Lab 3: Registers, Counters, and the "Brainless CPU"

Prerequisites: Before beginning this lab, you must:

- Understand how to use the tool flow (See the installation guide and Lab 0)
- Describe the operation of an edge-triggered D flip flop.
- Identify the building blocks of a synchronous counter.
- Have completed Lab 1: Half Adder, Full Adder, 4-bit Incrementer and Adder.
- Have completed Lab 2: Multiplexers, Decoders and the Arithmetic Logic Unit.

Equipment: Personal computer with the required software installed.

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

alu.dig
and_add.dig
four_bit_adder.dig
four_bit_mux.dig
full_adder.dig
half_adder.dig
incrementer.dig
not_neg.dig
two_bit_mux.dig

Files to download:

four_bit_reg_top.v
four_bit_reg_stim.v
ram_vals.hex
brainless_top.v
brainless_stim.v

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

- Build and test a 4-bit D register.
- Build, test and implement a 4-bit up counter.
- Build a 16-word, 4-bit RAM.
- Build and simulate a microprocessor that is absent a controller.
- Act as the controller for an elementary microprocessor.

Introduction

In Labs 1and 2, we built almost all of the combinational logic circuits we'll need for our microprocessor. In this lab, our aim is to build a few more circuits that we'll need to build a complete microprocessor. First, we will solve the problem of not being able to store any data by building a 4-bit parallel D register that consists of four individual D flip-flops wired to a common bus. We will then use this memory block to build a counter along with the incrementer we built in Lab 1. Then we will look at another memory circuit, the RAM, that will be used in our microprocessor to store the data for our CPU. In the last part of this lab (careful, this will take quite some time!!!), we will assemble our microprocessor and call it the “brainless microprocessor”. You will act as the “brains” (or controller) for this processor and manually manipulate the signals controlling the ALU, registers, decoders, and memory, to cause the microprocessor to perform a series of operations we will eventually call an instruction, and a series of instructions we will eventually call a program.

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 Lab3. Into that folder, copy the files listed above from Lab 2. Be sure to only copy the required files. While it is tempting to do Lab 3 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 Lab 3 folder will only have the files necessary for Lab 3.

Once you’ve copied the files from Lab 2, download the files provided for Lab 3 and place them in the Lab3 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 3-1: Build and Test a 4-Bit D Register with Enable

Because our microprocessor should operate on 4-bit numbers, we will have to build a 4-bit storage circuit. Equipped with what you know about D flip-flops, you can now construct a **4-bit parallel D register**. The 4-bit parallel register should have 4-bit input and output buses, similar to those you used for the 4-bit full adder in Lab 1. The clock input, however, remains a 1-bit input. It also makes sense to include an **enable** input that allows you to control if the flip-flops should change value when they encounter a clock edge or remain at their current value.

How does the enable work? Consider the circuit shown in Figure 1. (You will not be building this circuit, it is being used to illustrate how the flip-flop with enable works.) Consider what will happen if there is a rising edge on the clock but **enable**=0. In this case, the mux will select the current value in the flip-flop and, therefore, the flip-flop’s contents will not change. On the other hand, if **enable**=1, the mux will select the **d** input and the value on **d** will be loaded into the flip-flop. Using an enable like this is very powerful as it allows us to control whether or not we should load a new value into the flip-flop or maintain the current value. We’ll need this capability so that our microprocessor can control which registers are to be loaded during any particular clock cycle.

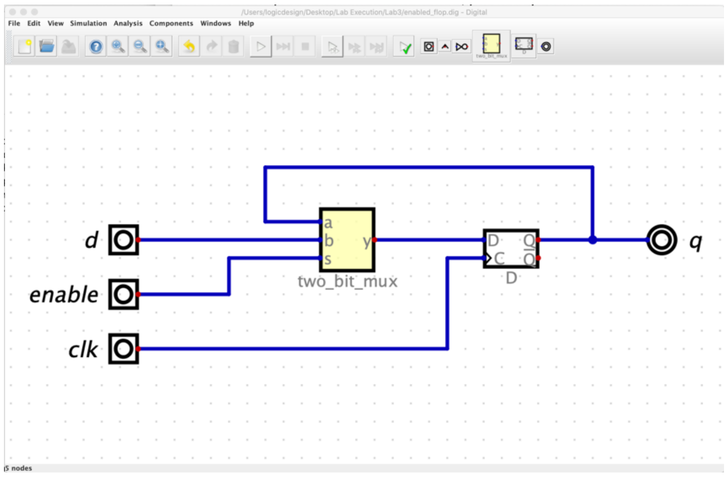


Figure 1. Adding an enable to a flip-flop.

Now let’s build a 4-bit register with an enable as shown in Figure 2.

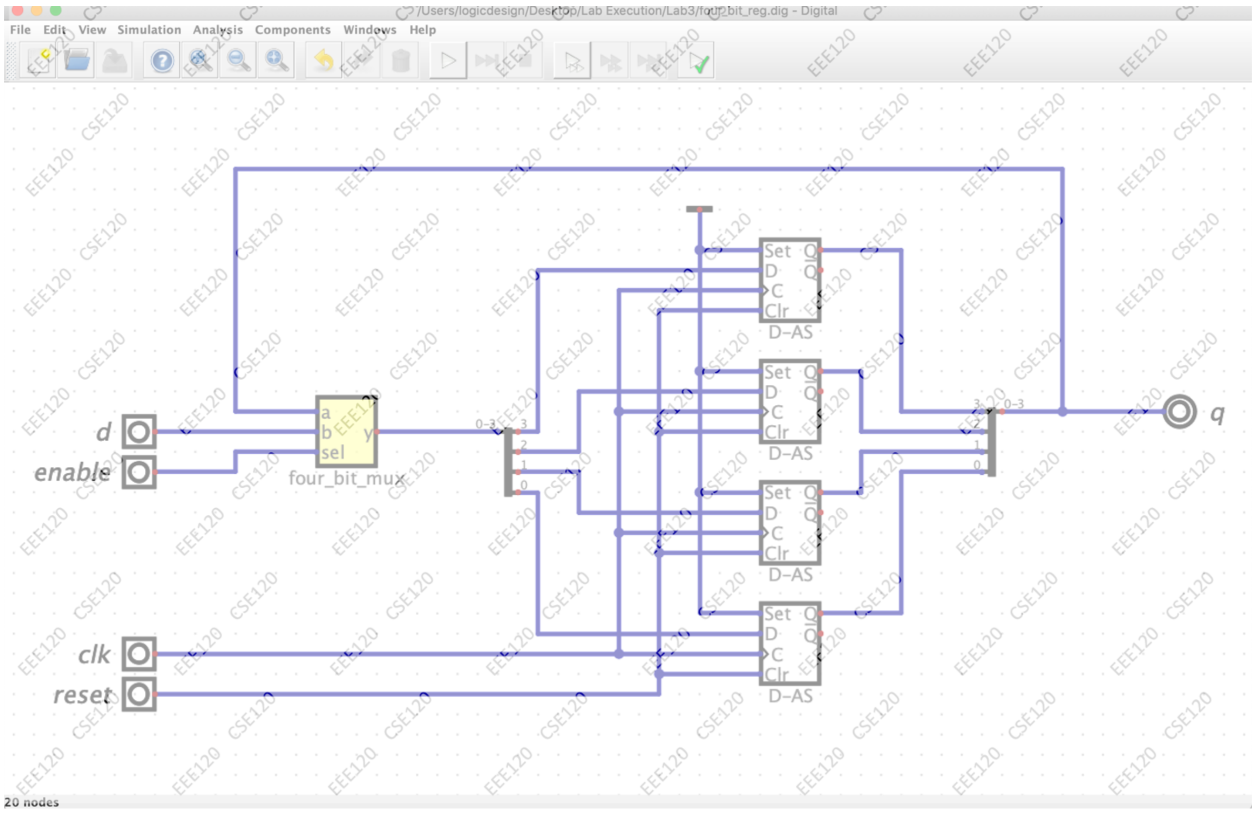

/Users/logicdesign/Desktop/Lab Execution/Lab3/four_bit_reg.dig - Digital
File Edit View Simulation Analysis Components Windows Help
d
enable
clk
reset
a
b
sel
y
four_bit_mux
Set
D
C
Clr
Q
D-AS
0-3
q
20 nodes

