Simulation Lab 4: The Microprocessor

Prerequisites: Before beginning this laboratory experiment you must be able to:

- Use Logisim.
- Have completed Simulation Lab 1: Half Adder, Increment & Two's Complement Circuit.
- Have completed Simulation Lab 2: 4-Bit Full Adder, Multiplexer, Decoder & Buffer.
- Have completed Simulation Lab 3: Arithmetic and Logic Unit.

Equipment: Personal computer and Logisim.

Objectives: In this laboratory exercise you will design three types of memory and the communication bus architecture of a simple microprocessor. Then you will act as its controller to cause it to execute a simple program. Finally you will complete the design of a microprocessor, create an instruction set, enter a program into memory and execute the program.

Outcomes: When you have completed the tasks in this experiment you will be able to:

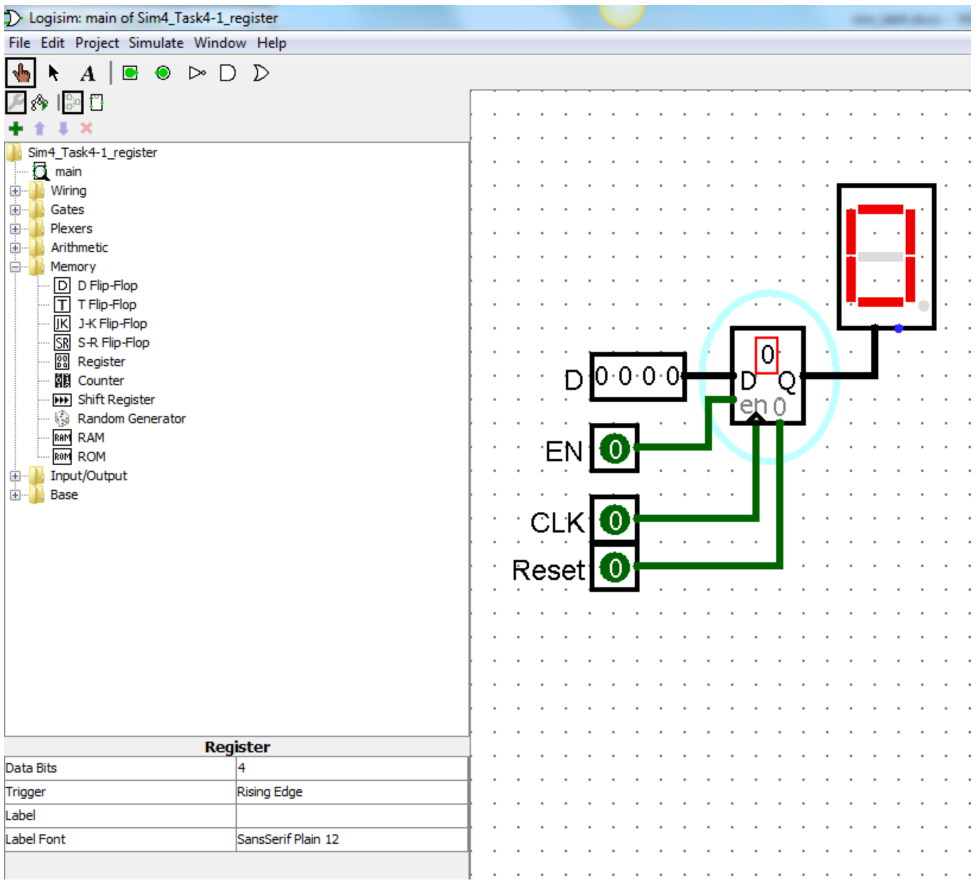
- Build, debug and control a simulation of a central processing unit (CPU).
- Build, debug and control a simulation of a ROM, RAM and an output port.
- Build and debug a simulation of a microprocessor that is absent a controller.
- Act as the controller for an elementary microprocessor.
- Design a PROM-based controller for an elementary microprocessor.
- Create an instruction set for an elementary microprocessor.
- Use the language of your instruction set to create a program and enter it into memory.
- Execute a program on your simulated microprocessor.

Introduction

In simulation labs 1, 2, and 3, we built almost all of the combinational logic circuits we'll need for our microprocessor. In this laboratory exercise, our aim is to build a few more circuits that we'll need and to use the subcircuits we have already created to build a complete microprocessor. In this lab exercise we want to accomplish several things. First we will build Logisim memory circuits including ROM, RAM and output port that will be used in our microprocessor to store program and save/display result. Second, we will assemble our microprocessor (without the controller) 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. Third, you will complete the microprocessor design by building a controller, defining an instruction set for the controller, and adding the controller to your brainless microprocessor. 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.

Task 4-1: Build the Brainless Central Processing Unit

Recall registers are arrays of flip-flops that share common set and/or reset control inputs and a common control enable signal for preventing the flip-flops from responding on an input-clocking event. To insert a register in Logisim, click *Memory->Register*. Set properties *Data Bits* to **4**, and *Trigger* to **Rising Edge**. Figure 1 shows how to use a 4-bit register in Logisim. Note the *Reset* input is **active high**, so in order to reset the register to value 0, click *Reset* to 1, but don't forget to change it back to zero, otherwise, the register will remain in the reset state. If EN is 0, Q does not respond to the rising edge of the clock, and will keep its previous value. If you want to latch the input D to the register, EN has to be set to a 1. Note the current stored value of the register shows at the top center of the symbol.



Register	
Data Bits	4
Trigger	Rising Edge
Label	
Label Font	SansSerif Plain 12

Figure 1. 4-bit Register in Logisim

In Simulation Lab 2, you learned how to build a 4-bit buffer from a 1-bit buffer. In Logisim, you can use a 4-bit buffer directly as shown in Figure 2. To insert a buffer in Logisim, click *Gates->Controlled Buffer*. Set *Data Bits* to **4**,

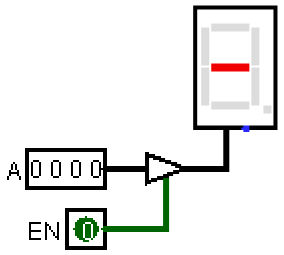


Figure 2. 4-bit Buffer in Logisim

Now using register, buffer and the ALU subcircuit you built in Simulation Lab 3, build the brainless microprocessor central processing unit (CPU) as shown in Figure 3. In the circuit, hex digit displays are used to get access to the information of the various buses. Also **Tunnels** are used to simplify the wiring of the *Clock* and *Reset* signals. This will become especially handy when you assemble your complete microprocessor. To insert a Tunnel in Logisim, click *Wiring->Tunnel*. Also note in order to assign a logic high or low value to an input, for example, *Cin*, click *Wiring->Constant*, then change the *Value* to 0x0, or 0x1.

