

CSC/EEE 120
MICROPROCESSOR DESIGN PROJECT
INSTRUCTION FOR LAB 5 (STAGE E)

E.1 INTRODUCTION

In this final stage, we will complete our microprocessor.

The microprocessor:

Look at the circuit on the next page. This figure is the complete microprocessor circuit. We have already designed all the blocks used in this circuit except for the “instruction decode.” We will discuss that circuit in a minute. To begin with, let’s look at the parts of the circuit that were not in the circuit on lab **D**.

First’ you will note that we have added more ROM memory. This additional memory is needed so that there is enough memory to store a program, which is just a series of instructions that the microprocessor is to perform. Instructions are given numeric codes, so that they can be stored in memory. For our microprocessor, the first code in a program is at address 0.

Many instructions, such as loading the number 3 into the accumulator, require data. In this case, the data is the number 3. When instructions require data, the data follows after the instruction code. Say, for example, that there is an instruction code at address Q (where Q could be any address, 0 through F). If the instruction requires data, the data will be at address $Q+1$. If the instruction requires more data, the additional data will be at address $Q+2$. After the data (if any) for that instruction will come the next instruction code. This next instruction may again require data, or it may not. More instructions will follow until the program is complete. The possible memory layouts of an instruction are illustrated below.

Instruction with no data

Memory	
Q	Instruction
Q+1	Next Instruction
Q+2	
Q+3	
Q+4	

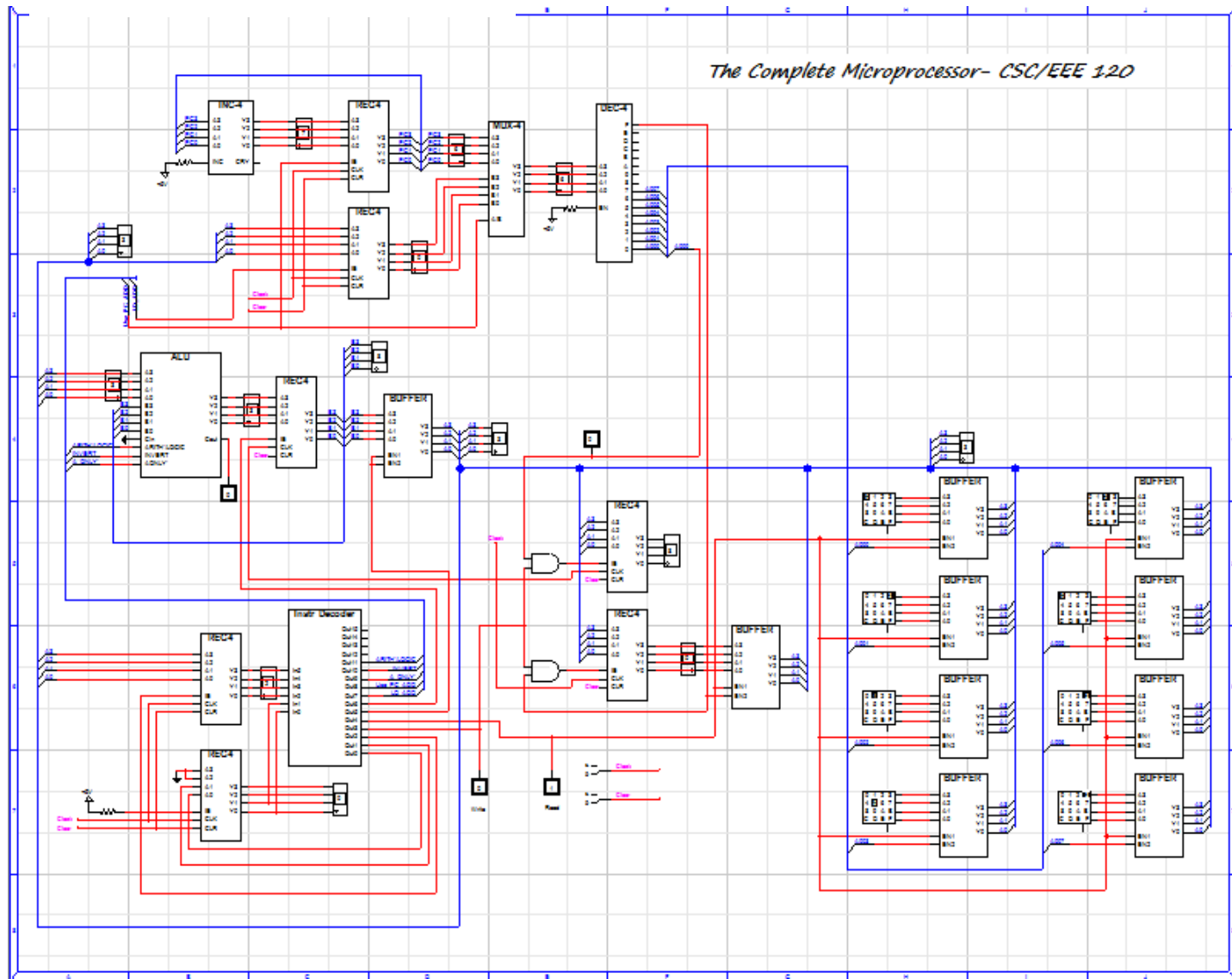
Instruction with one address of data

Memory	
Q	Instruction
Q+1	Data
Q+2	Next Instruction
Q+3	
Q+4	

Instruction With two addresses of data

Memory	
Q	Instruction
Q+1	Data
Q+2	Data
Q+3	Next Instruction
Q+4	

See Canvas for a cleaner copy



It is not a coincidence that instructions are laid out in this fashion. This is a consequence of the microprocessor's design. Let us look further at the new parts of the circuit, to see more of this design.