Figure 2. A 4-bit register with enable and reset.

The first step is to add the flip-flops. Select File->New and then File->Save As and name the new circuit four_bit_reg. Digital will automatically append the .dig extension. Make sure it is saved in the Lab3 folder.

To find the flip-flop, Select->Components->Flip-Flops->D-Flip-flop, asynchronous and place it on your schematic. Note that there are 4 inputs: **Set**, **D**, **C**, and **Clr**. If **Set** is 1, it forces **Q** to a 1. If **Clr** is 1, it forces **Q** to 0. **C** is the clock and **D** is the data input.

We will use the **Clr** input and will not use the **Set** input. Therefore, we need to disable the **Set** input by connecting it to a logic 0. We do this by connecting it to ground. Find this under Components->Wires->Ground.

You are now ready to complete building the circuit. Make sure you have copied the .dig files from Lab 1 so you can insert the four-bit 2-to-1 mux, four_bit_mux. Create the splitter/mergers in the same manner as in your prior labs and make sure the **d** input and **q** output are 4 bits wide.

When done building your schematic, save it. Now, simulate it in Digital. Note that you'll have to click the **Clk** input twice to complete a clock period. When you are satisfied that it is working, save it, then select File->Export->Export to Verilog and save it as four_bit_reg.v. Make sure it is exported to the Lab3 folder! Let's now simulate the register in Verilog.

You have been given the files four_bit_reg_stim.v and four_bit_reg_top.v. Make sure they have been downloaded to the Lab3 folder. You will modify four_bit_reg_stim.v to run up to 8 tests on your circuit. You'll need to decide which tests to run. Make sure you only modify the file below the line which contains:

// IMPORTANT: ONLY MODIFY BELOW THIS LINE ***

The format of the data in the file is shown in Table 1. The clock signal will be automatically generated for you and, therefore, is not included in the stimulus. The values will be applied on the falling edge of the clock. Since the flip-flops are positive-edge triggered, **enable** and **d** will be sampled on the rising edge of the clock. However, **reset** will have an effect immediately. The clock period is 10 ns and the output will be checked 4 ns after the rising edge of the clock.

Table 1: Bit definitions for four_bit_mux_stim.txt.

Bit #	11:8	7:6	5	4	3:0
Meaning	exp_q	unused	reset	enable	d

The stimulus provided in four_bit_reg_stim.v resets the circuit, then a couple of clocks later loads 0xF, then two tests later, applies reset again. Feel free to change the file as you see fit to properly test your circuit. Figure 3 shows how the waves appear for the provided stimulus. To simulate the design, execute:

```
iverilog -o four_bit_reg.exe four_bit_reg.v four_bit_reg_top.v four_bit_reg_stim.v
On Mac: ./four_bit_reg.exe
On Windows: vvp four_bit_reg.exe
```

To view the waves on Mac, open GTKWave and then CMD-T to open the four_bit_reg_waves.vcd file. On Windows, type gtkwave four_bit_reg_waves.vcd.

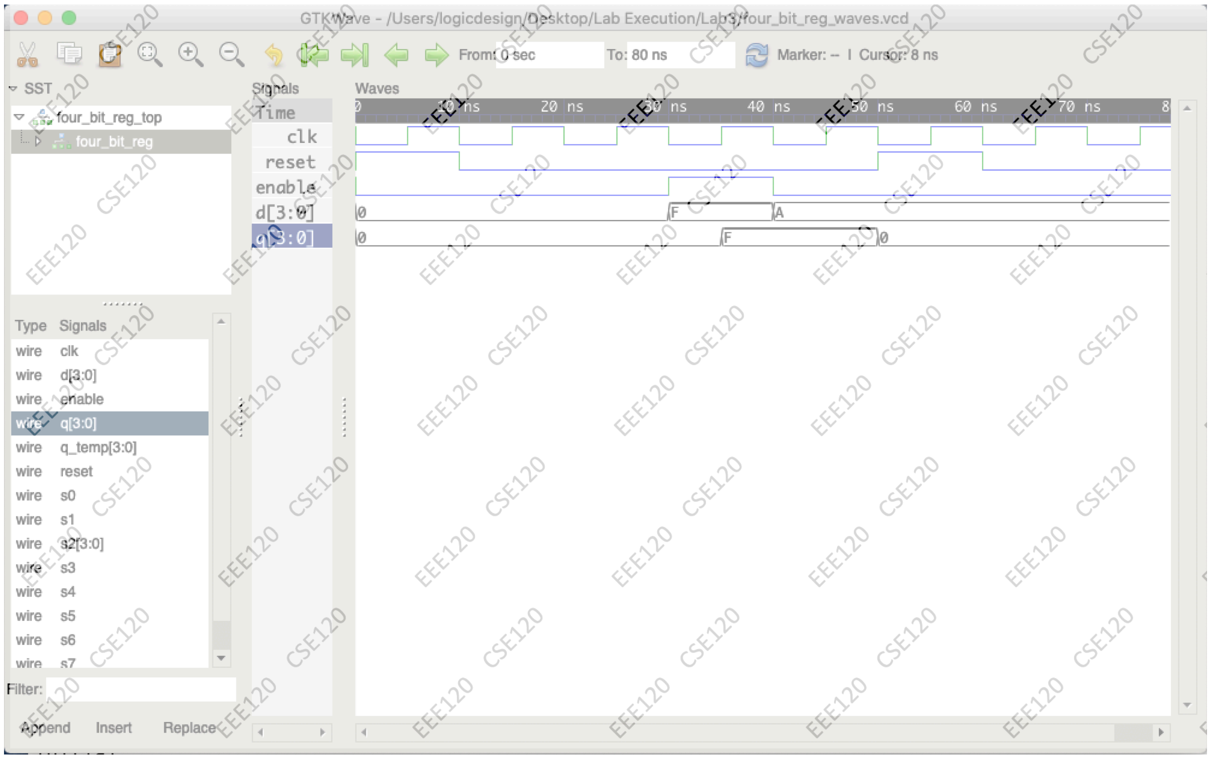


Figure 3. Example waves for the four-bit register.