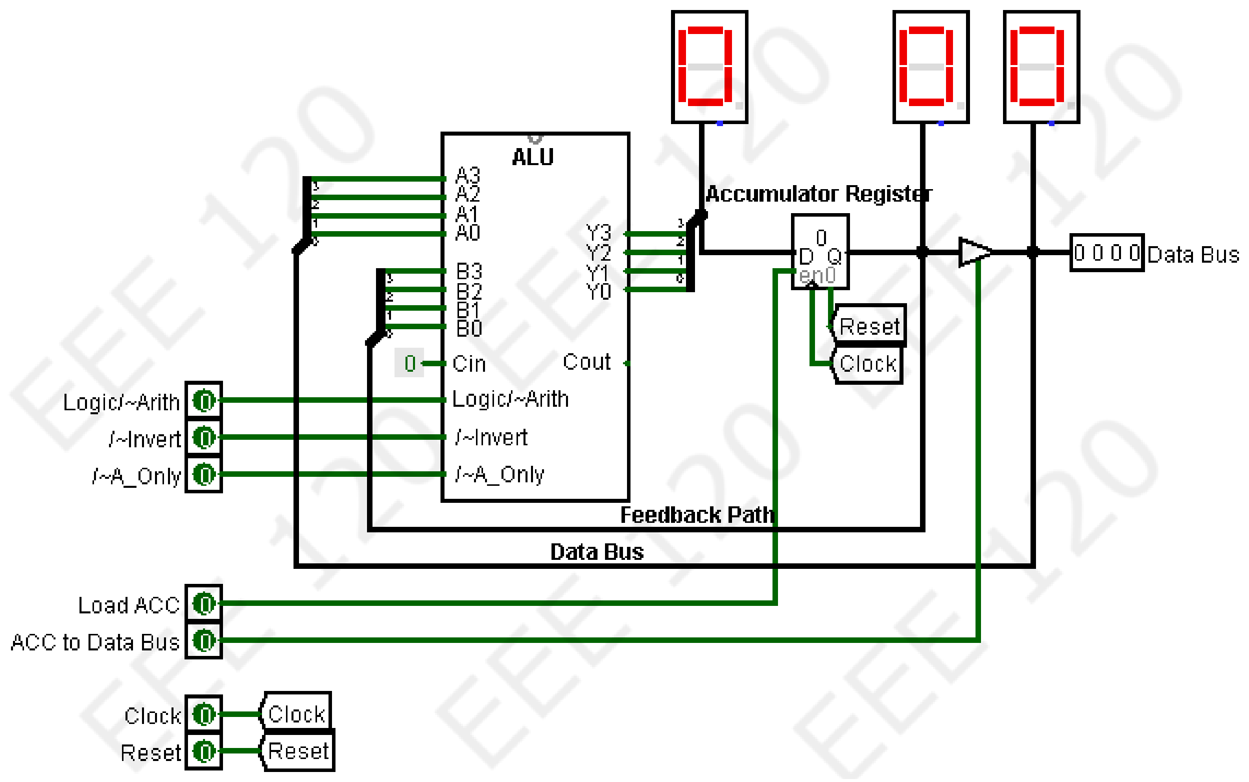


Figure 3. Brainless central processing unit

The presence of the accumulator register (controlled by the clock signal) in Figure 3 transforms the circuit we have been building in a fundamental way: it transforms it from a combinational-logic circuit into a synchronous circuit. We are getting very close to completing the design of a microprocessor CPU. In the next task, you are asked to test and understand the workings of the partial CPU you just constructed.

Task 4-2: Test and Control the Brainless Central Processing Unit

Because this brainless CPU contains an ALU and a register for storing information, we can now perform meaningful arithmetic and logical operations. To test this circuit, you will add two numbers together and store them in the accumulator by doing the followings:

- Set the 4-bit binary keyboard that controls the *Data Bus* values to any number. Observe this number on the data bus hex digit display.
- Set the input pins connected to the ALU control lines so that the ALU implements the pass-through operation. (You may want to refer to the ALU function definition table you constructed in the previous lab exercise.)
- Set *Load ACC* = 1 so that the accumulator register is enabled and make sure that the *Reset* = 0 so that the register can store data under synchronous control.
- Toggle the *Clock* input pin twice so that you lock the output of the ALU into the accumulator with a rising clock edge.
- Set the 4-bit binary keyboard to any number that, when added to the one in the accumulator is less than F. (You can set it to any number but getting a sum less that F simplifies checking your result.)
- Set the input pins connected to the ALU control lines so that the ALU implements the ADD operation. Observe the correct sum at the output of the ALU.
- Lock the sum in to the accumulator by toggling the *Clock* input pin twice. (Make sure the *Reset* switch remains reset to 0.) Observe that the correct sum appears at the output of the accumulator.

Test a few other additions and any other operations you believe necessary. Once you are convinced that your circuit is working correctly, remove the 4-bit binary keyboard (*Data Bus*), set the *ACC to Data Bus* pin to 1, and observe that the output of the accumulator appears on the data bus. When this output appears on the data bus, how does that change the output of the ALU? If the 4-bit binary keyboard is not removed and the *ACC to Data Bus* pin is set to 1, what would you expect to see in the hex digit display attached to the data bus? Why do you think the register at the output of the ALU is called the 'accumulator'? Play with this circuit until you are sure you understand how it works. It is important that you understand how this circuit works because it forms the kernel of our processor.

Task 4-3: Build the Addressing Logic

The addressing logic we will need in our microprocessor is shown in Figure 4. We will simulate addresses in this laboratory exercise using the 4-bit binary keyboard. These addresses, when applied to the decoder, will cause one of the decoder's output lines to become active (high). Each of these output lines will be routed (in a later task) to a different memory location and used to enable that memory location to make its contents available to our data bus.

Build the addressing logic circuit shown in Figure 4 using a 4-bit binary keyboard, a hex digit display and the 4-to-16 decoder you have built. Connect the first eight output lines to a Logisim bus as shown. Test your circuit, record the tests you performed, then add it to the partial CPU circuit you built in the previous task. (Locate this circuit anywhere for now. We will place it in an appropriate location in a later task.)

