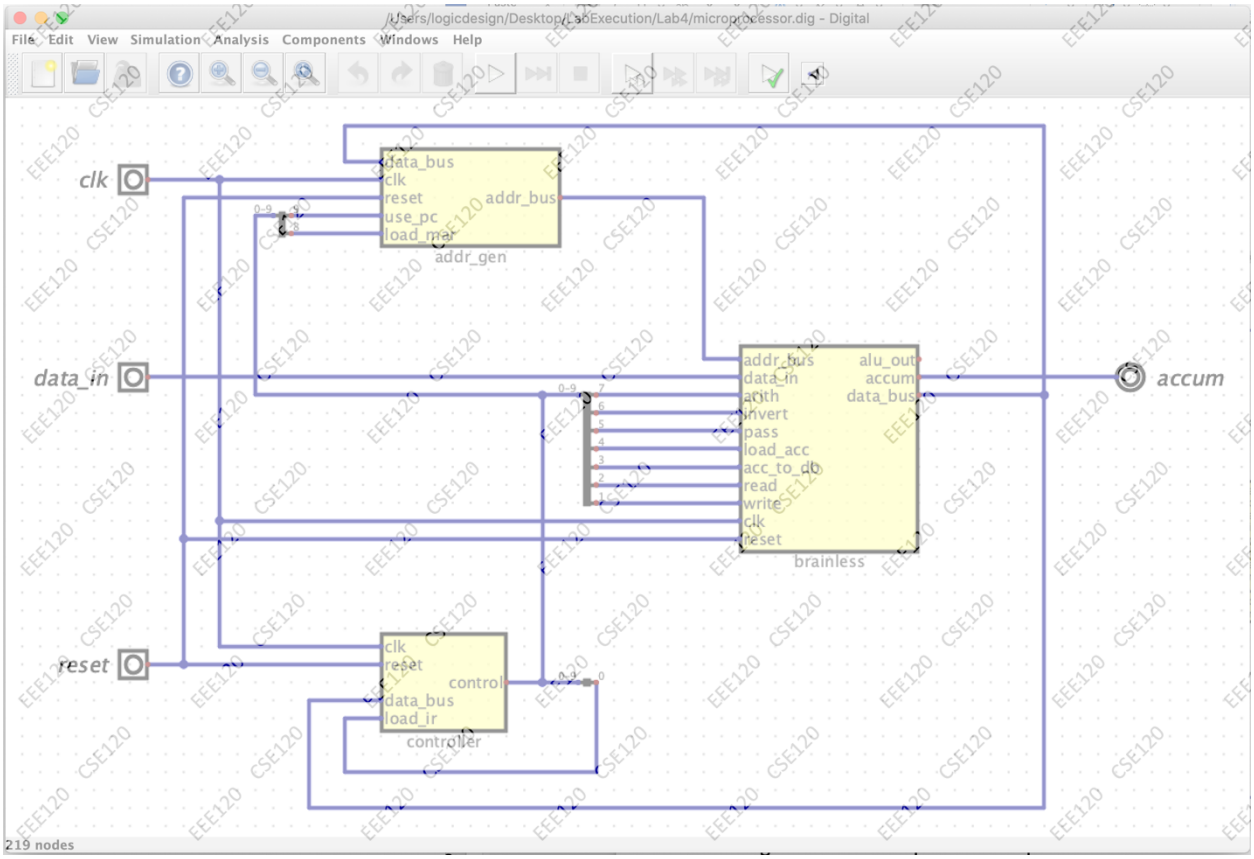

Figure 12. The completed microprocessor.

The circuit shown in Figure 12 is hierarchical. This means that the guts of the three large modules are hidden from view. This makes connecting them much easier and has the added benefit of a much simpler schematic. However, this does make it harder to see how all the pieces work together. Therefore, another view of the completed microprocess is shown in Figure 21 at the end of this lab.

Note that the subcircuit inputs have been reordered to make the wiring much simpler. Remember to include the backslash when you label **data_in** and that both **data_in** and **accum** are 4 bits wide.