There are two new circuit blocks in the microprocessor that were not in the circuit of stage D. let's focusing our attention on these new circuits.

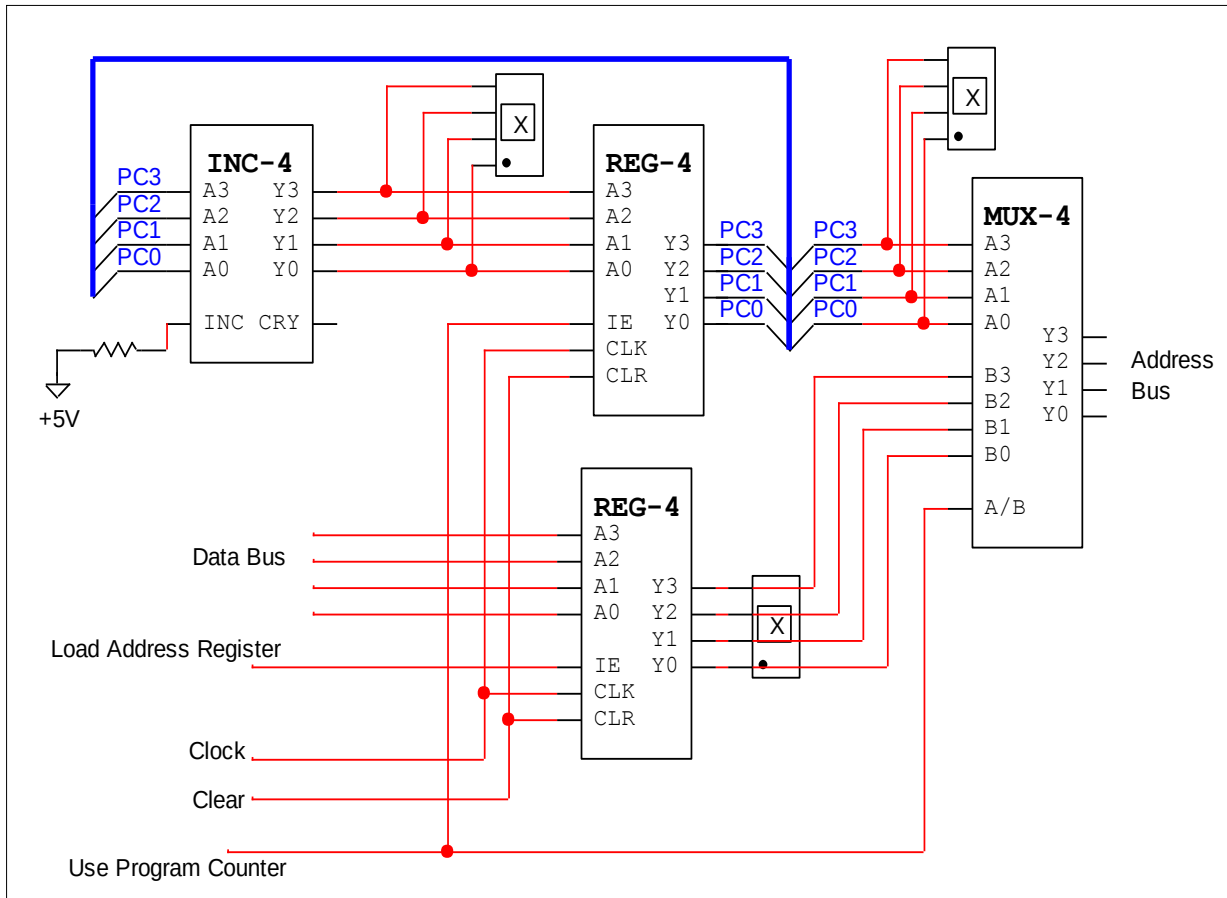
Program Counter and Address Generation

The first new circuit block is **Address Generation** circuit, which looks like this:

When you build the address generation circuit, you should test it separately before you insert it in your main circuit. It is quite easy to test it alone, but a little more difficult to test in the circuit.

You **do not need** to record any test results in your report.

The address generation circuit has two separate means of generating address. First, it



can generate an address from the address Register. Second, it can generate an address from the program counter.

First, let us consider the address register. Data is loaded into the address register from the data bus when **LOAD ADDRESS REGISTER** is a 1. This data is put on the address bus when the **USE PROGRAM COUNTERS (PC) FOR ADDRESS** is a 0. This allows a microprocessor instruction to access any desired address by just loading that address into the address register and then putting the address directly on the address bus.