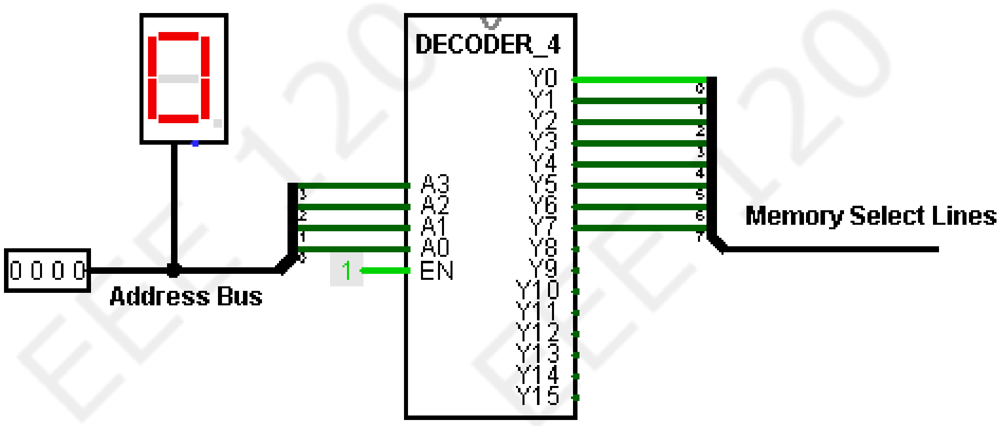


Figure 4. Addressing logic

Task 4-4: Build a 4-Bit ROM Memory Cell

During lecture, different types of external memory were introduced. We will simulate ROM in such a way that we can conveniently change its contents using our computer mouse and we will connect the output of this ROM to our data bus using a buffer circuit. We will use the Logisim 4-bit keyboard as our ROM and control its connection to the data bus using the 4-bit buffer circuit. By 'clicking' on the keyboard you will be able to change the value 'stored' in the simulated ROM. It will not be possible to change the value stored in the keyboard by using control lines in our simulation or by feeding data from the data bus into the keyboard; hence this memory circuit truly functions as ROM.

Figure 5 shows that a ROM memory cell will gain access to the data bus only when the *Read* line is high and when the memory select line (coming from the addressing logic) is high. (The *Read* line, which you will control in this exercise, will be high when we want to read a value from memory, and low when we want to write to memory.) Build the 4-bit ROM memory and buffer circuit shown in Figure 5, test it, and save it. Remember to record the results of your tests in your lab template.

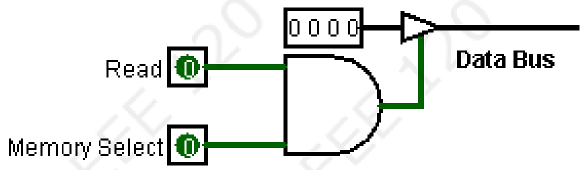


Figure 5. 4-bit ROM memory cell

Task 4-5: Build a 4-Bit Output Port

The I/O port we will simulate will be unidirectional; we will be able to write data to the port (and view it using a Logisim hex digit display), but once written, the data will not be retrievable by the microprocessor; hence our port will be an output port.

To control writing into this output port we will need to use the *Write* and memory select control lines. When both the *Write* signal and the memory select line coming from the decoder are high, we will be able to write into memory. The circuit we will use to simulate this port is shown in Figure 6. The hex digit display at the output of the 4-bit register will allow us to view the data written into the register. Build the 4-bit output port shown in Figure 6, test it, and save it. Remember to record the results of your tests in your lab template.

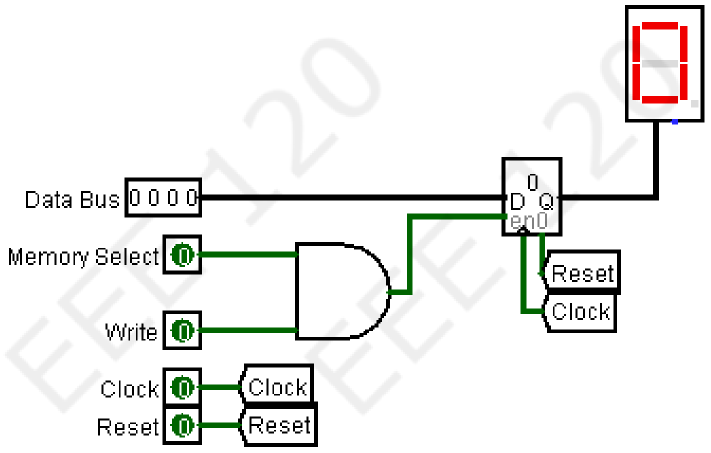


Figure 6. 4-bit output device

Task 4-6: Build a 4-Bit RAM Cell

The last type of external memory we will use in our microprocessor will be random access memory (RAM). The simulated RAM circuit we will use is shown in Figure 7. We will use the same control philosophy for writing into the RAM register as we used with the output port: two control signals, the *Write* line and the connected memory select line (from the decoder), will need to be high to write into the RAM. To read from the RAM, we will use the same control philosophy we used with ROM: the memory select line (from the address decoder) and the *Read* line will need to be high; these two lines, when high, will cause the buffer to access the data bus.

Build the 4-bit RAM in Figure 7, test it and save it. Remember to record the results of your tests in your lab template.

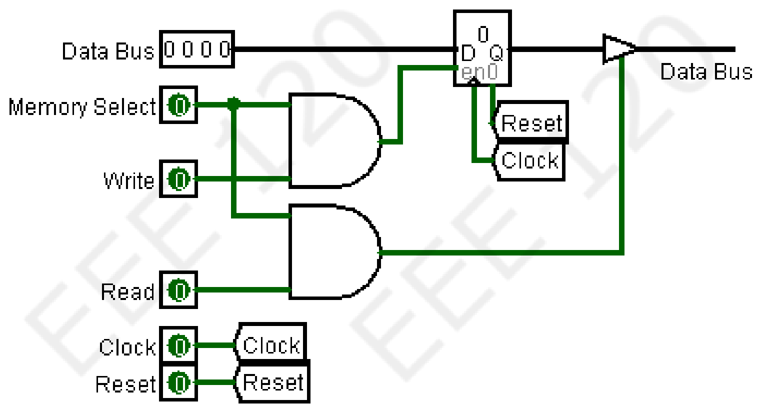


Figure 7. 4-bit RAM

Task 4-7: Build the Brainless Microprocessor

The Logisim circuit schematic of the brainless microprocessor is shown in Figure 8. The left side of the diagram contains the CPU you built in Task 4-1, the top center contains the addressing circuitry you built in Task 4-3, the bottom right contains eight 4-bit ROM memory cells you built in Task 4-4, and the bottom center contains the output port and 4-bit RAM cell you built in Task 4-5 and Task 4-6. Notice that because the enable of each memory cell is connected to a different memory select line from the address decoder, each memory cell occupies a different address in the memory-address space. Verify for yourself that addresses 0H – 7H access ROM, address EH (14 decimal) accesses the output port and FH (15 decimal) accesses the RAM location.

Build the circuit shown in Figure 8. Do not panic when you look at this circuit! You already have everything you need to wire up this circuit. It is really not too hard to build, and it has some very promising capabilities.

Task 4-8: Test and Control the Brainless Microprocessor

The circuit in Figure 8 is almost a microprocessor; it is only missing the 'brains', or more frequently called the controller or control circuitry. In a later task you will build the controller for your microprocessor, the objective of this task is for you to get familiar with how the microprocessor is controlled, by acting as the controller, so that the controller design will make more sense. For now, you will function as the 'brains' of the microprocessor, by manipulating the control lines to get it to process the data.