Open the properties of the subcircuits and set the width of the controller to 7 and the width of both the addr_gen and brainless subcircuits to 10.

Notice how the control bus is distributed. The programming for the splitters is shown in Figures 13-15. Note that you can resize the Splitter/Merger window to accommodate the number of splits required in front of the brainless CPU.

Finally, we need to specify the RAM file to be loaded at startup. Select Edit->Circuit specific settings and click on the Advanced tab. Select “Preload program memory at startup.” Then, on the next line, click the 3 dots and navigate to your Lab4 folder and select the ram_vals.hex file that you downloaded. Then click OK and save the design.

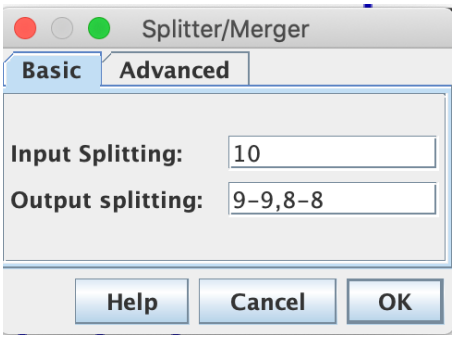

Figure 13. The splitter in front of addr_gen.

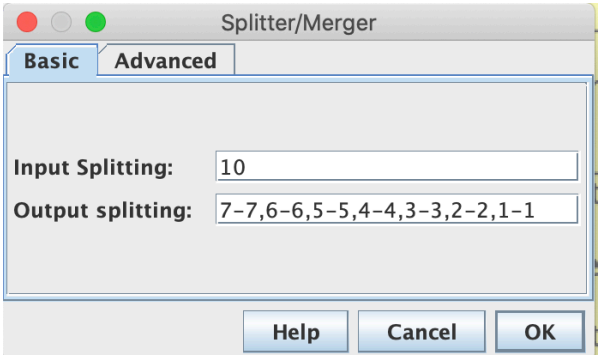

Figure 14. The splitter in front of the brainless CPU.

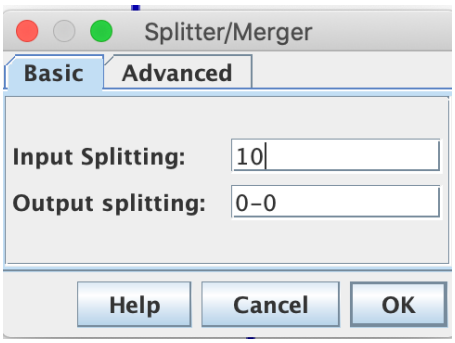

Figure 15. The splitter to extract **load_ir**.

Now we'll simulate your design in Digital to see if the program we've provided works. First, let's have a look at the contents of ram_vals.hex shown in Figure 16.

```
v2.0 raw
0   # Load ACC
3   #   value to load
1   # ADD ACC
5   #   value to add
2   # STOP
```

Figure 16. The first 5 values in the ram_vals.hex file.

The first line in the file, v2.0 raw, makes it so Digital will properly load the file. Then we start an actual program! The 0 is the instruction to load the accumulator with a value and the value to load, 3, is on the next line. Following that is a 1, which is the ADD instruction, which adds the next value, 5, to the value currently in the accumulator. Finally, 2 is the STOP instruction. If you simulate digital using this file, you should see the following behavior:

When you start the simulation, the circuit will behave as if it's been reset. To be certain, set the **reset** to 1 and then back to 0 by clicking on it twice. The **addr_bus** and **data_bus** should both be 0, and the **use_pc**, **read**, and **load_ir** signals should all be 1. The circuit is ready to load an instruction.

On the first positive edge of **clk**, which occurs when you click **clk**, the first instruction will be loaded into the Instruction Register. You should see the **addr_bus** change to 1 and the **data_bus** change to 3. In addition, the **pass**, **load_acc**, and **read** lines should all be 1. You're now ready to get the value of 3 into the accumulator. Click **clk** again so it returns to 0.