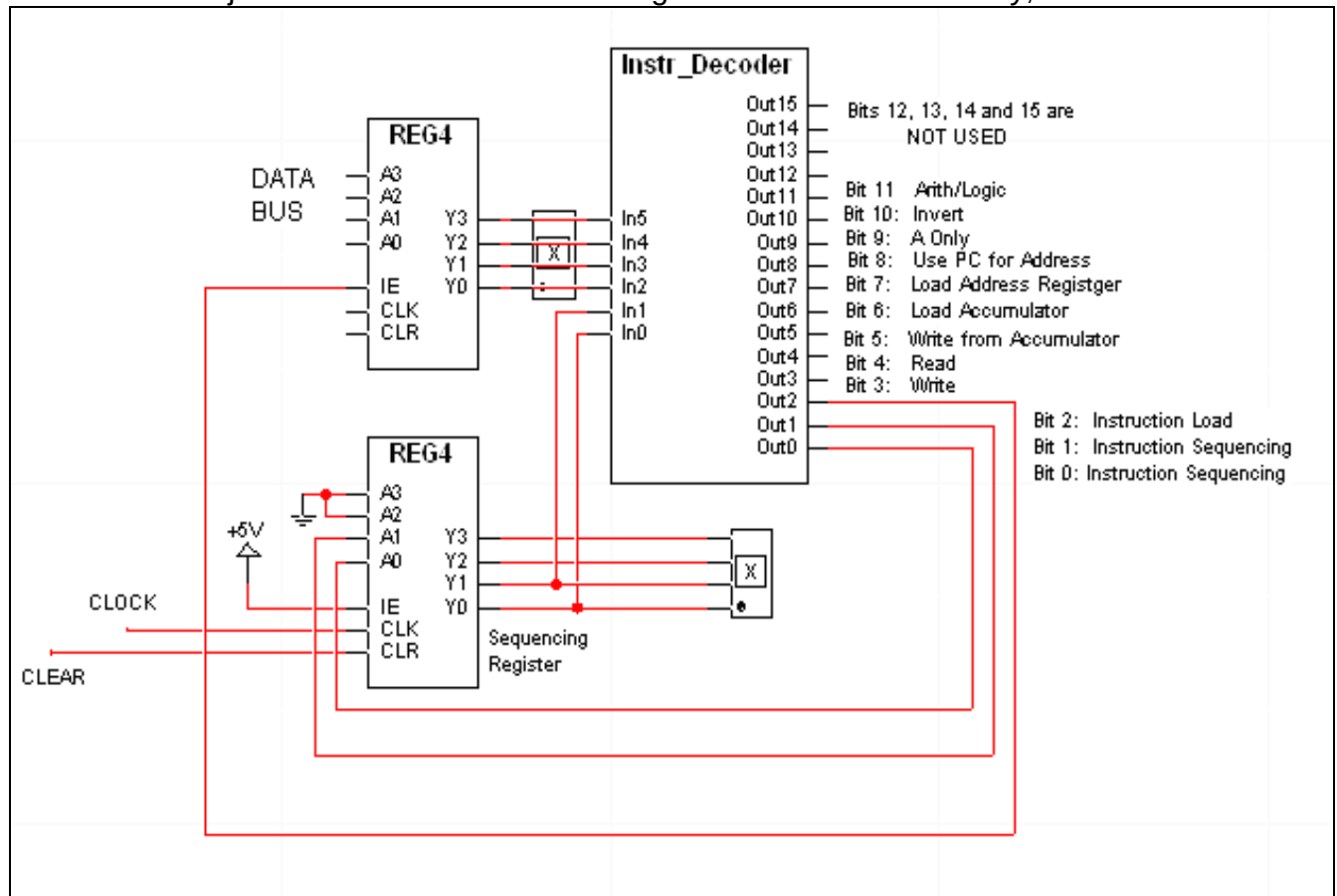
The second way of generating address is from the Program Counter (**PC**). As its name implies, the program counter is only used to access the instruction codes and the instruction data in memory.

The program counter is set to address 0 when the microprocessor is reset using the **RESET** line. (A reset is performed by setting the **RESET** briefly to a 1, and then setting

it back to 0.) Because the program counter starts at 0, the first instruction in the program will be at address 0. Whenever the program counter is used to access an instruction code or instruction data by setting **USE PROGRAM COUNTER FOR ADDRESS** to a 1, the program counter is automatically incremented so that it points to the next address in the program. This causes programs to be executed in the methodical fashion discussed earlier.

Control Circuit and Instruction Decoder

The second major addition to the circuit of stage D is the control circuitry, which we see below:



When you build the control circuitry, you may wish to test it separately before you insert it in your main circuit. The choice of testing it or not is up to you. Either way, you do not need to record any test results in your report.

The control circuitry is the brain of a microprocessor. It tells everything else what to do. It consists of three blocks, which are the Instruction Register, the Instruction Decode, and the Sequencing Register.

The instruction Register holds the current instruction code. As we said earlier, an instruction code is just a number read from memory that tells the control circuitry what needs to be done. Each of the instruction codes in a program is moved in turn into the instruction register so that the instruction can be executed.

The **Instruction Decode** is the block that decodes the instruction and the sequencing bits into the control line settings that will execute the appropriate step of that instruction. The instruction decode is a combinational circuit. It could be built out of AND's, ORs, and NOTs, but in our circuit we will use a LogicWorks ROM. A ROM can be used to give the same truth table as any combinational circuit.

We can use a ROM by connecting our inputs (the Outputs of the instruction register and sequencing register) to the address pins of the ROM, and then connecting each of the control signals we wish to generate to an output. Each combination of inputs generates a unique address in the ROM. We can set the value in that address of the ROM to generate any outputs we desire. This allows us to get any combinational input/output function we desire. Building and modifying circuits using ROMs can be much easier than building the circuit out of individual gates.

The **Sequencing Register** keeps track of instruction steps. Each instruction may have several micro-instructions, and all instructions have at least two micro-instructions (step 00 and 01), which correspond to fetch and execute states as we discussed earlier in our introductory notes on microprocessors way back at the beginning of this lab manual.

Programming the Instruction Decoder

When the sequencing register is 00, we are in micro-instruction 00. 00 is the “fetch instruction” step. Note: “step = micro-instruction”. When you are in step 00, the instruction has not yet been read from memory into the instruction register. Therefore, the instruction decode is set up so that if the sequence register is 00, a new instruction is loaded into the instruction register. The microprocessor is in the FETCH state. After this new instruction is loaded into the instruction register, the instruction decode will also set the sequencing register to 01. The microprocessor is in the EXECUTE state and the 01 micro-instruction will be executed. If the instruction is done after step 01, the instruction decode will set the sequencing register to 00 again to begin the next instruction.

