EE 306 Supplemental Guide

- IEEE Computer Society -

Contact Info:

| publicitychair@ieeecs.ece.utexas.edu | ENS 131 |

| ieeecs.ece.utexas.edu |

About

This packet was developed by The IEEE Computer Society at the University of Texas at Austin. An essential component of our mission is to expand knowledge and understanding of important topics in Computer Engineering, especially the fundamentals. We have spoken with many students and identified critical sections in the freshman curriculum with which we believe students struggle. This packet is our attempt to address many of those issues.

If at any point you find an error with the packet, please report it on our website at ieeecs.ece.utexas.edu. Indicate the error, the page number, and your name. If the error turns out to be legitimate and you are the first to report it, then you will be entitled to one free beverage per error courtesy of IEEE CS. If you have any questions please do not hesitate to stop by our office and talk to one of our officers. We are always willing to help.

2012-2013 Officers

President	Stephen Pruett
Vice President	Eula Tolentino
Treasurer	Nikhil Garg
System Admin	Hershal Bhave
Secretary	Noah Imam
Publicity Chair	Zain Sheikh
Inventory	Blake Jennings
Corporate Director	David Knopf

Other Contributing Members

Chris Correll	Connor Healy

Contents

Chapter 1

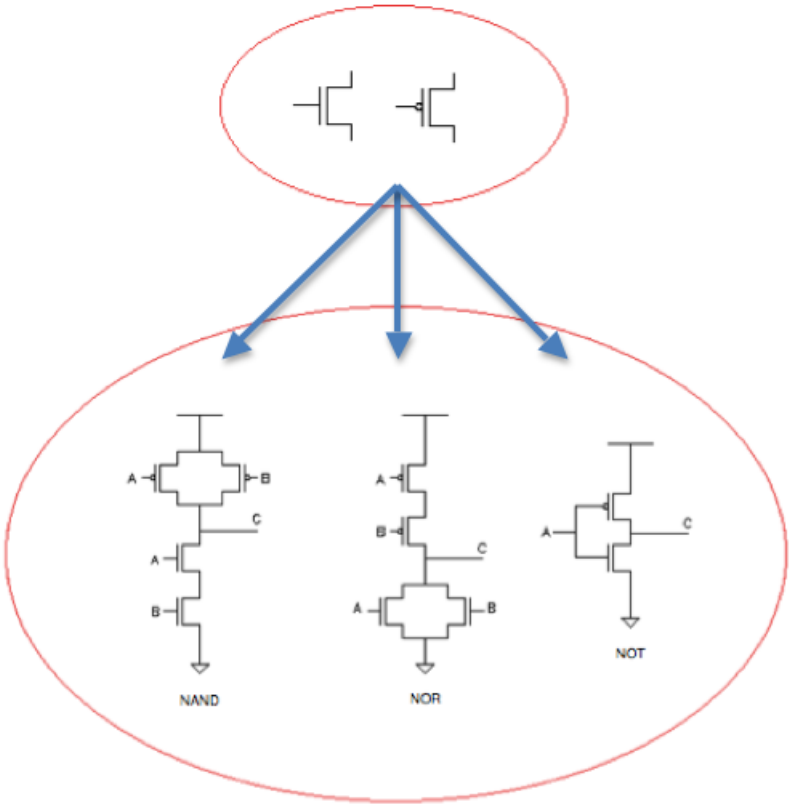
Logic

1.1 The Transistor

Most computers use MOS transistors, metal oxide semiconductor transistors. This is the lowest level of abstraction that we will go over in this class, and we must only know that there are two types, p and n, and their basic properties. In our scope, these transistors operate similarly to switches. For an n-type transistor, if there is a logic '1' at its gate, the transistor acts as a short between the source and drain, and if there is a '0', it acts as an open between the source and drain. Note that for a p-type transistor, there is a logical NOT before the gate such that if there is a logic '0' at that terminal, the transistor will act opposite to the n-type and provide a short between the source and drain, and with a logic '1', will provide an open.