Now click **clk** so you get a second positive edge. The **accum** output should now be 3. In addition, the **addr_bus** should be 2, the **data_bus** should be 1, and the **use_pc**, **read**, and **load_ir** signals should be 1. We're ready to load the next instruction. Click **clk** again so it returns to 0.

Click **clk** to get a third positive edge. The **accum** output should still be 3. The **addr_bus** should have changed to 3 resulting in the **data_bus** now being 5. The **use_pc**, **read**, **arith**, and **load_acc** signals are all 1. Great – we're ready to add the 5 to the 3 already in the accumulator. Click **clk** again so it returns to 0.

Click **clk** to get a fourth positive edge. Success! The **accum** output is now 8 and the controller has set things up to read the next instruction: **use_pc**, **read**, and **load_ir** are all 1. The **addr_bus** is 4 and the **data_bus** is 2. Click **clk** again so it returns to 0.

Finally, click **clk** one more time to get a fifth positive edge. The **accum** output stays 8 and all of the control signals are 0 because the STOP instruction has been loaded. Since **use_pc** is 0, the **addr_bus** is now equal to the Memory Address Register. Since we've not loaded the MAR, it is still 0 and so, too, is **addr_bus**. No matter how many more times you click on **clk**, nothing will change. Figure 17 shows the final state of the circuit following the simulation.