If the instruction is not done after step 01 or has more micro-instructions, the instruction is usually a two-step instruction. Then, the sequencing register will be incremented to 02 to begin the second step (step 02). After step 02 has completed, the instruction decoder will then load 00 into the sequencing register to begin the next instruction. We will make one infinite-step instruction, which we will use to stop our microprocessor. This can easily be done by having the instruction decode load 01 into the sequencing register at step 01. The instruction will then never leave step 01, and never move on to another instruction.

As you can see, there is a lot of interplay between the instruction register, the sequencing register, and the instruction decodes to get the microprocessor to work. Let us look at some sample instructions to illustrate this.

Note:

We will not make any four-step instructions, so this is true. Remember that a four-step instruction would have steps 0-3.

Fetch State:

Let's look at step 00. Since this is a "fetch" instruction, it would be the same for ALL instructions.

When the sequence register is 00, the control lines setting are shown in below table

FOR ALL INSTRUCTION

Output Bits	Control Lines	00	Hex. Value	01	10	11
Bit 1-0	Next Sequencing Bits	01	5			
Bit 2	Instruction Load	1				
Bit 3	WRITE	0				
Bit 4	READ	1	1			
Bit 5	Write form Accumulator	0				
Bit 6	Load Accumulator	0				
Bit 7	Load Address Register	0				
Bit 8	Use PC for Address	1	1			
Bit 9	A ONLY-bar	X				
Bit 10	INVERT	X				
Bit 11	ARITH-BAR / LOGICAL	X				
Bit 12	X	X	0			
Bit 13	X	X				
Bit 14	X	X				
Bit 15	X	X				
			0115			

Let us investigate each of the control line:

1. After the step 00 "**fetch state**", The **Next Sequencing Bits** has to be set to 01 to have the microprocessor in the EXECUTE state.
2. **READ** and **INSTRUCTION LOAD** are set to 1, to allow data transfer into the Instruction Register.
3. Since READ is 1, then **WRITE** has to be set to 0.
4. The accumulator is set to 0 by neither writing nor reading to it.
5. The instruction decoder puts the program counter of the address bus by setting **Use PC for Address** to 1. The Program Counter (**PC**) has to increment.

Since ROM can not store don't care (X's), therefore, all the X's in the above table are set to O's in the instruction decode ROM. If you consider the outputs as a single number (with X's replaced with O's), the output of the instruction decode for these cases is

0000 0001 0001 0101 in binary (reading from bottom to top), or **0115** in hex as shown at the bottom of the table. We will use this number later to make the instruction decode ROM.

Let us now look at the design of some sample instructions. There can be a total of 16 instructions, numbered 0-F. We will give 4-bit code for each instruction that we design. The first instruction 4-bit code is 0000, which is equivalent to 0 in HEX. Suppose we want instruction 0 (which we can say is stored at some arbitrary address Q) to load the data value in the next address (address Q+1) into the accumulator. We could nickname this instruction **LOAD ACC**, short for "load accumulator." This instruction could have the following table implemented in the instruction decode:

For Instruction 0, LOAD ACC

Output Bits	Control Lines	00	01	Hex. Value	10	11
Bit 1-0	Next Sequencing Bits	01	00	0		
Bit 2	Instruction Load	1	0			
Bit 3	WRITE	0	0			
Bit 4	READ	1	1	5		
Bit 5	Write form Accumulator	0	0			
Bit 6	Load Accumulator	0	1			
Bit 7	Load Address Register	0	0			
Bit 8	Use PC for Address	1	1	1		
Bit 9	A ONLY-bar	X	0			
Bit 10	INVERT	X	0			
Bit 11	ARITH-BAR / LOGICAL	X	X			
Bit 12	X	X	X	0		
Bit 13	X	X	X			
Bit 14	X	X	X			
Bit 15	X	X	X			
		0115	0150			

If you consider the outputs as a single number (with X's replaced with O's), the output of the instruction decode for instruction 0, sequencing bits 01 is **0000 0001 0101 0000** in binary, or **0150** in hex. We will use this number later to make the instruction decode ROM.

The instruction above takes two steps: first (step 00) it loads in the instruction and second (step 01) it executes it. When the instruction is loaded in, it comes from

address Q. When the instruction is executed, it accesses address Q+1, and moves the value in that address onto the data bus. The ALU is set so that the value on the data bus passes through it and is stored in the accumulator. This is the instruction we desired. Since the instruction needs only step 00 and step 01, the **Next Sequencing Bits** at step 01 are set to 00. This will cause the next instruction's step 00 to begin when this instruction's step 01 is done. This next instruction will be obtained from address Q+2, because address Q and Q+1 have already been accessed, and the program counter automatically increments. This is a good example of an instruction, which loads data into the accumulator.

For what we want our microprocessor to do, simply loading data is not enough. We want to be able to manipulate it. One thing that would be useful is an ADD instruction. We would want the ADD instruction (at some address Q) to move data from the next location (address Q+1) and add it to the value in the accumulator. The ADD instruction will have a 4-bit code of 0001, which is equivalent to 1 in HEX. This could be done as follows:

For Instruction 1, ADD

Output Bits	Control Lines	00	01	Hex. Value	10	11
Bit 1-0	Next Sequencing Bits	01	00	0		
Bit 2	Instruction Load	1	0			
Bit 3	WRITE	0	0			
Bit 4	READ	1	1	5		
Bit 5	Write form Accumulator	0	0			
Bit 6	Load Accumulator	0	1			
Bit 7	Load Address Register	0	0			
Bit 8	Use PC for Address	1	1	3		
Bit 9	A ONLY-bar	X	1			
Bit 10	INVERT	X	0			
Bit 11	ARITH-BAR / LOGICAL	X	X			
Bit 12	X	X	X	0		
Bit 13	X	X	X			
Bit 14	X	X	X			
Bit 15	X	X	X			
		0115	0350			

If you consider the outputs as a single number (with X's replaced with 0's), the output of the instruction decode for instruction 1, sequencing bits 01 is **0000 0011 0101 0000** in binary, or **0350** in hex. We will use this number later to make the instruction decode ROM.