LIGHT BULB EXAMPLE:
Figure 3.1 on page 52 of Intro To Computing Systems shows a simple circuit consisting of 4 "circuit elements": the 120-volt power supply, the wall switch, the lamp, and the wire that connects all the other elements. Now, the goal here is to get current to flow through the light bulb, so that it turns on. The only way to get current to flow through the light bulb is to have a voltage drop across it. That is to say, we need to connect one end of the light bulb to a relatively high voltage, and the other end to a relatively low voltage. Luckily, we have our 120-volt power supply here. Notice that this also has 2 terminals. We refer to the lower terminal as "ground" and we say that it is at 0 volts. By doing this it allows us to reference the upper terminal as 120 volts. If we were to simply connect the 2 terminals of the battery to the 2 terminals of the light bulb, the light bulb would turn on, because we created a voltage drop across it (a drop of 120 volts to be exact). However, such a circuit would be rather useless. We would rather have the ability to turn the light on and off, so we add a switch, as shown in the figure. When the switch is open, the wire will not connect to the bulb and current cannot flow through a broken circuit. Then when the switch is closed current will flow through the bulb and the light will turn on. Now let's throw out the switch and replace it with an n-type transistor as in figure 3.2b of textbook. As you know when you supply a relatively high voltage to the terminal of an n-type transistor, it will act as a piece of wire. This will cause the 120-volt terminal of the battery to be connected to the upper end of the light bulb and the light bulb will turn on. When a low voltage is supplied to the transistor, the circuit will be broken. We usually simplify the drawing to look like the picture in figure 3.2c of the textbook where we remove the battery and simply label the high voltage with a horizontal bar and low voltage with an arrow on different ends of the circuit.

Figure 1.1: Transistor diagrams for common logic gates



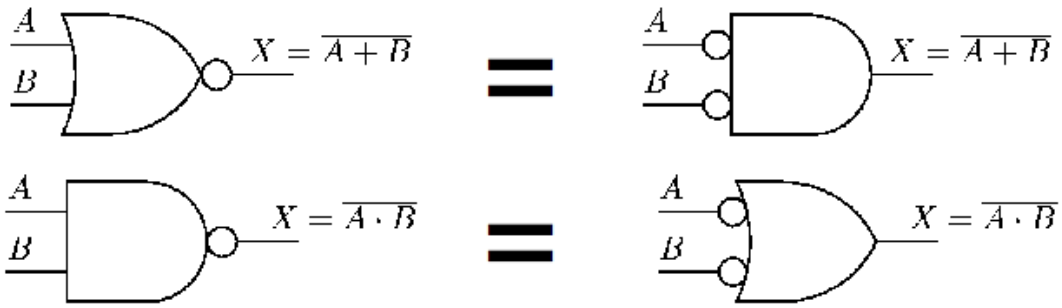
1.2 Introduction to Logic Gates

The next step is to build what are called "logic gates". In chapter 2 of the textbook, you learned all about different logic operations. Now our goal is to implement these logic operations with the transistors. We will refer to a relatively high voltage as '1' and a relatively low voltage as '0'. The implementation of a NOT gate is shown in figure 3.4 of the textbook. Notice that when the input is set to 0, the output is connected to 1 (high voltage), and when the input is set to a 1 the output is connected to 0 (low voltage). The NAND gate shown in figure 3.7 of the textbook is a bit trickier. Consider the AND operation. An AND will only produce a 1 if all of the inputs are a 1. That is to say that if at least one of the inputs is a 0, the output of the AND gate will be 0. The NAND gate is just the opposite. Therefore if at least one of the inputs to a NAND gate is a 0, the output will be a 1. If none of the inputs are 0, then the output will be a 0. When we think of the NAND gate this way, the structure of the transistors makes more sense. We connect the p-type transistors in parallel because one of them being supplied with a 0 is a sufficient condition to connect the high voltage to the output. Similarly, we connect the n-type in series because it is necessary for all of the inputs to be a 1 for the output to be connected to 0. Notice that if all the inputs are 1, none of the p-type transistors will be connected; therefore the connection to the high voltage will be broken. Also notice if even one of the inputs is 0, then the connection to ground will be broken at some point too. This means that it will never be the case that the output is connected to both 1 and 0.

We NEVER want to connect the output to both 0 and 1. Similarly, it is possible to create a circuit in which out is not connected to either a 1 or 0. In general, this is also a situation we would like to avoid. We say that "out floats" referring to the fact that it is not at any particular voltage, the output is just sitting there floating in space. The last piece of curious information about this section is the construction of the AND gate. The design is to build a NAND gate and then feed the output of the gate into an inverter, thus forming an AND gate. This begs the question, "Why do we not just connect the n-type transistors in series to the high voltage and the p-type transistors in parallel to the low voltage?" The issue with this is beyond the scope of this course; however it is worth mentioning to put the curious mind at rest: This is not a valid circuit due to the makeup of the materials of the transistors. This is not an important concept to understand now, it is just making clear that the design in the book is valid.

DeMorgan's law
DeMorgan's law is a very important concept that will not go away. I will leave the textbook to discuss it in detail and I will focus on a few tricks that you can use to help apply DeMorgan's law in designing structures out of logic gates. Look at the pattern that occurs when we apply DeMorgan's law to gates (figure 3.8 of the textbook). We see that an AND gate is equivalent to an OR gate completely surrounded by bubbles. Similarly, we see that an OR gate is equivalent to an AND gate completely surrounded by bubbles. We can use this fact to complex gate structures.