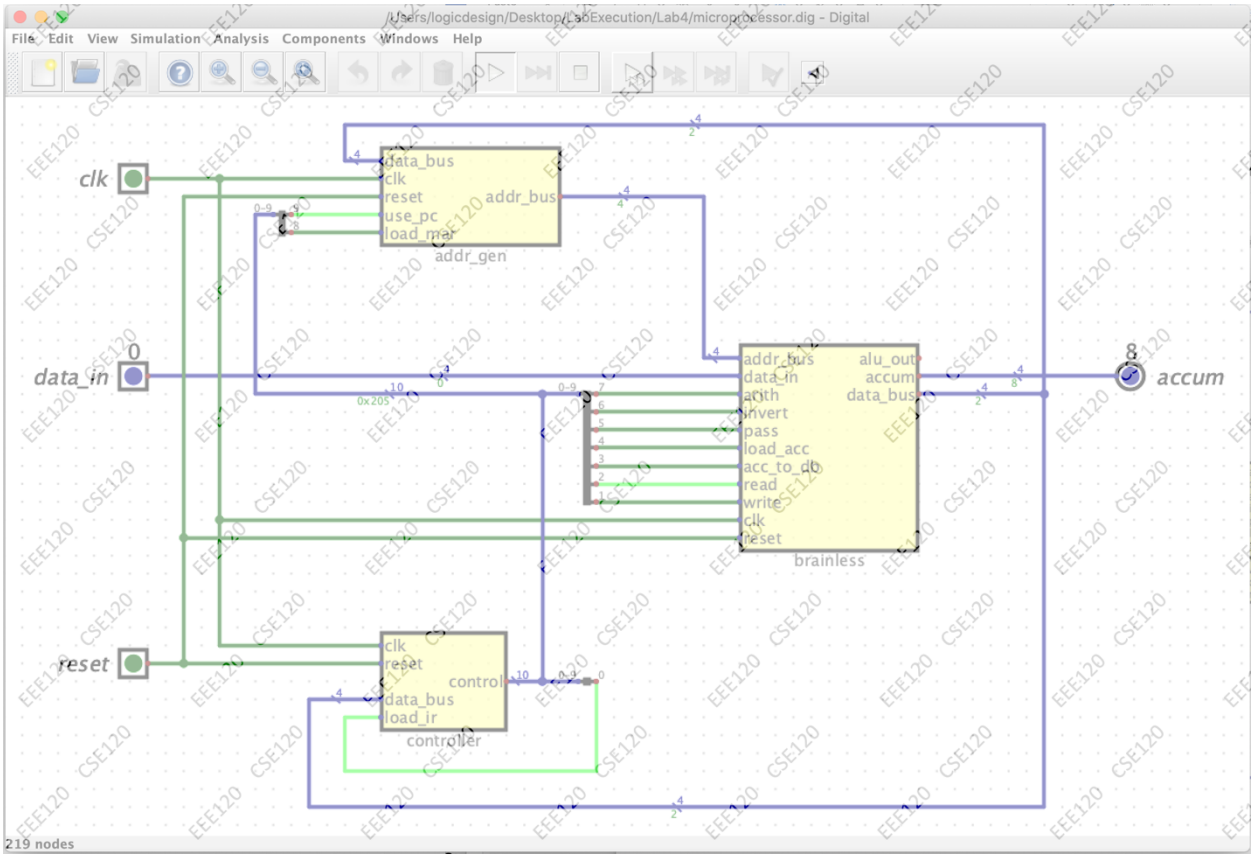

Figure 17. The final state of the microprocessor following the first simulation.

Make sure you understand how the above simulation worked before proceeding as you'll be writing your own tests and implementing your own instructions in the next task.

The ram_vals.hex file contains another 11 lines of 0 which you can use as necessary in the next task to build your own programs.

Once your simulation is working, take a screenshot showing the final values as shown in Figure 17 and paste it into your template. Then stop the simulation, save the design, and then select File->Export->Export to Verilog. Make sure it saves to the Lab4 folder.

Task 4-4: Simulate the Design in Verilog

In order to simulate in Verilog, you'll have to modify the RAM module as we did in Lab 3. Edit microprocessor.v and find the code shown in Figure 18.

```
module DIG_RAMDualPort
#(
   parameter Bits = 8,
   parameter AddrBits = 4
)
(
  input [(AddrBits-1):0] A,
  input [(Bits-1):0] Din,
  input str,
  input C,
  input ld,
  output [(Bits-1):0] D
);
  reg [(Bits-1):0] memory[0:((1 << AddrBits) - 1)];

  assign D = ld? memory[A] : 'hz;

  always @ (posedge C) begin
    if (str)
      memory[A] <= Din;
  end
endmodule
```

Figure 18. The original RAM code in microprocessor.v.

Now modify it by adding the lines shown in bold in Figure 19.

```
module DIG_RAMDualPort
#(
   parameter Bits = 8,
   parameter AddrBits = 4
)
(
 input [(AddrBits-1):0] A,
 input [(Bits-1):0] Din,
 input str,
 input C,
 input ld,
 output [(Bits-1):0] D
);
 reg [(Bits-1):0] memory[0:((1 << AddrBits) - 1)];

 assign D = ld? memory[A] : 'hz;

 always @ (posedge C) begin
  if (str)
    memory[A] <= Din;
 end

 initial
 begin
   $readmemh("ram_vals.txt",memory);
 end
endmodule
```

Figure 19. Add the lines in bold to the RAM module in microprocessor.v.

Now, copy the file ram_vals.hex to ram_vals.txt and delete the first line, v2.0 raw. In addition, the comment delimiter in Verilog is //, so you'll need to substitute // for each instance of # in ram_vals.txt. If this isn't clear, check out the comments in micro_top.v.

Let's now look at the file micro_stim.v that you downloaded and placed in your Lab4 folder. This file will allow you to simulate for up to 32 clocks. Table 2 shows the bit assignments for the test_vals array.

Table 4: Bit definitions for micro_stim.v.

Bit #	11:8	7	6:5	4	3:0
Meaning	exp_accum	done	unused	reset	data_in

The **exp_accum** value is what you expect the accumulator contents to be after each clock cycle. The **reset** and **data_in** signals are the inputs you'll control since the clock will be provided for you. The **done** signal is new. Set the **done** signal to 1 when you want the simulation to end. If you forget to set it, then the simulation will automatically end after 32 clock cycles and a message will print out telling you that's why it ended. The version of micro_stim.v you've been given will run the simulation you ran in Digital once you create the ram_vals.txt file and modify microprocessor.v as described above.