The ADD instruction is different from the LOAD ACC instruction only in the control bits to the ALU. For the LOAD -ACC instruction, the ALU was in pass-through mode. For the ADD instruction, the ALU manipulates the data before the data is stored in the accumulator.

Another useful instruction might be useful is to read the next byte of data, treat it as an address, and store the accumulator out to that address. We could call this instruction **STORE_TO_ADDR**. It would have the following table.

This instruction has three steps: First, (step 00) the instruction is read in from some address Q. Second (step 01), the instruction loads the value in address Q+1 into the address register. Suppose that address Q+1 contains some number R, where R is in the range of 0 to F. Third (step 10), the accumulator is written out to address R in memory. This instruction can be very useful for writing data to the output device. To write data to the output device, you have to always write to the same memory location. In our circuit, that location is address 0. The program counter is not suited for this, so somehow you have to load an address into the address register as shown in this instruction so that you can write your data out to that address.

For Instruction 2, STORE_TO_ADDR

Output Bits	Control Lines	00	01	Hex. Value	10	Hex. Value
Bit 1-0	Next Sequencing Bits	01	10		00	
Bit 2	Instruction Load	1	0	2	0	8
Bit 3	WRITE	0	0		1	
Bit 4	READ	1	1		0	
Bit 5	Write form Accumulator	0	0	9	1	2
Bit 6	Load Accumulator	0	0		0	
Bit 7	Load Address Register	0	1		0	
Bit 8	Use PC for Address	1	1		0	
Bit 9	A ONLY-bar	X	X	1	X	0
Bit 10	INVERT	X	X		X	
Bit 11	ARITH-BAR / LOGICAL	X	X		X	
Bit 12	X	X	X		X	
Bit 13	X	X	X	0	X	0
Bit 14	X	X	X		X	
Bit 15	X	X	X		X	
		0115	0192		0028	

If you consider the outputs as a single number (with X's replaced with O's), the output of the instruction decode for instruction 2, sequencing bits 01 is **000 0001 1001 0010** in binary, or **0192** in hex. For instruction 2, sequencing bits 10, the output is **0000 0000**

0010 1000 in binary, or 0028 in hex. We will use this number later to make the instruction decode ROM.

One final instruction, which we will consider, is instruction 3, the **STOP** instruction. When the program is done executing, we would like a way to stop the microprocessor so that we can see what data came out of our program. That is what the STOP instruction is for.

The microprocessor can be halted quite simply by just setting the sequencing bits so that they don't **go** back to 00, setting **Use PC for Address** to a 0, so that the program counter is not continually incremented, and setting everything else to a 0 so that nothing is read, written, or loaded. This gives the following table:

For Instruction 3, STOP

Output Bits	Control Lines	00	01	Hex. Value	10	11
Bit 1-0	Next Sequencing Bits	01	01	1		
Bit 2	Instruction Load	1	0			
Bit 3	WRITE	0	0			
Bit 4	READ	1	0	0		
Bit 5	Write form Accumulator	0	0			
Bit 6	Load Accumulator	0	0			
Bit 7	Load Address Register	0	0			
Bit 8	Use PC for Address	1	0	0		
Bit 9	A ONLY-bar	X	0			
Bit 10	INVERT	X	0			
Bit 11	ARITH-BAR / LOGICAL	X	0			
Bit 12	X	X	X	0		
Bit 13	X	X	X			
Bit 14	X	X	X			
Bit 15	X	X	X			
		0115	0001			

If you consider the outputs as a single number (with X's replaced with O's), the output of the instruction decode for instruction 3, sequencing bits 01 is **0000 0000 0000 0001** in binary, or **0001** in hex. We will use this number later to make the instruction decode ROM.

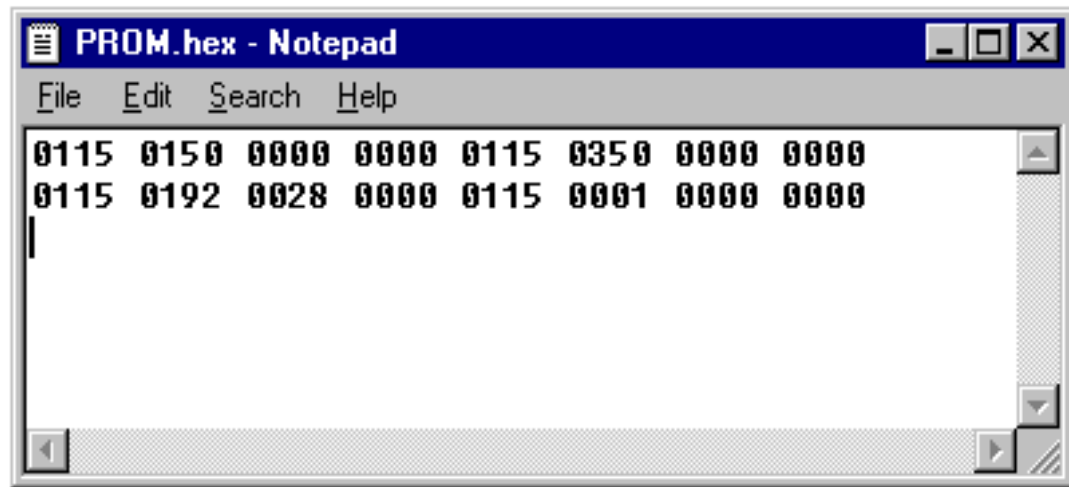
When the STOP instruction is executed, it will get “stuck” in step 01 of the instruction and no further instructions will be executed.