Figure 1.2: Demorgan's Law with Logic Gates



Combinational and Sequential Logic Circuits
Just as we used the transistors to build the logic gates, we can now raise the level of abstraction even higher and use our gates to build more complicated logic circuits. Note that we could build the logic circuits directly out of the transistors, however this would be difficult and tedious. Using the abstraction of the logic gate makes the job MUCH simpler. There are 2 types of logic circuits: combinational and sequential. We will first go over combinational then return to sequential.

1.3 Combination Logic

Combinational circuits are all about the "now". That is to say, the output will reflect whatever the inputs happen to be at that time. The past inputs will have no effect on the output. Common

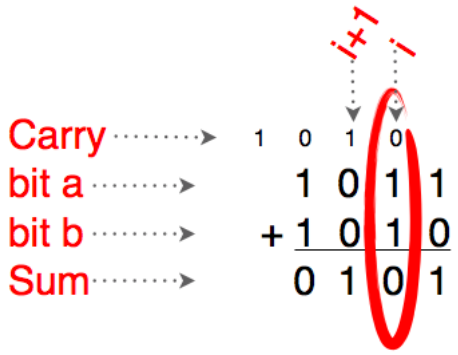
combinational circuits include: a decoder, a mux and a full adder.

Decoder
The purpose of the decoder is, as its name suggests, decoding things. For example, say we have 8 devices, and we also aren't very creative, so we call the devices, device 0, device 1, device 2, ect. Now say we give the user 3 wires, and we tell them to drive a signal on those wires to select the device they would like to turn on. As you know, to uniquely identify 8 different things we need 3 bits, hence the 3 wires. The user will drive a 000 if he wants to refer to device 0, a 001 if he wants to refer to device 1, a 010 if he wants to refer to device 2 and ect. Now if only there were some sort of combinational logic circuit to take the 3 inputs in binary and convert it into 8 outputs, which corresponded to the 8 unique representations of 3 bits. This is where a decoder comes into play. If the input 011 is sent into the input wires then the 3rd AND gate will produce a 1, if the input 000 is sent into the input wires then the 0th AND gate will produce a 1.

Questions:
Easy: What do the inputs need to be to activate device number 6?
Medium: Could it ever be the case that more than one of the devices is activated at once?
Medium: if the output of the 7th AND gate is a 1, what is the output of the 4th AND gate?
Hard: Add 1 gate and some connections to a decoder to change it in the following way: there will now only be 1 output (as opposed to 8), if the number driven on the input wires is even, make the output a 0, if the number driven on the input wires is odd make the output a 1.

Mux
Now a different situation: say we have 2 devices, device A and device B. Each of these devices has a 1 bit output either a 1 or a 0. Sometimes we want to read the output from device A and other times we want to read the output from device B. This is where a mux comes into play. Based on what the select signal is we can choose between device A and device B.

Figure 1.3: Addition with outputs labeled



Full Adder
Think about the addition of two binary numbers, 1011 + 1010. As you've learned, it's just like regular addition from elementary school where the numbers are added by individually adding their digits including any carries from the previous digit's sum. Let's take our example 1011 + 1010: think about how you can separate this addition into 4 separate sums, each sum with three inputs.

Two inputs would be the two numbers being added and the third input would be the value of the carry bit. The output of each sum would be the addition of all three inputs. We can design an adder using the same idea of separating digits. To better understand the flow, we'll follow the second digit position of bits as we design the full adder.

The best way to design this is to first draw out a truth table as shown below in Figure 1.4 This

Figure 1.4: Truth Table for addition

a_i	b_i	$carry_i$	$carry_{i+1}$	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

truth table should include all inputs (bit a, bit b, and carry bit), as well as all the outputs (the next carry bit and the sum). The next thing to do is to set up an array of AND gates in the same format as a decoder. **Why 8 AND gates?** As mentioned above, a full adder has 3 inputs, therefore, there are 8 unique combinations of those inputs, so 8 AND gates will be needed to portray all 8 combinations. Just like a decoder, when a particular sequence of bits comes in, the AND gate that corresponds to that sequence will output a 1 and all other AND gates will output 0. Now look at the number of outputs you want to have. Since we want to have 2 outputs (a sum output and a carry output) this means that we want to have 2 OR gates. **Why 2 OR gates?** Remember, we have 8 combinations, as conveyed by the 8 AND gates. When some of these AND gates produce a 1, we may want our sum output to be a 1, and when others are a 1, we may want the carry output to be a 1. This is because the carry and the sum bits do not necessarily use the same AND gates to determine their value. For example, when the input 010 is received, we have the output set to S=1 and C=0. Or when the input is 110, we have the output set to S=0 and C=1, as marked on the truth table above. Remember that an OR gate will produce a 1 if any of its inputs are a 1. So based on the truth table values, we know that we want the carry-bit OR gate to be a 1 when we get inputs 011, 101, 110, or 111. Similarly, we want the other OR gate, the one that outputs the sum, to be a 1 when we get inputs 001, 010, 100, or 111. Now we can connect the wires. Again, to connect the wires, simply refer to the truth table. If the sum output shows a one, then we look at what inputs caused that one, and connect the corresponding AND gate to the OR gate labeled "sum". Similarly done with the carry output. If neither of the outputs is a 1 then we leave the corresponding AND gate unconnected to the OR gates. To see the full adder implemented, refer to figure 1.5. Now that we have designed the circuit to add up a single digit we just need to connect them in such a way to add numbers with multiple digits. We will abstract our circuit with a box labeled full adder so that we have a simpler time drawing the completed diagram, which is shown in the next figure, 1.6 on page 6.