You're now ready to simulate! Execute:

iverilog -o microprocessor.exe microprocessor.v micro_top.v micro_stim.v
On Mac: ./microprocessor.exe
On Windows: vvp microprocessor.exe

Now, open GTKWave and set up your view similar to that shown in Figure 20. Grouping the signals this way makes it really easy to see what is going on in the circuit. We'll also see why we used the Tunnel components. Open the waves using File->Open New Tab and select micro_waves.vcd.

To create the waves as shown in the figure, click on the symbol next to micro_top and it will expand to show the microprocessor circuit name. Now click on the symbol next to the microprocessor and that section of your window should match that shown in the figure. Select microprocessor and the signals should appear in the Signals section on the lower left. Double clock on **clk** and **reset**. These are signals that go to all the modules so it's handy to have them up top rather than repeating them with each group of signals. Now select Edit->Insert Blank to insert a blank row. This separation makes things easier to see.

Let's add the signals associated with the controller circuit. First, let's name this group so we remember where these signals are used or generated. Select Edit->Insert Comment and give the group the name Controller. Now select the controller in the hierarchy panel (that's the small pane in the upper left) and the controller signals will appear. Add the signals as shown in the figure. Note that some of the signals in the controller have names of the form s#. If we hadn't used the Tunnel components, the **instr_reg**, **step_reg**, and **rom_out** signals would have been of that form as well and we would have had figure out which signals they were. While not too difficult in this case, it would have been hard if they all had the same number of bits!

Now add another blank and another comment as shown and add the signals from the addr_gen module. Once done, add a final blank and comment and add the signals from the brainless module.