Task E1: Build the Instruction Decode Circuit

Now we need to enter these instruction into the decode ROM.
First we need to create a text file with all the Hex values.

You can create the text file using any word processor or using Notepad in your windows accessories:

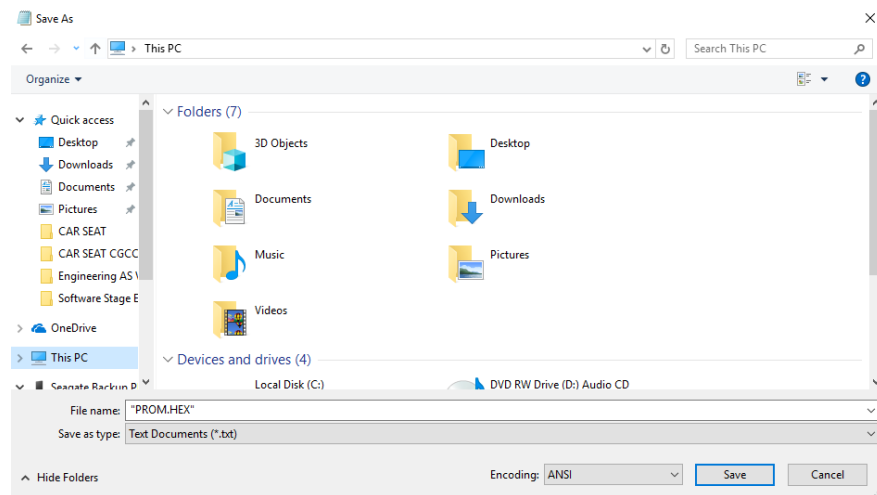
1. Select **Programs⇒Accessories⇒Notepad** and enter the Hex. Values as follow:



In above window, address 0 input to the PROM gives an output equal to the top left number, which is 0115 in hex. Addresses increase along the row, so address 1 gives an output of 0150, address 9 gives an output of 0192, and so on.

Consider instruction 0, For instruction 0, there are four sequencing bit combinations, which are 00, 01, 10, and 11. These combinations correspond to PROM addresses 0, 1, 2, and 3 respectively. So instruction 0, step 0 gives an output of 0115 (the contents of address 0), instruction 0, step 1 gives an output of 0150 (the contents of address 1), and instruction 0 steps 2 and 3 give outputs of 0000 (the contents of address 2 and 3). Each of the steps of instruction 0 occurs in order in the first half of the first row in the table. Therefore, it will have an output of 0115, 0150, 0000 and 0000 for steps 0, 1, 2, and 3 respectively. Instruction 2 takes the first half of the second row, and instruction 3 is the remainder of the second row. Additional instructions could be added starting on the third row.

2. Save the file as text only file with **.hex** extension. Example **PROM.hex**. Make sure you save it to your drive. You may want put double quotes around **"PROM.hex"** to eliminate the txt extension at the end of the file name.

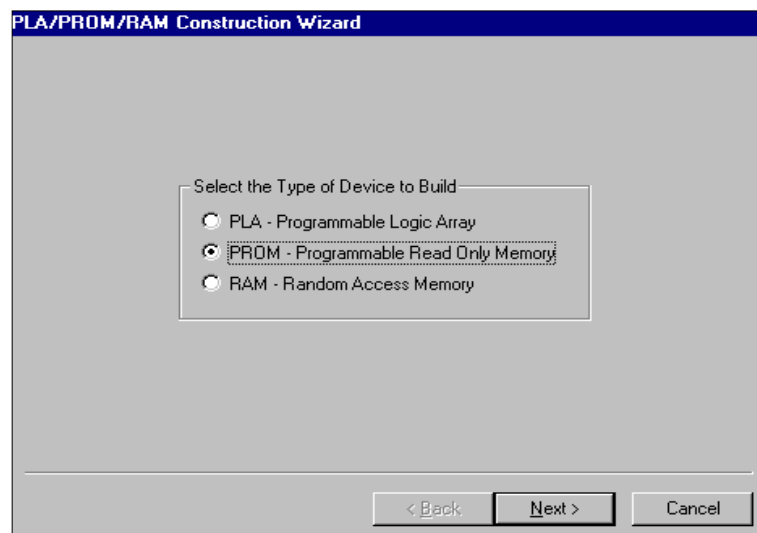


Creating the Instruction Decoder Device

To make a PROM, you need to first create a file that specifies its contents. Since there are six input bits, there are 64 different input combinations. We need to specify what the Outputs are for each of these combinations. This is done as follow:

Through LogicWorks 4 for windows, you have the capability of constructing the instruction decoder in an easy manner.

1. Open the PROM/RAM/PLA Wizard by clicking on its button . In the toolbar or from the Simulation menu, select "PROM/RAM/PLA Wizard". Select the PROM (Programmable Read Only Memory) device type and click on the Next button:



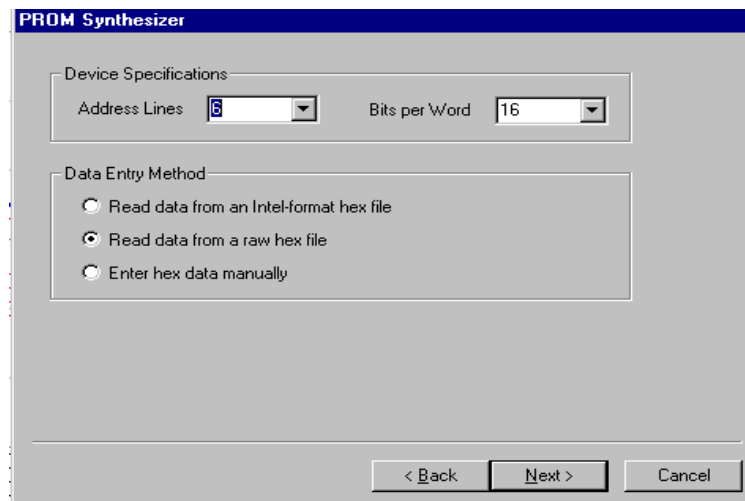
2. Enter the following information: (use the TAB key or your mouse to move from one box to another)