If you have ever wondered how a computer works, you will come very close to understanding how it works as you act as the controller when performing this task. You will test whether your brainless microprocessor functions as desired. Let's conduct one test to validate its performance; let's add two numbers, 3 and 5, and observe the sum, 8, in the accumulator.

To complete this test, perform the following steps in order:

1. Entering Data in ROM Memory

- Enter 3 in ROM at address 0, and 5 in ROM at address 1. (Remember you do this by clicking on the 4-bit keyboards so you have 0011 and 0101 at these respective locations.)

2. Getting the First Operand (Load ACC with 3)

- *ACC to Data Bus* = 0 (Prevent data bus conflicts.)
- *Read* = 1, *Write* = 0 (Read number 3 from ROM address 0)
- *Address Bus* = 0 (Enter 0000 in the 4-bit keyboard connected to the address bus.) At this point you should observe the value '3' on the hex digit display connected to the Data Bus.
- Put the ALU in 'pass-through' mode. (You may want to refer to the ALU function definition table you constructed in Simulation Lab 3.) At this point you should observe the value '3' on the hex digit display connected to the output of the ALU.

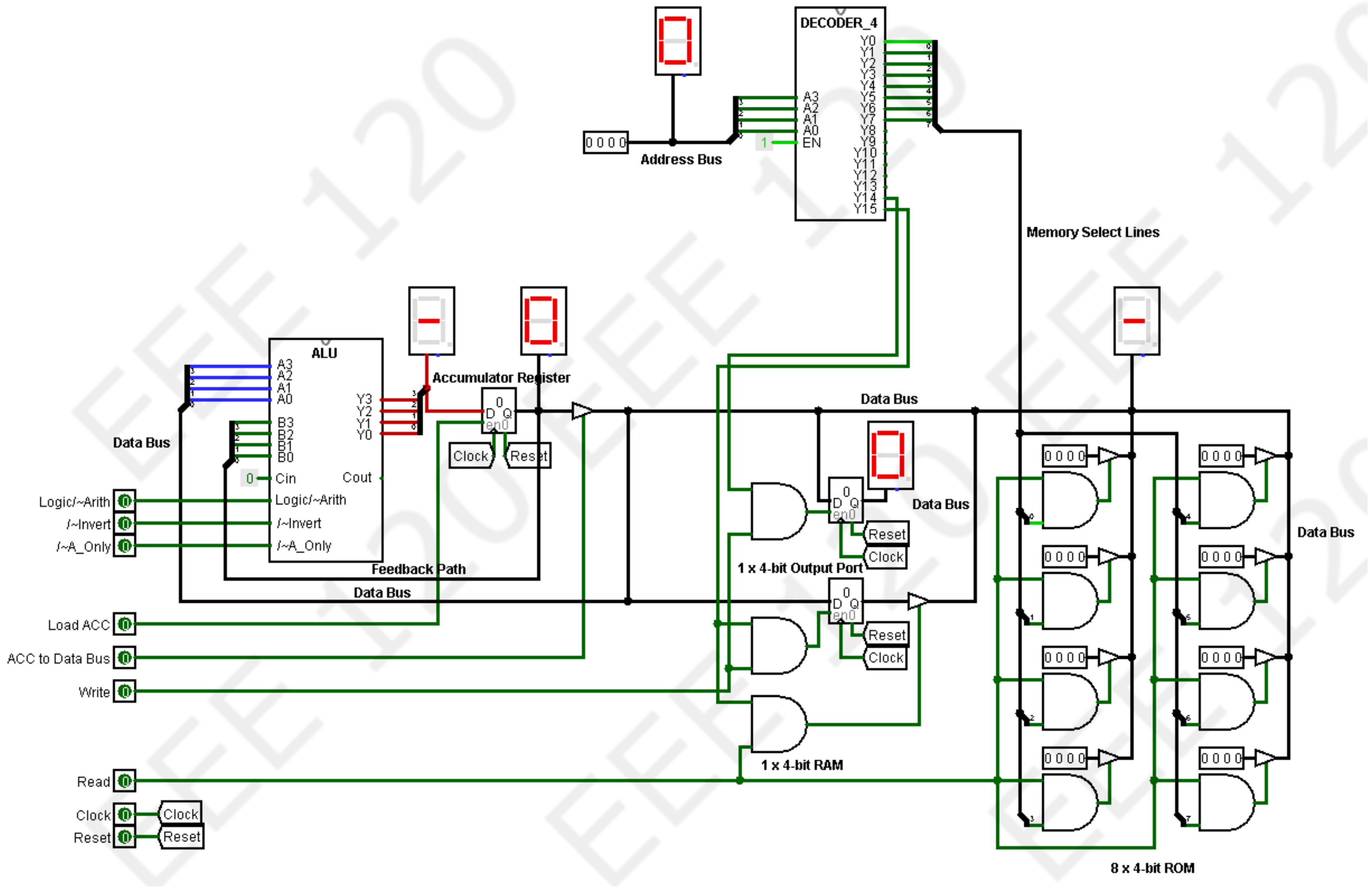


Figure 8. The brainless microprocessor

- *Load ACC* = 1 (This will enable the accumulator to be loaded.)
- Make sure the *Reset* = 0. (This will allow the accumulator [and RAM/Output Port] to accept data under synchronous control.)
- Click the *Clock* switch through one complete cycle to load the accumulator. You have now loaded the accumulator with '3'. The value '3' should be viewable on the hex digit display connected to the output of the accumulator.

3. Getting and Adding the Second Operand (ADD 5 to ACC)

- Make sure the following switches remain in their previous positions:
  ACC to Data Bus = 0,
  Read = 1,
  Write = 0,
  Load ACC line = 1,
  Reset = 0.
- Enter 0001 in the 4-bit keyboard connected to the address bus. At this point, you should observe the value '5' on the hex digit display connected to the data bus.
- Put the ALU in ADD mode. (You may want to refer to the ALU function definition table you constructed in Simulation Lab 3.) At this point you should observe the value '8' on the hex digit display connected to the output of the ALU.)
- Click the *Clock* switch through one complete cycle to load the accumulator. You have now completed an addition and loaded the accumulator with the result, 8. The value '8' should be viewable on the hex display connected to the output of the accumulator.

4. Storing Result to Memory (Store ACC to RAM, or, Store ACC to Output Port)

- The last step is to store this result into memory. You can store to the output port, or the RAM. The steps needed to store this result into memory are left for you to discover.

Your First Program

Congratulations! What you have done in the preceding steps is to act as the microprocessor's controller to execute a three-instruction program:

- Load ACC with 3
- ADD 5 to ACC
- Store ACC to RAM (or Output Port).

Take a moment to revel in your own success and congratulate yourself!