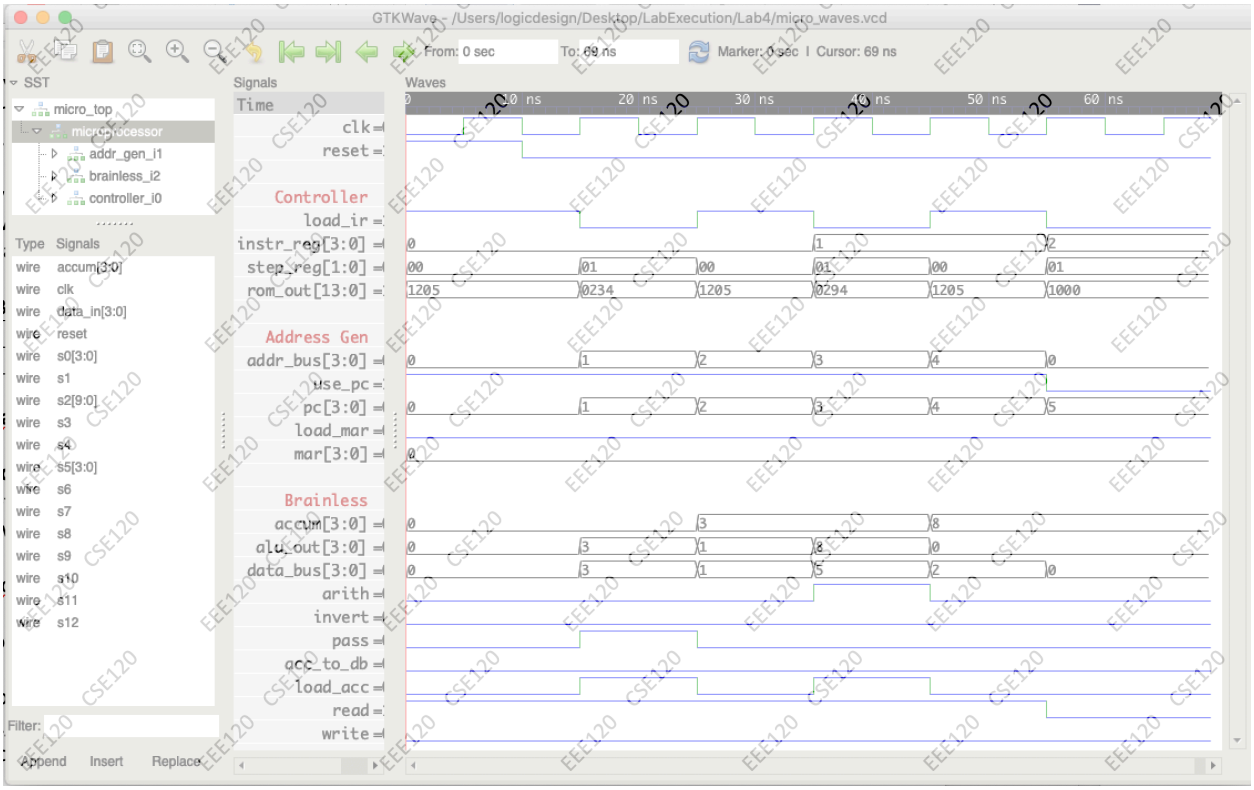

Figure 20. Simulation waveform.

Remember that you can select File->Write Save File to save the waveform setup so you can come back later and select File->Read Save File to get it back. That way, you don't have to recreate it every time. Also recall that you can select File->Reload Waveform if you've run a new simulation and want to see how things changed rather than having to open things from scratch.

If your waves look like those in the figure, take a screen shot and paste it into your template. Otherwise, you'll need to debug the circuit. **Remember – if you go back to Digital and make changes, you'll have to edit microprocessor.v to add the readmem statement for the RAM!**

Task 4-5: Add the AND, ZERO, SUB, and STORE ACC Instructions

In this task you get to demonstrate your understanding of your microprocessor by adding instructions to your instruction set. Add the AND, ZERO, SUB, and STORE ACC instructions to your microprocessor circuit.

- The AND instruction should perform a bit-wise AND operation of the value in the accumulator with an operand from the RAM and store the result in the accumulator.
- The ZERO instruction should store the hex digit 0 in the accumulator by subtracting the accumulator from itself and storing the result back in the accumulator.

- The SUB instruction gets the next value in the RAM and subtracts it from the accumulator by forming its two's complement, adding the result to the value already in the accumulator, and then storing this result back into the accumulator.
- The STORE ACC instruction should write the contents of the ACC into a location in RAM specified by the instruction's operand.

For each instruction above, define the control signals for all the microinstructions by modifying the contents of the ROM. Update the contents of the Instruction ROM in the controller with the added instruction definitions. Then write and execute programs that test each of the instructions**.** As long as you clearly document what you've done, you may test multiple instructions with each test. You'll need to paste the waves for each of your simulations into your template along with an explanation of which instructions each simulation tests. Note that different tests may require that the values in the RAM be modified. If running in Verilog, remember that changes to the ram_vals.txt file do NOT require you to recompile the design.

To be absolutely clear, the AND instruction will be the third instruction and will therefore start in the ROM at address C, or the 12th entry. ZERO will then start at hex address 10, or the 16th entry.

You have three choices as to how to update the ROM contents. The first choice is to make the changes in Digital manually. Open the properties for the controller and choose Open Circuit and then open the ROM properties and click Edit. Then, enter the values you want. Since these are hex values, the format is to put 0x in front of the entry you need in each spot. For example, if you want the hex value 1205, then enter 0x1205. You can then run simulations in Digital to see if your coding is correct. Once you've made the changes, click OK. If you'd like, you can then save the changes and they'll be written to the rom_vals.hex file. You'll still have to simulate in Verilog, so save the file and then export it again. Remember to edit microprocessor.v to add the readmemh statement for the RAM and to update the ram_vals.txt file as necessary. **Warning: when you save the updated ROM file, the comments in the file will be lost!**

The second choice is to update the rom_vals.hex file directly. Simply open it an editor and replace the 0 values with the values you need. When you start a simulation in Digital, the changes you've made will be automatically read. When you are satisfied, export to Verilog and proceed as described above for the first choice. Since you are editing the ROM file, the comments will not be lost if you use this method.