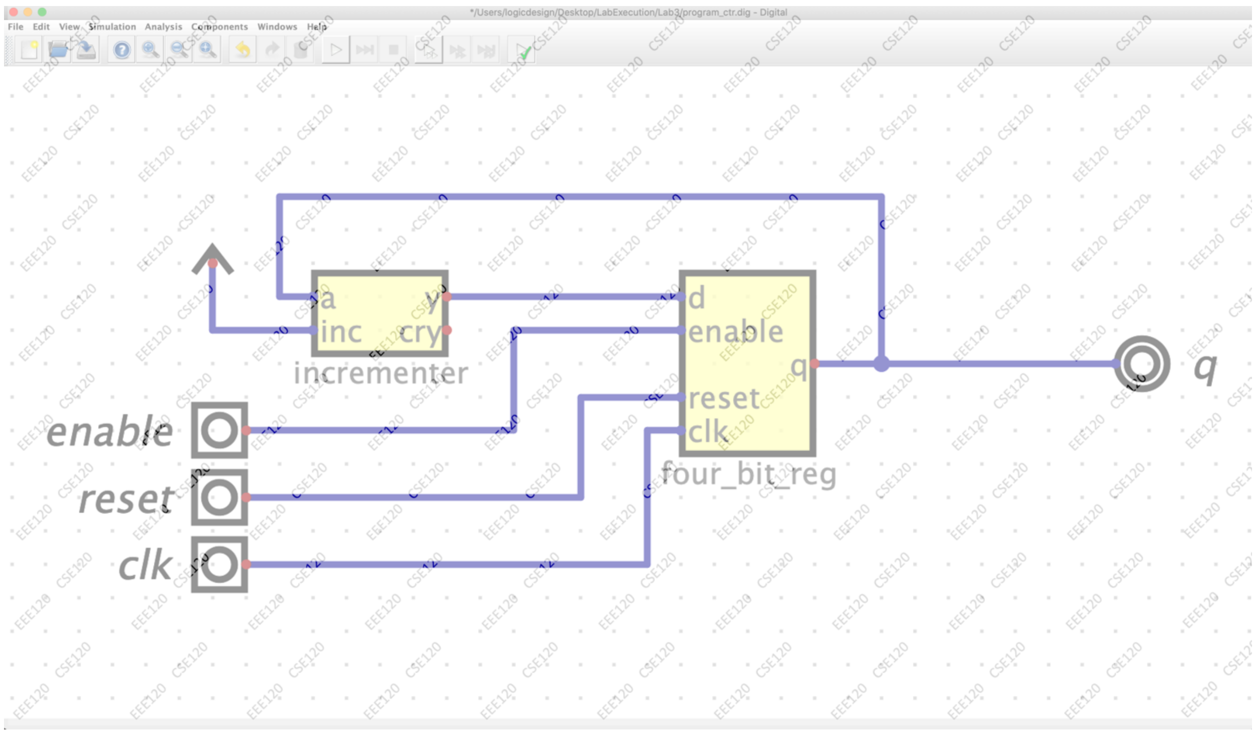
Once you are satisfied, take screenshots of your circuit in Digital and the waveforms in GTKWave and paste them in the template. Also comment on any issues that you encountered.

Task 3-2: Build and Test a 4-Bit Program Counter

Now that we have a 4-bit storage device, we can move on to actually build something that we could not build before: a counter. Let's keep it simple and construct a counter that counts up from zero to 15 (Hex F), then returns to zero and keeps going, incrementing with every positive-going clock pulse. Because we have included the **enable** signal in our 4-bit D register, we can use this input to stop the counter at its current value. You can accomplish the "plus one" function by using the 4-bit incrementer that you built in Lab 1.

In Digital, select File->New. Then select File->Save As and save the new file as program_ctr, again making sure it is saved in the Lab3 folder. We will use this circuit as the program counter for our microprocessor, hence the name. In Lab 4, you'll learn why this is called the program counter. You should be able to see the incrementer under Components->Custom if you copied it and the half adder from Lab 1. Go ahead and build the circuit shown in Figure 4. Set the width of the incrementer circuit to 4 by opening its properties, clicking on Open Circuit, and then selecting Edit->Circuit specific settings. Do the same for the four_bit_reg circuit. You'll connect the **inc** input of the incrementer to logic 1 using Components->Wires->Supply voltage. Note that the **cry** output is not connected. Remember that the **q** output is 4 bits wide.

Finally, reorder the inputs of the 4-bit register so that **reset** is above **clk**, by opening its properties, opening the circuit, and selecting Edit->Order Inputs.



*/Users/logicdesign/Desktop/LabExecution/Lab3/program_ctr.dig - Digital
File Edit View Simulation Analysis Components Windows Help
a
inc
y
cry
incrementer
d
enable
reset
clk
q
four_bit_reg
enable
reset
clk
q

Figure 4. Circuit schematic for the program counter.

Note the extra space that separates the symbols for the inputs and the output from the splitters. When we simulate in Digital, that extra space will allow the value on the bus to be seen.

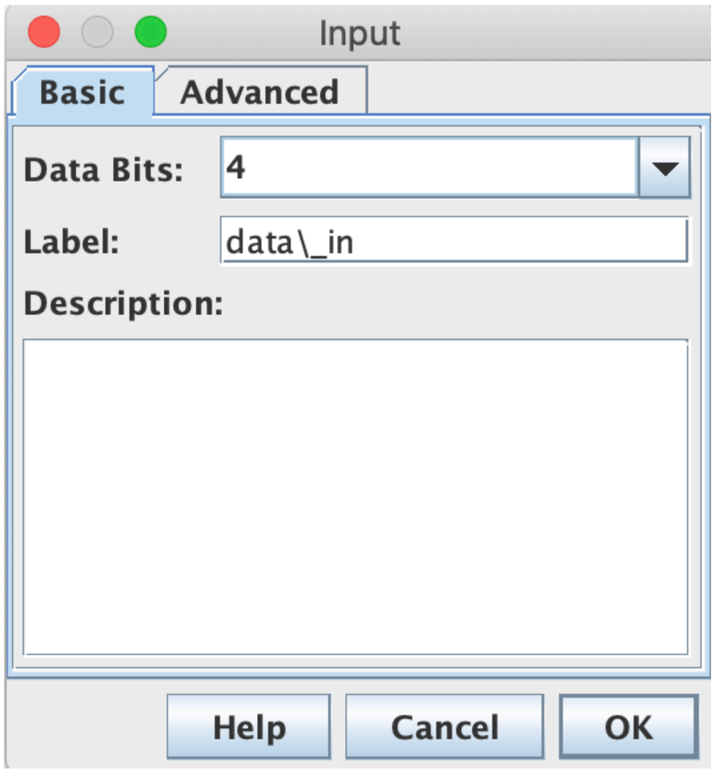
Verify that the circuit is working by simulating it Digital. Make sure it increments when **enable**=1 but holds its value when **enable**=0 even if the clock is going up and down. Also verify that setting **reset**=1 will cause **q** to be all zeros. When the circuit is working properly, take a screenshot of the schematic. (Be sure to stop the simulation first!) Then paste the screenshot into your template. And make sure you save the circuit.