With the execution of the above program you have tested some of the capabilities of your brainless microprocessor. What other tests do you feel are necessary to provide evidence that your circuit operates as designed? Describe the tests you perform and enter the result in your lab template. The following is the minimum set of tests you need to perform:

- Perform an AND operation on two numbers instead of addition.
- Subtract two numbers by taking a two's complement of the first number and then adding it to the second number. (Do not worry about overflow or carries.)

For all of the instructions you perform, e.g., Load ACC, Negate, AND, ADD etc., create a table listing the values to which all control lines must be set during every clock cycle of each step of each instruction you test. (This information will be useful to you when you design your controller later.) It is important that you stop and make sure that you understand how this circuit works. If you do not understand how this circuit operates, you will not be able to implement the controller.

Task 4-9: Build 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. Build the memory address generation circuit shown in Figure 9. The memory address generation circuit is comprised of two independent pieces. One piece consists of a register and increment circuit 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). The output of the address generation circuit is a mux, which controls whether the PC or MAR drives the address bus. The control input to the mux, *Use PC* (short for Use Program Counter for address), is high when the PC values are to drive 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 be able 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.

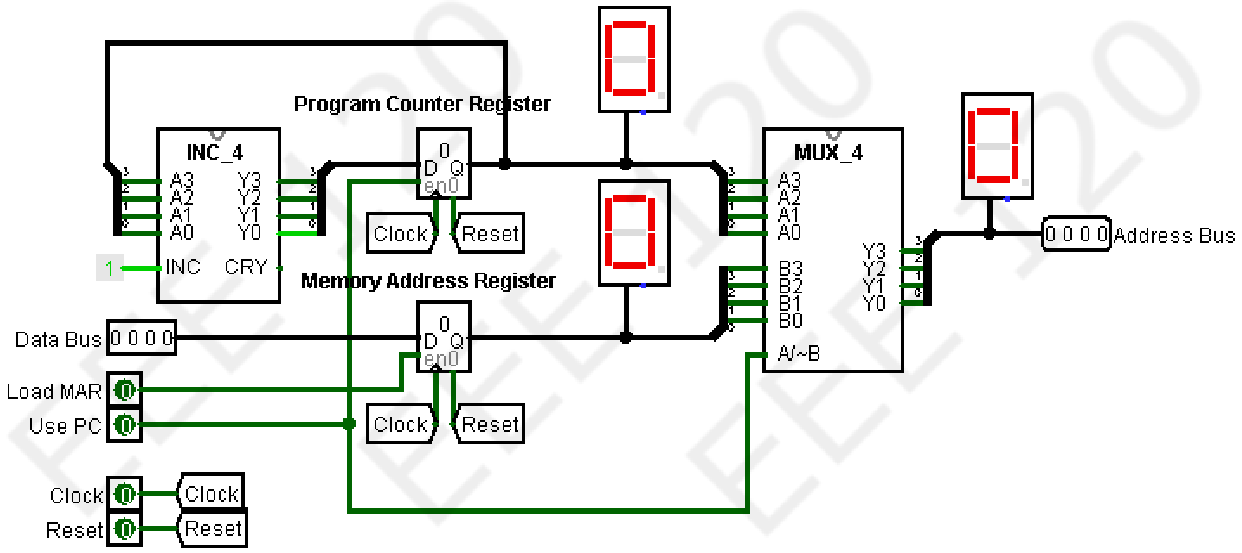


Figure 9. Memory address generation circuit

Task 4-10: Build the Controller Circuit

Recall the brainless microprocessor, how did the microprocessor know how to execute each of the instructions in our program? The answer, of course, is because we acted as the controller, we knew how to control all of the control signals to execute each instruction. Next, we will design a controller that will be able to automatically view instructions (in the form of a program) stored in ROM and then automatically perform the operations needed to carry out each instruction.

The control signals the controller need to generate along with their function and active values are listed in Table 1. We will manually control the *Clock* and *Reset* signals by toggling the input pins.

Table 1. Definition of Controller Output Signals

Control Signal	Function	Active Value
Load IR	Enables IR to be loaded with opcode from data bus.	High
Write	Enables memory to be written into.	High
Read	Enables memory to be read from.	High
ACC to Data Bus	Enables ACC contents to drive the data bus.	High
	Prevents ACC contents from driving the data bus.	Low
Load ACC	Enables ACC to be loaded with ALU output.	High
Load MAR	Enables MAR to be loaded from data bus.	High
Use PC	Selects PC to drive data bus.	High
	Selects MAR to drive data bus.	Low

/~A_Only	Causes ALU to operate on the A & B operands.	High
	Causes ALU to operate on operand at port A.	Low
/~Invert	Causes ALU to operate on A and/or B port operands without inverting either.	High
	Causes ALU to invert the A operand.	Low
Logic/~Arith	Causes ALU to perform a logical operation.	High
	Causes ALU to perform an arithmetic operation.	Low

The controller we will design and build is a PROM-based synchronized Mealy finite state machine as shown in Figure 10.