If you are using choice 1 or choice 2, you'll need to include a screenshot of the final version of rom_vals.hex in your template.

The third choice is to edit the Verilog file directly. Digital takes the ROM contents and hardcodes it in the Verilog fie. Open microprocessor.v and look for the module shown in Figure 21.

```
module DIG_ROM_64X14_microcodeROM (
    input [5:0] A,
    input sel,
    output reg [13:0] D
);
    reg [13:0] my_rom [0:31];

    always @ (*) begin
        if (~sel)
            D = 14'hz;
        else if (A > 6'h1f)
            D = 14'h0;
        else
            D = my_rom[A];
    end

    initial begin
        my_rom[0] = 14'h1205;
        my_rom[1] = 14'h234;
        my_rom[2] = 14'h0;
        my_rom[3] = 14'h0;
        my_rom[4] = 14'h1205;
        my_rom[5] = 14'h294;
        my_rom[6] = 14'h0;
        my_rom[7] = 14'h0;
        my_rom[8] = 14'h1205;
        my_rom[9] = 14'h1000;
        my_rom[10] = 14'h0;
        my_rom[11] = 14'h0;
        my_rom[12] = 14'h1205;
        my_rom[13] = 14'h0;
        my_rom[14] = 14'h0;
        my_rom[15] = 14'h0;
        my_rom[16] = 14'h1205;
        my_rom[17] = 14'h0;
        my_rom[18] = 14'h0;
        my_rom[19] = 14'h0;
        my_rom[20] = 14'h1205;
        my_rom[21] = 14'h0;
        my_rom[22] = 14'h0;
        my_rom[23] = 14'h0;
        my_rom[24] = 14'h1205;
        my_rom[25] = 14'h0;
        my_rom[26] = 14'h0;
        my_rom[27] = 14'h0;
        my_rom[28] = 14'h1205;
        my_rom[29] = 14'h0;
        my_rom[30] = 14'h0;
        my_rom[31] = 14'h1fff;
    end
endmodule
```

Figure 21. The ROM in microprocessor.v.

Now, you can directly edit the ROM entries. For example, the AND instruction will start at my_rom[12]. Simply replace the 0s with the appropriate values, recompile and rerun the simulation. If you use this method, be sure to take a screenshot of the above from your design and paste it into your template once you are done with your design. (If you are doing the extra credit, wait to take this screen shot until after Task 4-6.)

Once you are convinced that your circuit is working properly, take a screenshot of your timing diagrams and copy them to your lab template. Include the programs from ram_vals.txt in your lab template. Also, comment on any issues that you encountered.

If you are not doing the extra credit, also include a screenshot of your ROM contents either from microprocessor.v or rom_vals.hex.

Task 4-6: Invent Your Own Instruction (Extra Credit)

The microprocessor you've built can actually do many more operations than what we've done so far. To earn extra credit, come up with a function we haven't done yet and implement an instruction to perform that function. Remember that you need to limit the function to no more than 4 steps. Simulate your instruction in Verilog and include the program your ran along with a screenshot of your waves in your template. And, now that you've added this last instruction, take a screenshot of your ROM contents, either from microprocessor.v or from rom_vals.hex and paste it into your template.

Task 4-7: Create a video and submit your report.

Take a screenshot of your final microprocessor design in Digital and paste it into your template.

Create a video showing your schematic in Digital and one of your waveforms and explain how your design works. (This can be the waveform for any simulation other than the one you were provided with!) Be sure to show yourself in the video and show your screen. Upload your video to your google drive. Be sure to set permissions so that everybody can see your video and paste the link into your template.

Make sure all of your files are in the Lab4 directory. Create a zip file of the Lab4 directory. Turn in the zip file and your completed template. (Double check that you turned in the completed template and NOT the blank one you downloaded. Unfortunately, turning in the wrong template file is a common mistake.)

Congratulations! You've completed Lab 4!