Task 3-3: Create a 4-Bit RAM with 16 4-Bit Words

Our microprocessor will need a memory into which we will place our instructions and data. In this task, we will instantiate a Random Access Memory (RAM) in Digital. A memory consists of some number of words where each word consists of some number of bits. In our case, the memory will have 16 words and each word will be made up of 4 bits. We select which word to read or write based on an address. The number of bits in the address has to be sufficient to select each of the words in the memory. Since our memory has 16 words, our address will need 4 bits. It is called a RAM because we can use an address to select any word at any time, that is, we can make random accesses rather than having to access the words in order. This is important since we'll want our microprocessor to determine which address to use during each clock cycle. Finally, there will be an enable signal, often called a write signal or a write strobe, which determines whether we are reading or writing the word at the current address. The RAM we'll use also has an input that enables the output. We will tie that high so that the output is always driven.

Open Digital, if it isn't already open, and select File->New and then File->Save As and name the design program_ram. Then add the RAM: Components->Memory->RAM->RAM, separated Ports. We'll then need to add 3 inputs: **addr**, **data_in**, and **clk**. Note that **addr** and **data_in** are 4 bits. We'll then add one output, **data_out**, which is also 4 bits.

IMPORTANT: Because of the way Digital maintains strings, you need to escape the underscore. When creating the label for data_in, type data_in. If you don't, then the formatting will assume you wanted the "in" to be a subscript and you'll get data$_{in}$ when we want data_in. You'll need to do the same for data_out! See Figure 5 for an example.



Data Bits: 4

Label: data_in

Description:

Figure 5. Escaping the underscore.

We'll not use the **ld** input, so connect it to Components->Wires->Supply voltage. Connect the rest of the circuit as shown in Figure 6. Leave some space between **addr** and **data_in** as shown.

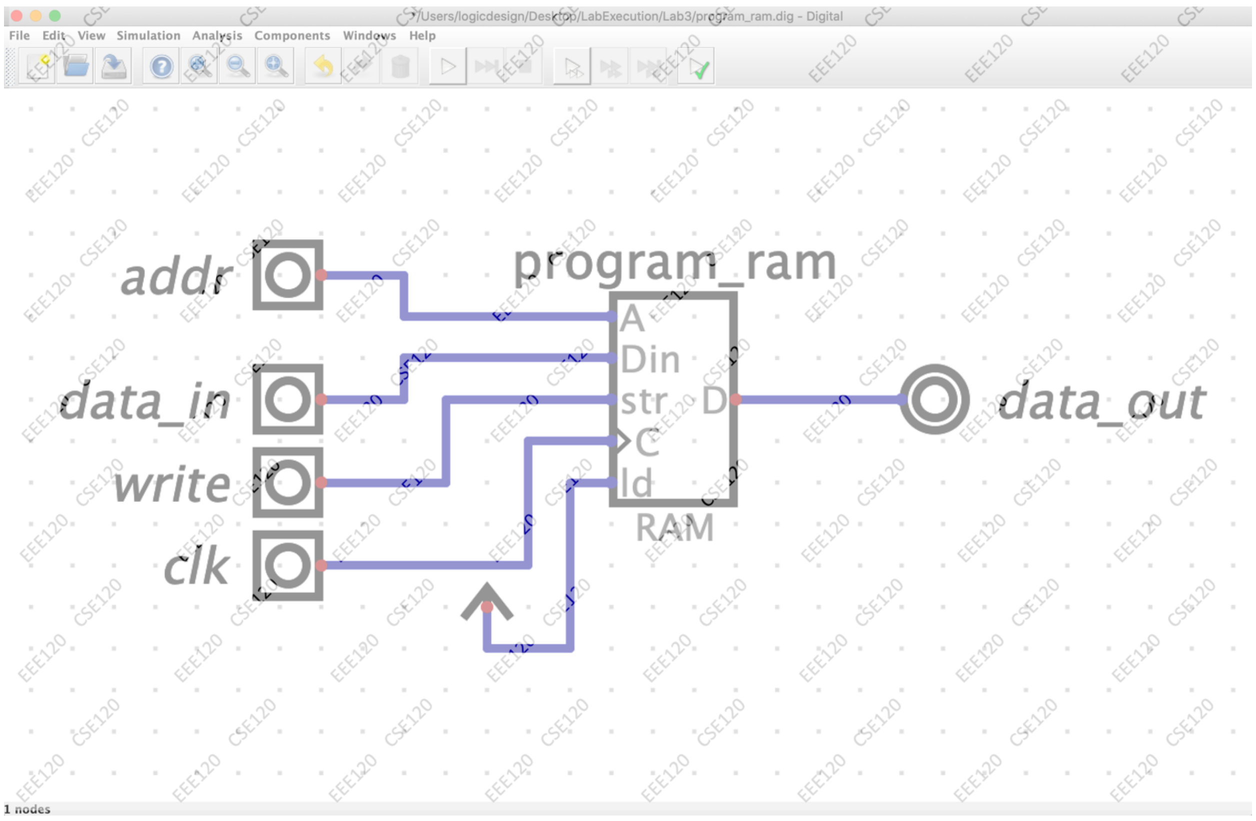
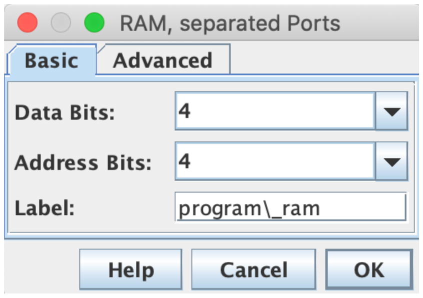

/Users/logicdesign/Desktop/LabExecution/Lab3/program_ram.dig - Digital
File Edit View Simulation Analysis Components Windows Help
addr
data_in
write
clk
program_ram
A
Din
str
C
ld
D
RAM
data_out
1 nodes

Figure 6. The program RAM.

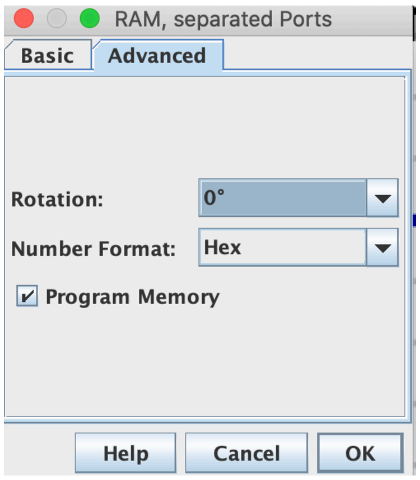
Now we have to set up the program RAM so it will work in our microprocessor. Open the RAM's properties. The Data Bits field determines the number of bits per word. Set this to 4. The Address Bits field determines the number of words in the memory. Set this to 4 as well. Finally, add a label since one is required. We'll call it program_ram. See Figure 7 to see how this should look.



Data Bits:	4
Address Bits:	4
Label:	program_ram

Figure 7. RAM set up.

Now click on the Advanced tab and click the Program Memory check box. Then select Hex for the Number Format. This is shown in Figure 8. Then click OK.