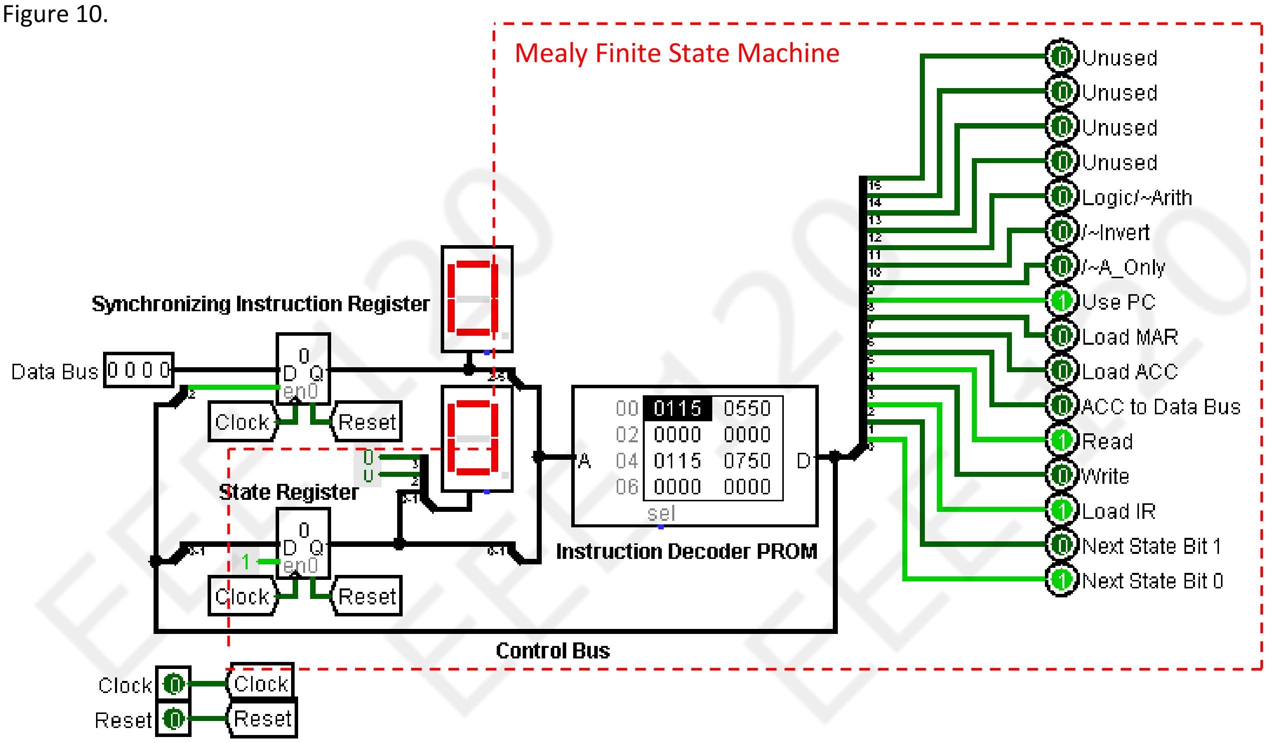
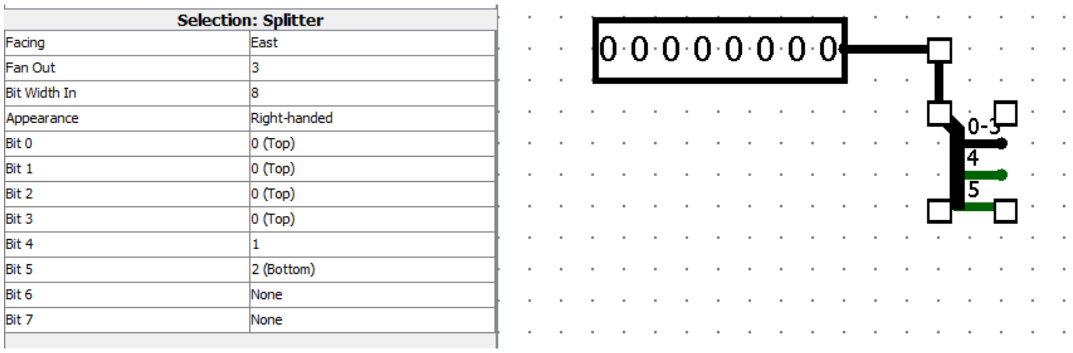


Figure 10. Microprocessor controller circuit

The controller circuit is divided into two pieces. One piece, in the red dashed lines, is the classical PROM-based Mealy finite-state machine (FSM). The external input to the Mealy machine controller consists of a 4-bit opcode, which is supplied by the data bus to the synchronizing Instruction Register (IR). The outputs from our Mealy machine controller will perform two conceptually distinct chores. One group of outputs (from bit 2 to bit 11) will provide control inputs to the various blocks of our microprocessor. You have manipulated manually most of these signals listed in Table 1 to control the brainless microprocessor. A second group of outputs (from bit 0 to bit 1) from our Mealy machine controller are needed to drive the state flip-flop array to achieve the appropriate next state—given knowledge about the present state and external input. As discussed in the lecture, there are four states in the Mealy FSM. It means that we need a two-flip-flop state array to hold the present-state information. Assuming that we use D flip-flops to store the present state, our controller needs two outputs to drive the state flip-flop array. Note that Figure 10 shows that we are implementing the controller functions using a PROM rather than combinational logic. (Notice also that the PROM in this figure has 4 unused outputs (from bits 12 to bits 15).

The other piece of Figure 10, the synchronizing IR , serves two functions. One function is to keep the opcode constant for the duration of clock cycles needed by the opcode to fulfill its task. The second function is to ensure that the input to the controller changes in synchronism with the processor's clock. Without the synchronizing IR, the controller would perform erratically. For example, if the controller is executing an opcode, the operand that is being operated upon is driving the data bus. If the IR is absent, then the controller interprets the value on the data bus as an opcode. This false ‘opcode’ causes the output lines of the controller to change, in turn changing (perhaps) the value driving the address bus, which in turn changes the memory location accessed. This changes the ‘opcode’ the controller receives, which changes the controller’s outputs, etc. Clearly, the synchronizing IR is important to the proper functioning of the controller.

Build the circuit (except the PROM) in Figure 10. Note the state register is a **2-bit** register. You **DON’T** have to build the 16-bit splitter on the right. Figure 11 below will help you build the more ‘sophisticated’ bus splitters in the circuit.



Selection: Splitter	
Facing	East
Fan Out	3
Bit Width In	8
Appearance	Right-handed
Bit 0	0 (Top)
Bit 1	0 (Top)
Bit 2	0 (Top)
Bit 3	0 (Top)
Bit 4	1
Bit 5	2 (Bottom)
Bit 6	None
Bit 7	None

Figure 11. Splitter demonstration

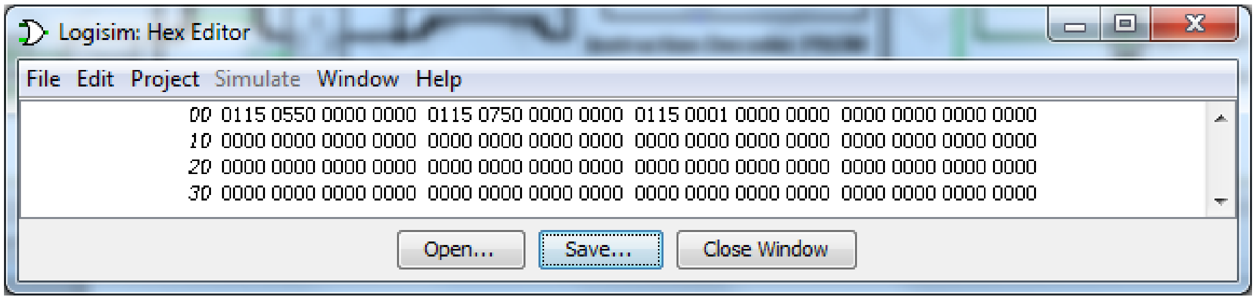
Completing the design of the controller means specifying the contents of the PROM so that its outputs control the ALU, muxes, decoders, and registers of the processor to perform the operations needed to fulfill the requirements of each opcode/instruction we desire. As discussed in the lecture, the input-output data for our limited instruction set instruction decoder PROM is shown in Table 2. The table defines microinstructions for three instructions “Load ACC with [operand]”, “Add [operand] to ACC” and “Stop”. In a later task, you will expand this table by adding microinstruction definitions for additional instructions such as “subtract”, “store” and etc.