Address Lines: 6 (Number of desired inputs)

Bits per Word: 16 (Because each code consist of 4 Hexadecimal numbers and each Hex. Number is 4 bits wide $4 \times 4 = 16$)

We will enter the appropriate values using two different entry methods:

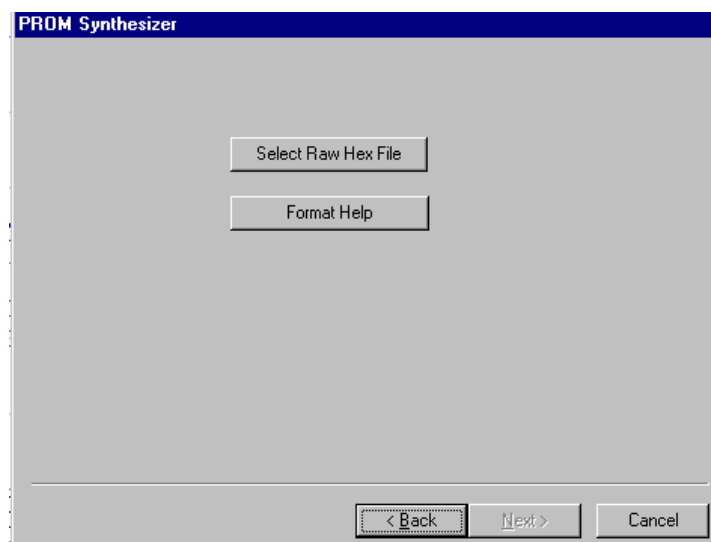
a. Read data from raw hex file



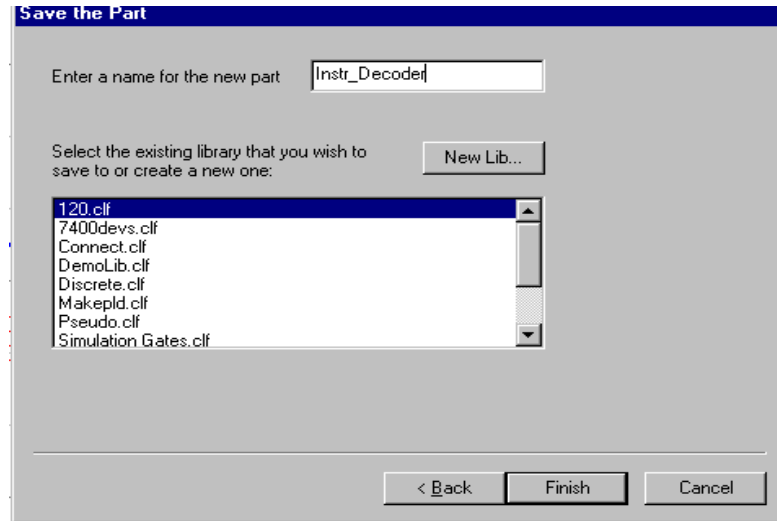
Click on Next after you select the option

“Enter data from a raw hex file”

3. Click on the “Select Raw Hex File”

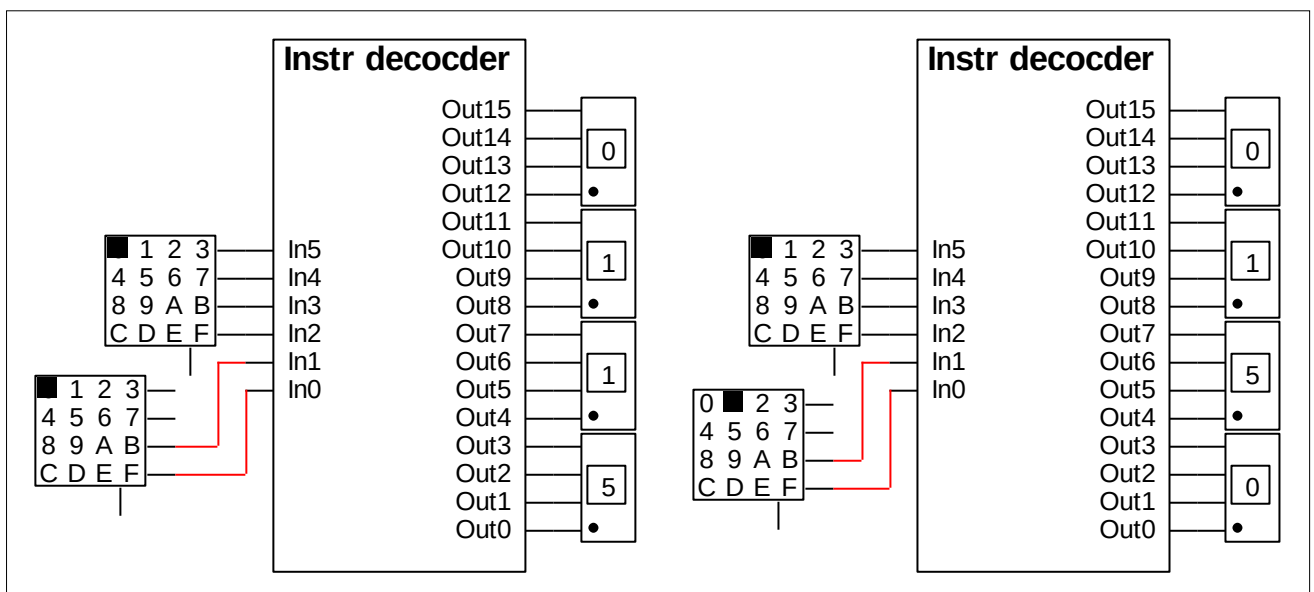


4. Select the PROM.hex file from your floppy A and then click on **Next**.
5. Enter A name for the new device. Example “Instr_Decoder” and save to your library.