RAM, separated Ports

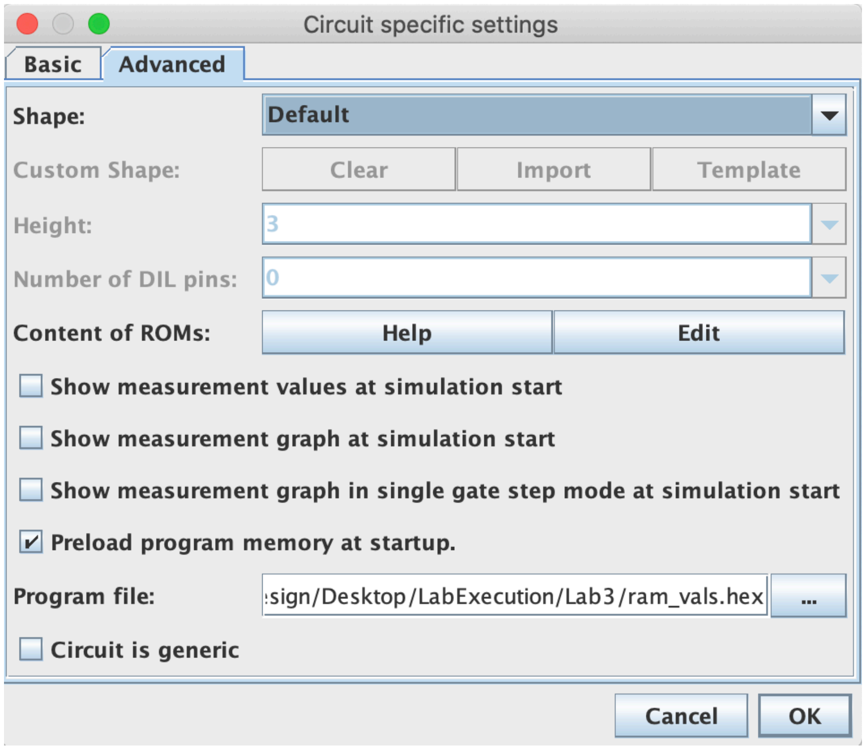
Rotation: 0°

Number Format: Hex

☑ Program Memory

Figure 8. RAM Advanced tab.

We have one more step to complete setting up the RAM: we want to make it so a file is read to initialize the contents of the RAM. **Make sure the file ram_vals.hex is in your Lab3 directory.** Select Edit->Circuit specific settings. When the window pops up, click on the Advanced tab. Click the check box next to “Preload program memory at startup”. Click on the 3 dots on the right side across from “Program file:” and navigate to your Lab3 folder and select ram_vals.hex. Finally, uncheck the box at the bottom so that the circuit is not generic. The window should appear as shown in Figure 9. Then click OK.


Circuit specific settings

Shape: Default

Custom Shape: Clear | Import | Template

Height: 3

Number of DIL pins: 0

Content of ROMs: Help | Edit

☐ Show measurement values at simulation start

☐ Show measurement graph at simulation start

☐ Show measurement graph in single gate step mode at simulation start

☑ Preload program memory at startup.

Program file: :sign/Desktop/LabExecution/Lab3/ram_vals.hex ...

☐ Circuit is generic

Figure 9. Specifying the file which holds the initial RAM contents.

Save your design and then start a simulation. The first line in the ram_vals.hex file is "v2.0 raw" and is required so that Digital knows how to read the file. The rest of ram_vals.hex is the contents we want to load into the RAM as initial values. Each line contains a hex character which is a 4-bit word for the memory. The next two lines are 3 and 5. So 3 will be loaded at address 0 in the RAM, 5 in address 1, and so on. The remainder of the file will load the rest of the RAM with 0s.

In the Digital simulation, increment **addr** and satisfy yourself that the output responds correctly. Then set **data_in** to a nonzero value, **addr** to a value other than 0 or 1, and set **write** to 1. Now, click on clk and verify that **data_out** now reflects the value on **data_in**. Change **data_in** and, after clicking **clk** twice more, the output should again match **data_in**. During the simulation, if you open the properties of the RAM, you will see its contents. When you are satisfied that your circuit is working correctly, save it and take a screenshot to paste in your template.

Task 3-4: Build and Test the Brainless Central Processing Unit

You are finally ready to assemble the actual CPU! You can do it by combining circuits that you built in previous labs. The schematic of the brainless microprocessor is shown in Figure 10. Do not panic when you look at this circuit! You already have everything you need to wire it up. It is really not too hard to build, and it has some very promising capabilities.

The left side of the diagram contains the ALU you built in Task 2-6, feeding into a 4-bit D register you built in Task 3-1, and the bottom right contains the sixteen-word 4-bit RAM you built in Task 3-3. All is being tied together by several 4-bit multiplexers that you built in Task 2-2, which switch between their inputs.

One of the key changes is the addition of a register at the output of the ALU, called the "Accumulator". The presence of the accumulator register (controlled by the clock signal) in Figure 10 transforms the circuit we have been building in a fundamental way: it transforms it from a combinational logic circuit into a synchronous logic circuit.

We will use a 4-bit 2:1 multiplexer to route the RAM output to the data bus only when **read** line is high. The **read** line, which you will control in this exercise, will be high when we want to read a value from memory. If you want to write to the memory, you need to enable the RAM by setting the **write** line high. In either case, you need to select which memory word you want to read from or write to. You accomplish that by setting a 4-bit address on the address bus, **addr_bus**.

We need to connect the RAM values to the RAM at this level as well. Repeat the steps from Task 3-3, starting with Edit->Circuit specific settings, as shown in Figure 9.

The circuit in Figure 10 is almost a microprocessor; it is only missing the 'brains', or more frequently called the controller or control circuitry. In Lab 4 you will build the controller for your microprocessor. For now, the objective is for you to get familiar with how the microprocessor is controlled by having you act as the controller. This way, the controller design will make more sense. You will function as the 'brains' of the microprocessor, by manipulating the control lines to get it to process the data.