Table 2. Input-Output Data for Limited Instruction Set PROM

	Instruction	Load ACC				Add to ACC				Stop			
	Opcode	0				1				2			
	Present State	0	1	2	3	0	1	2	3	0	1	2	3
Description	**Pin number**												
Next State Bit 0	0	1	0	0	0	1	0	0	0	1	1	0	0
Next State Bit 1	1	0	0	0	0	0	0	0	0	0	0	0	0
Load IR	2	1	0	0	0	1	0	0	0	1	0	0	0
Write	3	0	0	0	0	0	0	0	0	0	0	0	0
Read	4	1	1	0	0	1	1	0	0	1	0	0	0
ACC to Data Bus	5	0	0	0	0	0	0	0	0	0	0	0	0
Load ACC	6	0	1	0	0	0	1	0	0	0	0	0	0
Load MAR	7	0	0	0	0	0	0	0	0	0	0	0	0
Use PC	8	1	1	0	0	1	1	0	0	1	0	0	0
/~A_only	9	x	0	0	0	x	1	0	0	x	0	0	0
/~Invert	10	x	1	0	0	x	1	0	0	x	0	0	0
Logic/~Arith	11	x	x	0	0	x	0	0	0	x	0	0	0
x	12	x	x	0	0	x	x	0	0	x	x	0	0

x	13	x	x	0	0	x	x	0	0	x	x	0	0
x	14	x	x	0	0	x	x	0	0	x	x	0	0
x	15	x	x	0	0	x	x	0	0	x	x	0	0
	HEX Equivalent	0115	0550	0000	0000	0115	0750	0000	0000	0115	0001	0000	0000

Now we will create a PROM in Logisim with the contents above. Click *Memory->ROM*. Set *Address Bit Width* to **6** and *Data Bit Width* to **16**. For *Contents*, click to edit according to Figure 12 below. Here we replace all don't cares, 'x's in Table 2, with zeroes.



```
00 0115 0550 0000 0000  0115 0750 0000 0000  0115 0001 0000 0000  0000 0000 0000 0000
10 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
20 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
30 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
```

Figure 12. PROM contents for instructions defined in Table 2

After you edit all the above properties, add the ROM symbol and complete the controller circuit in Figure 10.

Task 4-11: Build the Complete Microprocessor Circuit

A term's worth of work is about to pay off! Build the complete microprocessor schematic as shown in Figure 13.

Task 4-12: Write and Execute a Simple Program for Your Microprocessor

Now that you have your microprocessor built, you may be very excited to try it out! As a test of your microprocessor, suppose we try the following program:

Instruction	**Comment**
Load ACC with 3	;Put the number 3 into the Accumulator
Add 5 to ACC	;Add 5 to the value stored in the Accumulator
Stop	;Stop executing new instructions

In memory, this program would look like this:

Address	**Contents**	**Comment**
0	0	;The 'Load ACC' Opcode
1	3	;The number '3' to be loaded into the Accumulator
2	1	;The 'Add to ACC' Opcode
3	5	;The number 5 to be added to the Accumulator
4	2	;The 'Stop' Opcode

This program will add the numbers 3 and 5 and store the result in the accumulator. Enter the program in your ROM starting at address 0, and then reset your microprocessor using the *Reset* line. Step through your program one step at a time by toggling the input pin that drives the *Clock* signal line, and make sure every step of every instruction works correctly. Record the operation of each step, i.e. record the values of the control lines, and record whether data is being moved properly according to those control line settings. Do you get '8' stored in the

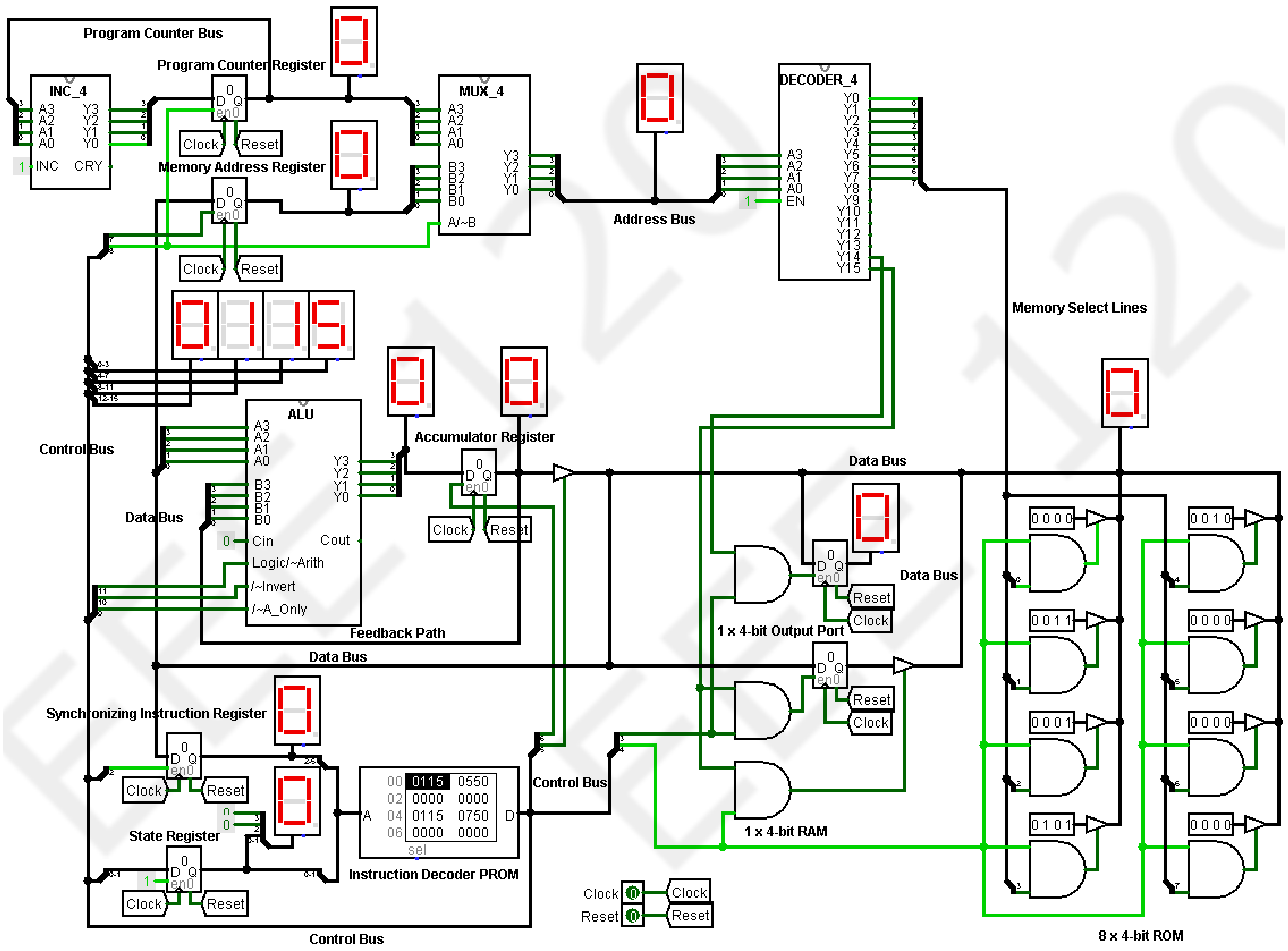


Figure 13. The complete microprocessor unit

accumulator when your program is done executing? If not, you have some debugging to do. Do not be too disappointed if your microprocessor does not work the first time.