Task E2: Testing the Instruction Decode Circuit

Now that you have built your Instruction Decode, test it briefly using the following circuits:



You should do this so that you know that your instruction decoder is working before you put it into your microprocessor, where trouble will be harder to find. ***You do not need to record your results.***

Task E3: Build The Microprocessor!!!!

We have now discussed everything you should need to know to build your microprocessor and have some understanding of it. A term's worth of work is about to pay off. Build the microprocessor circuit.

Task E4: Writing a Program

Now that you have your microprocessor built, you should be very excited to try it out. Do not be too disappointed if it does not work the first time. It may take a little debugging to find wires that did not connect, and other minor problems. Since you know how each element of the circuit is supposed to react, you should be able to track down these minor sorts of problems. If you have serious difficulty that you cannot figure out with repeated effort, take a break, and then come back to it. If you have properly tested everything, this is unlikely to occur.

As a test of your microprocessor, suppose we try the following program:

Address	Instruction	Comment
0	LOAD_ACC 3	Put the number 3 into the accumulator
2	ADD 5	Add 5 to the value in the accumulator
4	STORE_TO_ADD R 1	Store the result to address 1
6	STOP	Stop the microprocessor

In memory, this program would look like this:

Address	Contents	Comment
0	0	The LOAD_ACC Instruction Code
1	3	The number 3 to be loaded into the accumulator
2	1	The ADD Instruction Code
3	5	The number 5 to add to the data in the accumulator
4	2	The STORE_TO_ADDRESS instruction code
5	0	The address 0 to store the accumulator to
6	3	The STOP Instruction Code

This program will add the numbers 3 and 5 and store the result out to the Output device at address 0. Enter the program in your ROM, and then reset your microprocessor. Step through the program one step at a time, 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 the result 8 sent to the output device? If not, you have some debugging to do. If you do have problems, record what the symptoms were what the problem was, and how you found the problem. Note: when you step through a program by toggling the microprocessor clock, it is important you never toggle the clock too fast. You need to give everything in the microprocessor enough time to settle after each clock before you toggle the clock again.

Microprocessor Extensions:

As you might guess, there are many ways to improve on this microprocessor. To begin with, it needs more instructions, and it needs more memory. It is not too difficult to add many new instructions, such as a NEG, NOT, OR, or LOAD -FROM-ADDRESS, because the capability is already in the microprocessor. Only the instruction decode would have to be changed. Other instructions are more difficult, such as instructions to load an address into the program counter, so that a program does not need to execute in order. Such an instruction might be called a GOTO or JUMP. Other instructions might handle subroutines, so that you could JUMP to the Subroutine (Often called JSR), and then the microprocessor would keep a note of where you JUMPed from so that the subroutine could RETURN. Still other conditional jumps might only JUMP if the ALU's carry was 0, for example. These instructions are all found in commercial microprocessors, but they are too difficult for us to try to make them here with limited time

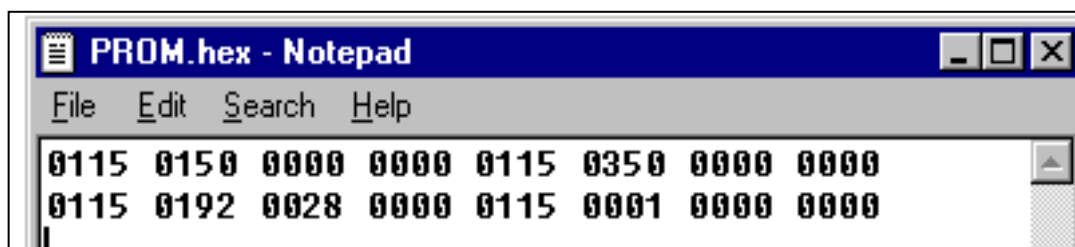
The memory limit in our microprocessor is 16 memory addresses. This is too little for serious work. One thing that is commonly done is to make the address register and program counter twice the size of other registers. When you load the program counter or address register, you would load the upper half of the register first, and then load the lower half second (or vice versa). In these two loads you would fill the entire register. In our microprocessor, this would mean an 8-bit address register and an 8-bit program counter and an 8-bit address bus. This would allow 256 memory locations, which is still not much by today's standards, but is sufficient for some minor work. You should be able to do this extension, since it is fairly straightforward, but we do not require it for this lab.

Task E5: Add NOT, ZERO and SUBTR Instructions

In this task you get to demonstrate your understanding of your microprocessor by adding instructions to your instruction set. Add the 'NOT', 'Zero', and 'Subtract', and instructions to your microprocessor circuit.

- The 'NOT' instruction should "perform a bit-wise one's compliment operation of the value in the ACC and store the result in the ACC".
- The 'Zero' instruction should store the hex digit 0_H 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.

For each instruction above, define the control signals for all the microinstructions by expanding the following table:



Update the contents of the PROM with the added instruction definitions. Then write and execute a few programs to test each of your instructions. Include the programs and their outputs in your lab template along with the microinstruction definition tables for each instruction.

Task E6: invent your own Instruction

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.