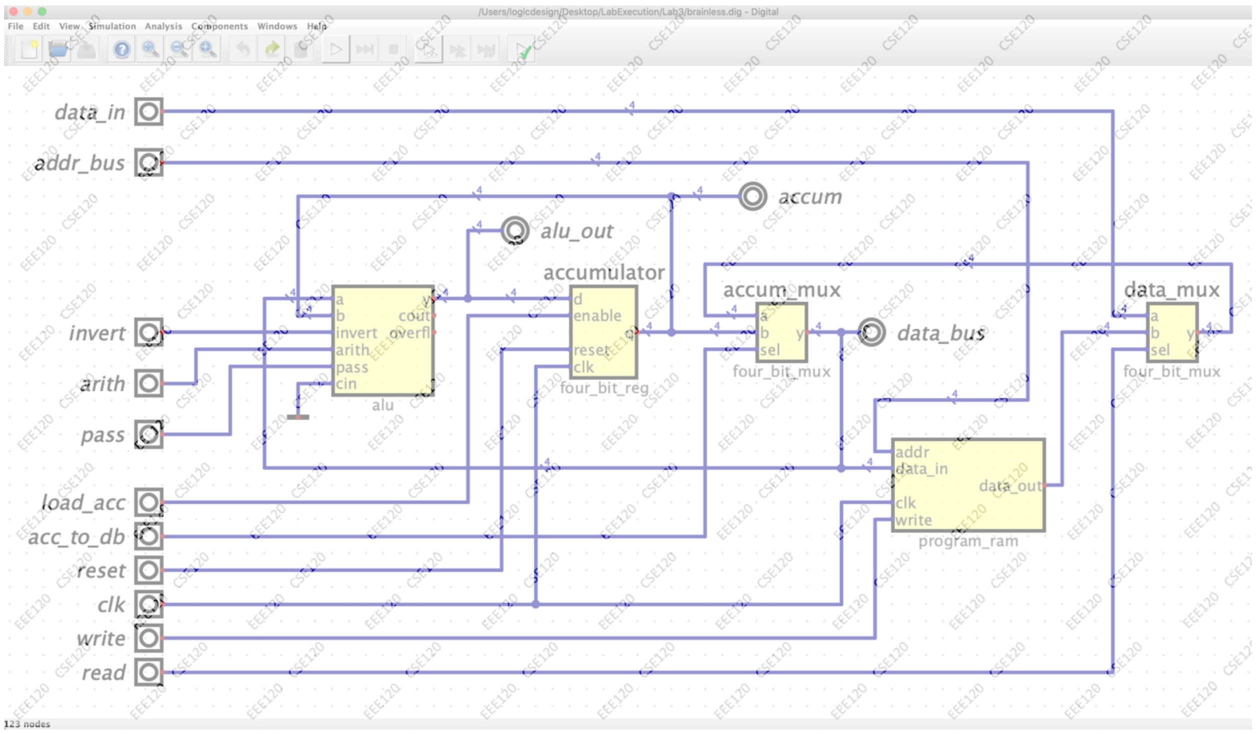


Figure 10. The brainless CPU.

Select File->New and File->Save As and call the design brainless. (It's called a brainless CPU because the controller circuit, or the brains of the processor, haven't been added yet.) Now connect the circuit as shown in Figure 10 noting which inputs and outputs are 4 bits wide. Make sure that the files from Lab 2 are already in the Lab3 folder. If they aren't, copy them now. If you try to load them into Digital via Components->Custom and don't see them, select Components->Custom->Update to get Digital to rescan the folder and check again. If you still don't see them, make sure they are in the folder.

NOTE: Do not copy the files from Lab 1 since the updates we made to the widths of the symbols in Lab 2 won't be present in the versions from Lab 1.

NOTE: Remember to use the backslash in front of underscores when labeling inputs and outputs like this acc_to_db so they appear without subscripts.

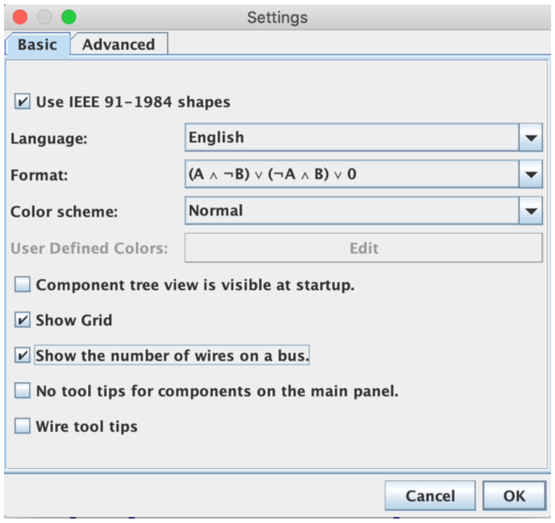
When you place the ALU, make sure you update its width to 6 by opening its properties, clicking Open Circuit, and then Edit->Circuit specific settings. In addition, we want to change the order of the inputs of the ALU so, from top to bottom, they are **a**, **b**, **invert**, **arith**, **pass**, **cin**. Do this by selecting Edit->Order Inputs and moving the inputs as needed. When you place the program_ram, set its width to 9. When these changes have been done, go ahead and add the wires.

On the ALU, **cin** is connected to ground and **cout** and **overfl** are not connected.

You will also have to specify the file to be preloaded into the RAM for the brainless design in the same manner you did for the program_ram. Do this by selecting Edit->Circuit specific settings and making the changes as shown in Figure 9.

Finally, notice that I've added labels to some of the subcircuits: accumulator, accum_mux, and data_mux. These can be added via the properties window. Again, remember the \ before the _. While not required, it is helpful to have them so we can reference them in the description below.

Looking at the schematic, it can be confusing when differentiating between a wire, which represents 1 bit, and a bus, which represents many bits. To fix this, select Edit->Settings and check the box "Show the number of wires on a bus." This is shown in Figure 11. You may have to click in the window to get it to add the annotations to the buses. Note that 1-bit wires are not annotated.



- [✓] Use IEEE 91-1984 shapes
- Language: English
- Format: (A ∧ ¬B) ∨ (¬A ∧ B) ∨ 0
- Color scheme: Normal
- User Defined Colors: Edit
- [ ] Component tree view is visible at startup.
- [✓] Show Grid
- [✓] Show the number of wires on a bus.
- [ ] No tool tips for components on the main panel.
- [ ] Wire tool tips

Figure 11. Show the number of bits in buses.

Let's conduct one test in Digital to validate the design's functionality; let's add two numbers, 3 and 5, and observe the sum, 8, in the accumulator. Since the RAM has been preloaded with the values 3 and 5, we just need to get those values to the correct places when we need them there.

First step is to get the 3 into the accumulator. Start the simulation and all of the inputs will start with the value 0. To get the 3 to the accumulator, we need to get it from the program_ram through the data_mux. To do this, we'll want to set the **read** input to 1. Then we need to get the 3 through the accum_mux, so we'll leave the **acc_to_db** input at 0. To get the 3 through the ALU, we'll set **pass** to 1 and leave the other ALU controls at 0. Now we need to make it so the ALU output will be loaded into the accumulator, so we'll set **load_acc** to 1. Your circuit should now look like the one in Figure 12. Note that Digitial is displaying the values on the wires so you can see that the 3 is on the **d** input of the accumulator.