One good approach to debugging your microprocessor is to check that the program you just executed is correctly performing each of the sequence of microinstructions as shown in Figure 14. This figure shows all the control signals and register values throughout execution of the entire program. Upon resetting, your controller accessed location 00H of your instruction decoder, which contained the fetch microinstruction. The outputs of your decoder caused the first opcode of your program, 0H, to be made available at the input of the IR. The first clocking event caused the fetch microinstruction to load the first opcode of your program, 0H, into your IR. With this opcode loaded in the IR and the state register loaded with 01B, your controller began executing the execute microinstruction associated with the first instruction in your program, 'Load ACC with 3'. The second clocking event loaded '3' into the ACC and your controller began executing the fetch microinstruction associated with the 'Load ACC' instruction. Etc.

Two functional blocks that you haven't tested so far are the memory address generation circuit and the controller circuit. So pay particular attention to make sure the address generation circuit outputs correct value on the address bus, and the controller generates correct control signals and next state bits.

Task 4-13: Add the 'AND', 'Zero', 'Subtract', 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', 'Subtract', and 'Store ACC' instructions to your microprocessor circuit.

- The 'AND' instruction should perform a bit-wise AND operation of the value in the ACC with an operand in memory and store the result in the ACC.
- The 'Zero' instruction should store the hex digit 0H in the accumulator by subtracting the accumulator from itself and storing the result back in the accumulator.
- The 'Subtract' instruction gets the next value in memory and subtracts it from the accumulator by forming its two's complement, adding the result to the accumulator, then storing this result back into the accumulator.
- The 'Store ACC' instruction should write the contents of the ACC into a location in memory specified by the instruction's operand.

For each instruction above, define the control signals for all the microinstructions by expanding Table 2. Update the contents of the PROM with the added instruction definitions. Then write and execute a few programs to test each of your instructions. When writing your program, 'Store' only to locations EH or FH, the output port and simulated RAM, respectively. (These are the only location that can store data under program control.) Include the programs and their outputs in your lab template along with the microinstruction definition tables for each instruction.

Task 4-14: Invent Your Own Instruction (Extra Credit)

Invent a new instruction and add it to your instruction set. Write and execute a program that tests your instruction. Include the program(s) and their outputs in your lab template along with the microinstruction definition table for your instruction.

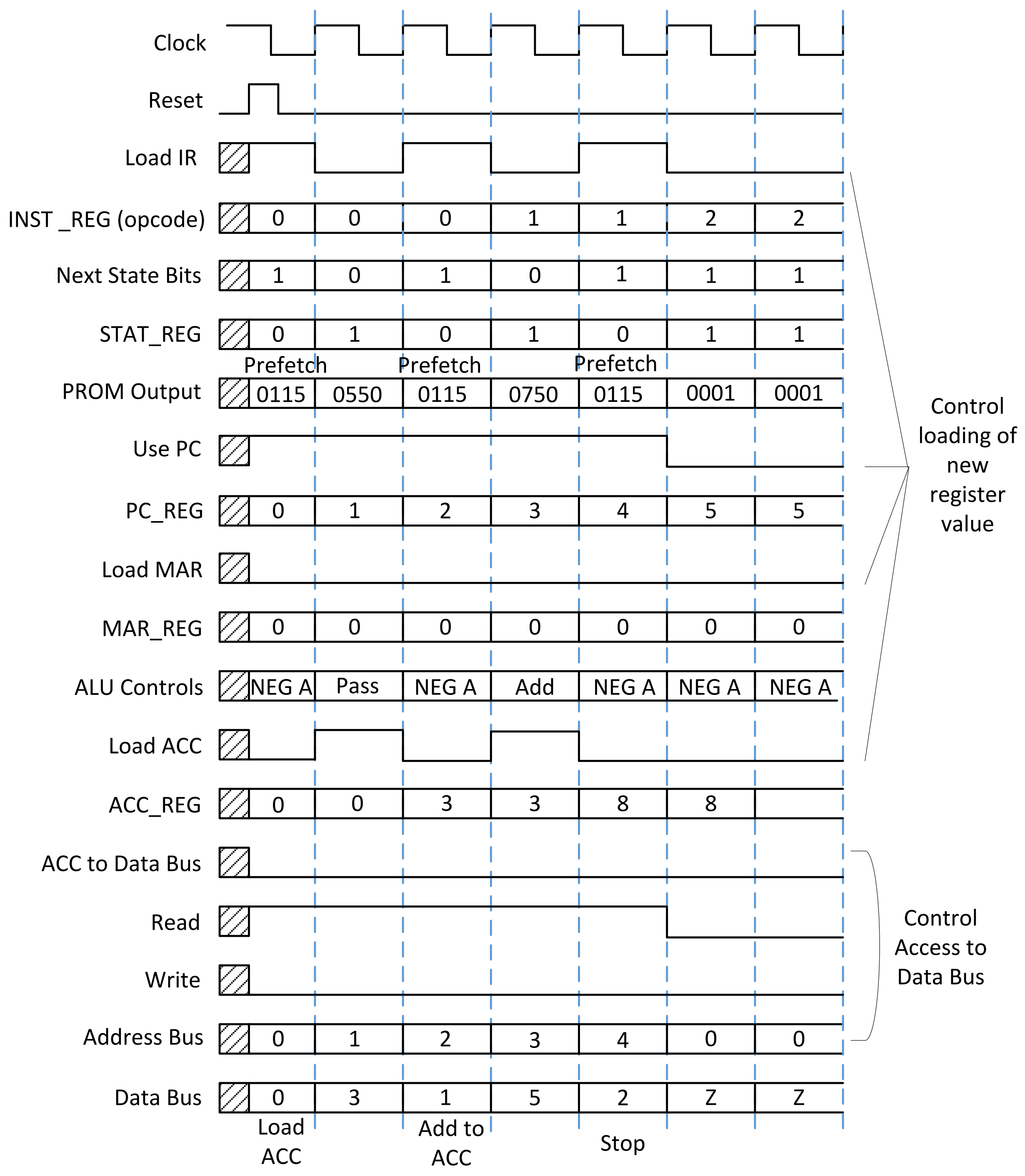


Figure 14. Control signals and register values for execution of program in Task 4-12