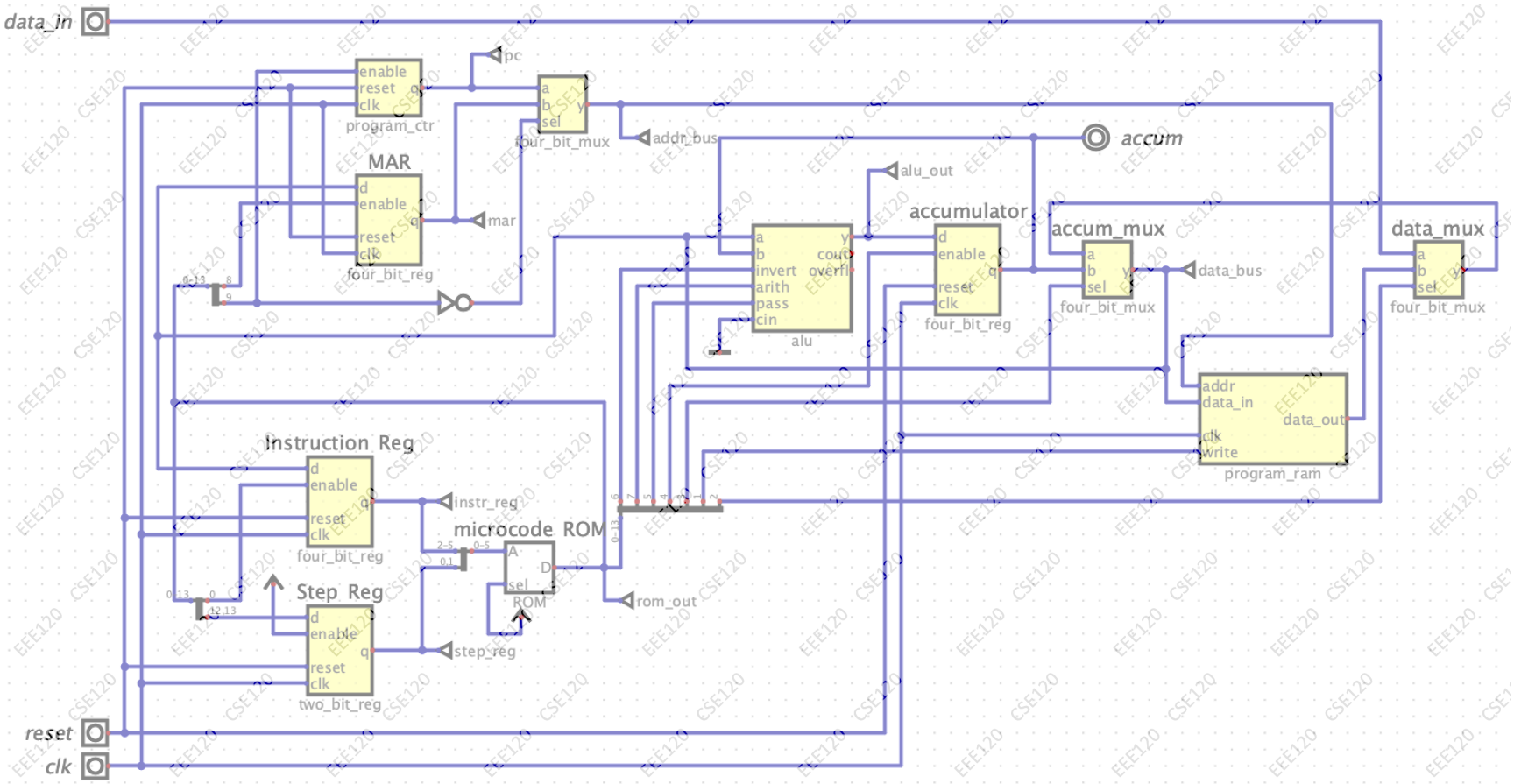

Figure 21. The microprocessor with a layer of hierarchy removed.