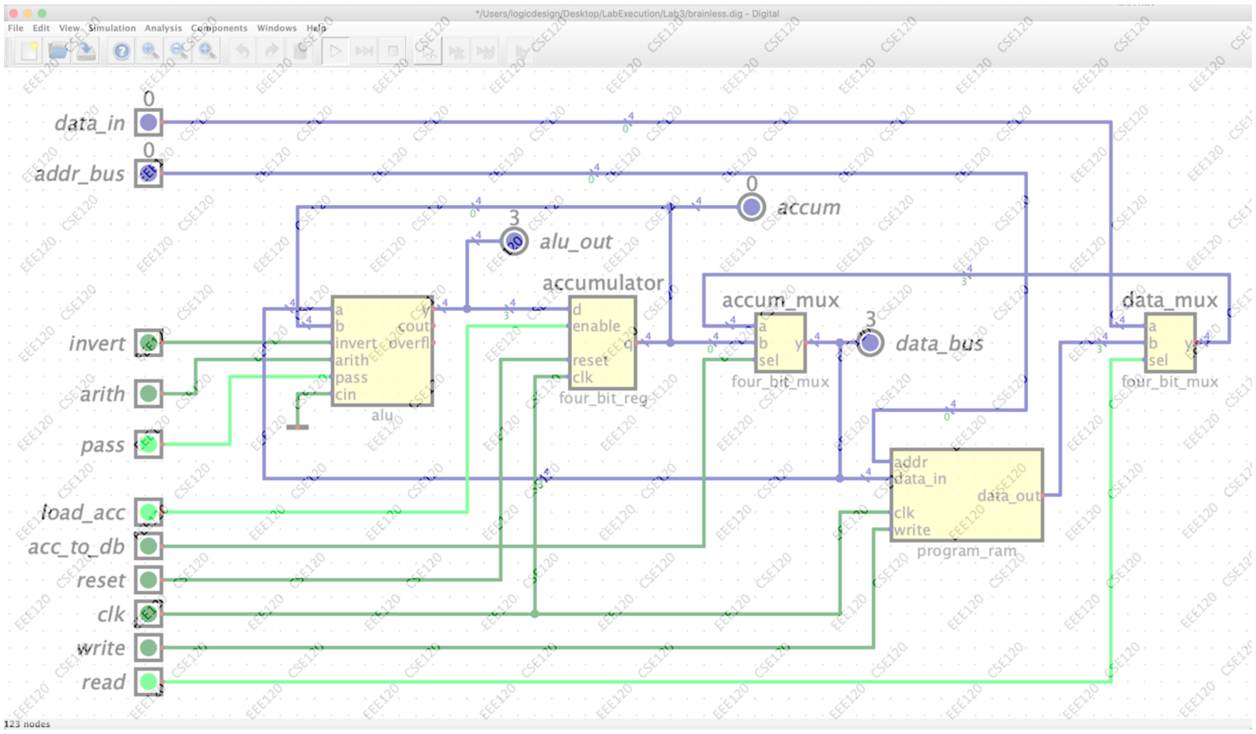


Figure 12. Preparing to load the 3 into the accumulator.

To actually, load the 3, set the **clk** input to 1. You should see the **accum** output change to 3!

Now to add 5, we'll start by incrementing the addr_bus. Click on it and click the up array so it changes to 1 and then click OK. The 5 should now be propagating all the way to the **d** input of the accumulator. But, that isn't what we want. Notice that the accumulator output is connected to the **b** input of the ALU, so we want to perform the addition **a**+**b**. To do this, we need to change the ALU controls. We'll want **arith**=1 and **pass**=0. Make these changes and you'll see that the value 8 is now on the **d** input of the accumulator. Set the **clk** back to 0 and then click it again to set it to 1. The **accum** output should now be 8 as shown in Figure 13.

By the way – notice that the **alu_out** value is now D, or decimal 13. It has that value because we've now loaded 8 into the accumulator. Since the ALU is still programmed to add its inputs, it is now adding 5+8, which is 13.

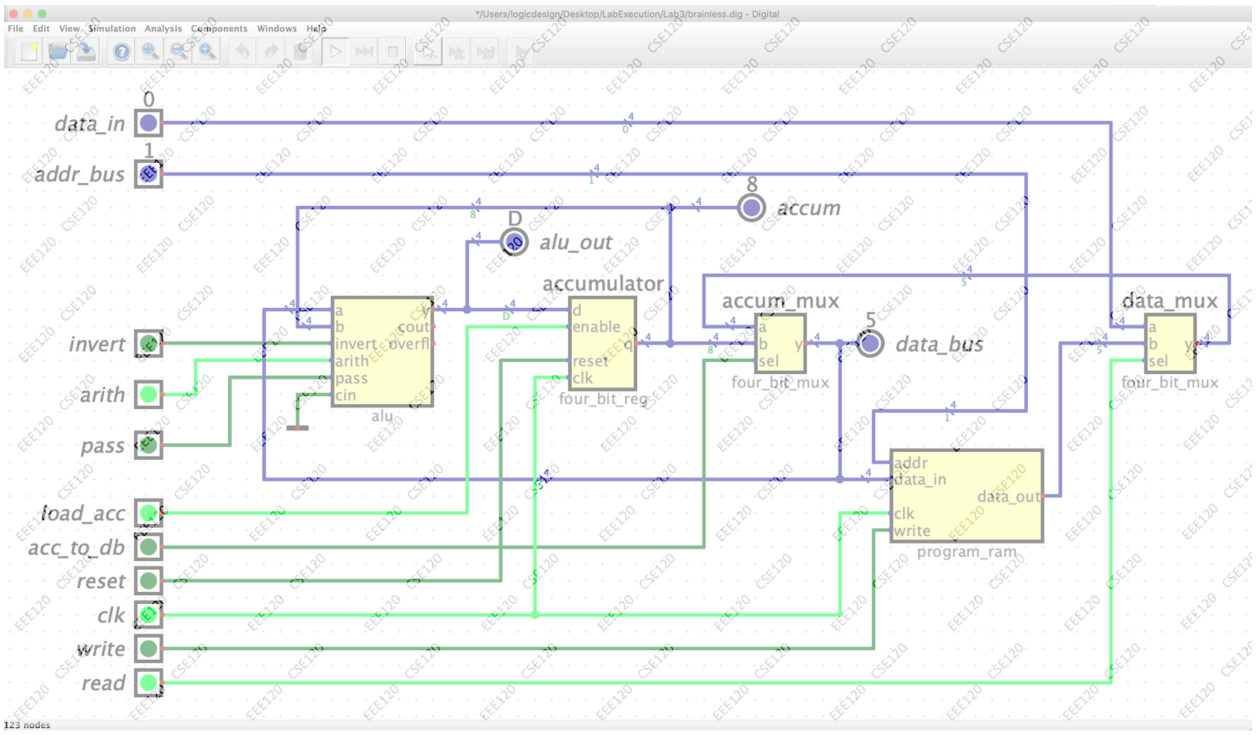


Figure 13. Successfully added 3+5.

Think about the steps we've gone through to achieve this. We figured out what values we needed on the inputs in order to get the data where we needed it and in order to get the various subcircuits to behave the way we needed them to. That's exactly what you'll be doing for the next steps.

Take a screenshot of the Digital simulation window showing that you have successfully added 3+5 similar to that shown in Figure 13 and paste the screenshot into your template. Then end the simulation.

Task 3-5: Simulate the Brainless Central Processing Unit

Let's now do this simulation in Verilog. Select File->Export->Export to Verilog and save the file as brainless.v in the Lab3 folder. Unfortunately, the initialization of the RAM is not included in the export, so we'll do a small edit to the file. Open the brainless.v file in an editor and search for the line:

module DIG_RAMDualPort

There, you will find the definition of the RAM design described in Verilog and it will appear as shown in Figure 14.

```
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 14. The original Verilog for the program_ram.

All you need to do is add the bold lines prior to the endmodule as shown in Figure 15. Open the file in an editor and add the lines. Then save the file and exit the editor. Make sure the editor saves the file as a text file, keeping the same file name: brainless.v. (Your simulation won't work if the file is saved in a format such as normally saved by Word. Most editors have the ability to save as a text file.)

Copy the file ram_vals.hex you used for the Digital simulation to a new file named ram_vals.txt. The only change to make is to delete the top line, v2.0 raw. (Alternatively, you can make it a comment by starting the line with // so that it reads // v2.0 raw. The // is the way a comment is started in Verilog.) The code we have added will allow you to initialize the RAM in a Verilog simulation.

```
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 15. The modification, in bold, required to initialize the RAM in Verilog simulations.

Let's have a look inside the file brainless_stim.v. This file is similar to the stimulus and response files we've used before. Table 2 details the bit positions for the inputs and outputs. Note that there are a total of 28 bits and that they are all used. And, as we've done in the past, the bits are represented in groups of 4 as hex digits. Note that the expected value for test_vals[2] is D! That is because we've added 3+5 and gotten 8, but the ALU inputs haven't changed so the ALU is now computing 8+5. Make sure you understand this before proceeding to the next task. Looking at the waves may help.

Table 2: Bit definitions for brainless_stim.v.

Bit #	27:24	23:20	19:16	15:12	11:8
Meaning	exp_accum	exp_data_bus	exp_alu_out	data_in	addr_bus

Bit #	7	6	5	4	3	2	1	0

Meaning	invert	arith	pass	load_acc	acc_to_db	reset	write	read

Now, we're ready to simulate in Verilog by executing the following commands:

```
iverilog -o brainless.exe brainless.v brainless_top.v brainless_stim.v
On Mac: ./brainless.exe
On Windows: vvp brainless.exe
```

Now you can open GTKWave with the file brainless_waves.vcd. Add the signals as shown in Figure 16. Your waves should match those shown. Note that you can select the subcircuits to check on values there in case your design isn't working as expected. Note that blanks have been added to separate the groups of signals to make things easier to see. Once your simulation matches, take a screenshot of the waves and paste it into your template.

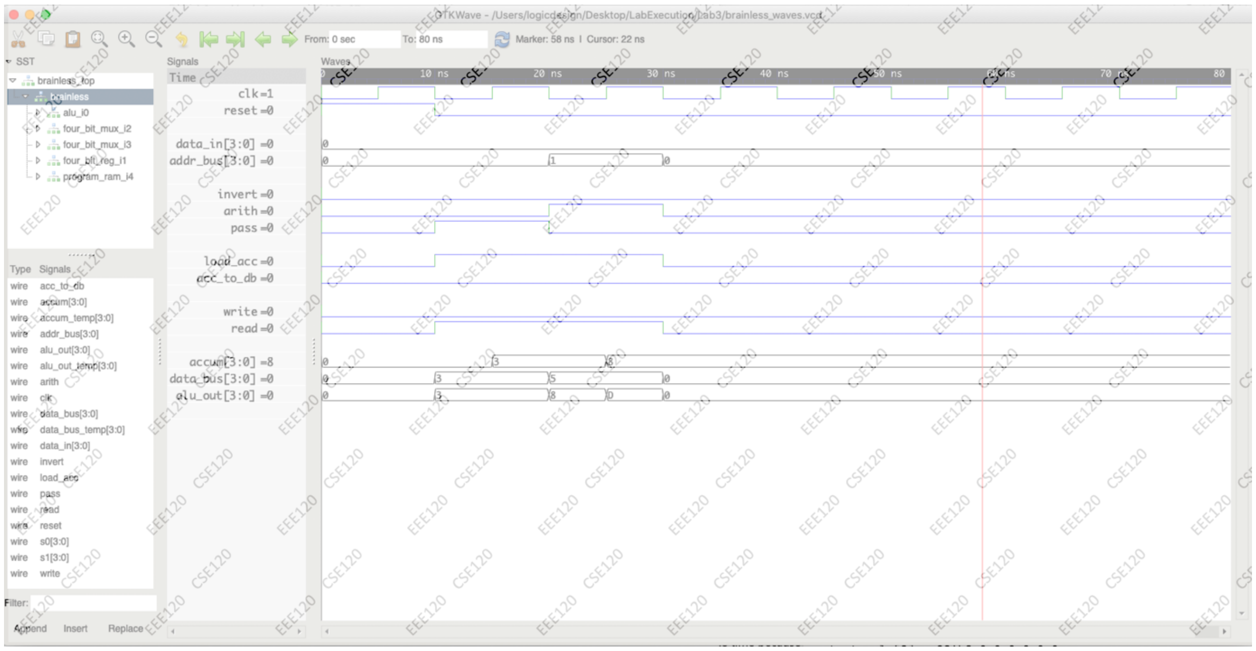


Figure 16. Waves for the simulation of 3+5.

Task 3-6: Create Additional Tests.

You will be using the brainless CPU as a subcircuit in Lab 4. Therefore, we want to make sure it is working properly now rather than trying to debug it while working on Lab 4. In this task, you'll create additional tests to verify that your circuit is, in fact, working properly. Make three copies of brainless_stim.v and name them brainless_ext_write.v, brainless_int_write.v, and brainless_alu.v. Then, modify the inputs and expected outputs in each so that you perform the following additional tests:

1. Write a value from data_in to an address in the program_ram. This is an external write to the RAM. (brainless_ext_write.v)

2. Store a nonzero value from the accumulator to an address in the program_ram. This is an internal write to the RAM. (brainless_int_write.v)
3. Demonstrate any two additional ALU functions, storing the results in the accumulator. This may be done with a single simulation. The ALU functions may not be the pass through or addition functions used in the example program provided in brainless_stim.v. (brainless_alu.v)

For each of the above, modify the stimulus and expected response as needed and run them with iverilog. When they work as expected, paste the stimulus you used and a screenshot of the waves similar to Figure 16 into your template. For the stimulus, just paste in the values assigned to test_vals as shown in Figure 17.

```
test_vals[0] = 28'h0_0_0_0_0_0_4;    // reset - this should always be the first vector
test_vals[1] = 28'h3_3_3_0_0_3_1;    // get 3 into the accumulator
test_vals[2] = 28'h8_5_D_0_1_5_1;    // add 3+5 and store in the accumulator
test_vals[3] = 28'h8_0_0_0_0_0_0;    // do nothing the rest of the way
test_vals[4] = 28'h8_0_0_0_0_0_0;
test_vals[5] = 28'h8_0_0_0_0_0_0;
test_vals[6] = 28'h8_0_0_0_0_0_0;
test_vals[7] = 28'h8_0_0_0_0_0_0;
```

Figure 17. Example of what to paste in your template.

For the two tests which write to the RAM, make sure that the RAM value shows up on the **data_bus** output so it can be seen. Do this by properly controlling the select lines to the muxes.

To create these tests, use the same approach we took for the example test in brainless_stim.v. Ask yourself where the data is now and how the controls need to be set to get the data where you want it.

IMPORTANT: Do not modify anything above the line in the stimulus file that reads:

// IMPORTANT: ONLY MODIFY BELOW THIS LINE ***

You will need to run the iverilog command to recompile your design for your new stimulus files, substituting the name of your new stimulus file for brainless_stim.v. However, you do not need to recompile if only ram_vals.txt changes since it is read in at run time.

Since the VCD file which holds the waves has the same name each time, you can simply reload the waves in GTKWave using File-> Reload Waveform. And remember that once you have a setup that you like in the waveform window, you can save it using File->Write Save File. That way, if you come back later, you can recreate the same arrangement by opening the waves and then selecting File->Read Save File and finding the file that you previously saved.

IMPORTANT: If you find that there is an issue with your design, you'll need to go back to Digital and fix it. Then, you'll need to export it to Verilog again. This means editing the brainless.v file to insert the lines into the RAM module as shown in Figure 15.

Once you are satisfied that all your tests are passing, take a screenshot of your brainless CPU schematic in Digital and paste it into your template. Also paste in screen shots of your simulation waveforms from GTKWave for the simulations your ran for this task.

Task 3-7: Create a video and submit your report.

Create a video showing your schematic in Digital, and your waveforms and explain how your design works. 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 Lab3 directory. Create a zip file of the Lab3 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 3! You will be using these designs in Lab 4 so be sure to hang onto them.