Figure 1.5: A standard full adder

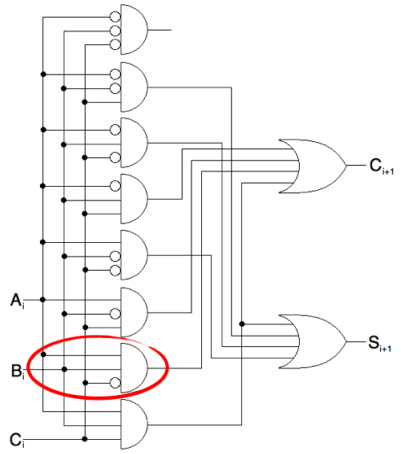
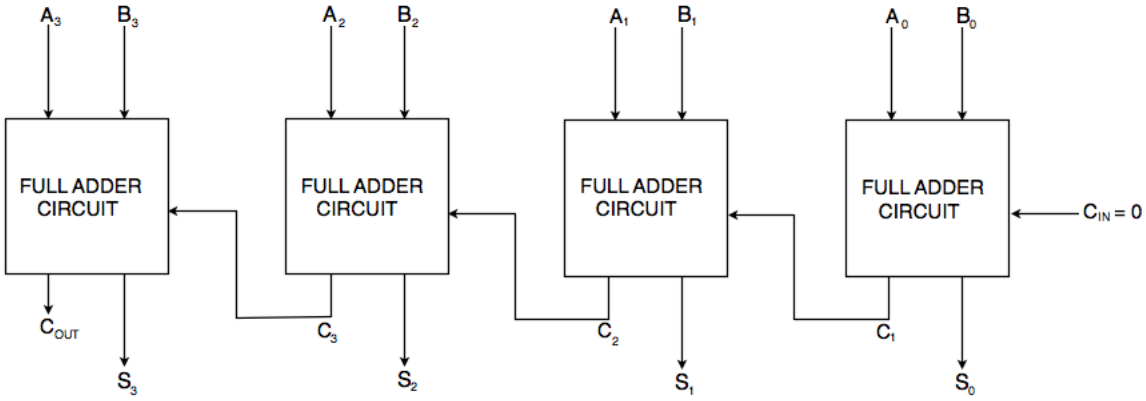


Figure 1.6: The full, abstract logic for a 4 bit adder



PLA

The concept that we used to design the full adder can be generalized into a broader topic called Programable logic arrays or PLA for short. If we have n inputs, we will need 2^n AND gates to uniquely identify all of the combinations of the inputs. This array of AND gates will form a decoder, whose outputs will feed into an array of OR gates. The number of OR gates is equal to the number of outputs. The connections between the AND array and the OR array will be based on which truth table rows have an output of a 1.

Because we can implement any truth table out of AND, OR, and NOT gates (using the above algorithm), we say that AND, OR, and NOT is "logically complete". Meaning that any logic function can be described using only AND, OR, and NOT operators (as opposed to using NAND, NOR, or XOR in addition).

Q: Are there any other sets of gates that are logically complete?

Q: Implement the following truth table using this method.

a	b	c	z_1	z_2	z_3
0	0	0	1	1	0
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	0	0
1	1	0	1	0	0
1	1	1	0	0	1

Q: Implement an AND gate out of a mux.

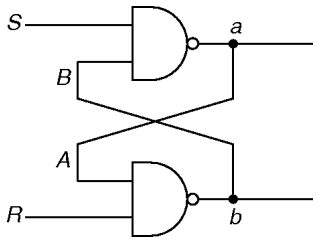
1.4 Sequential Logic

Sequential circuits are circuits whose output is affected not only by current inputs, but also previous inputs. We do this by saving information about the inputs that have already been applied to the circuit. We call his information the "state" of the circuit. State is a very important and simple concept to understand. The classic example is a vending machine. Imagine you are trying to buy a snack out of a vending machine that costs 75 cents. For now, let's say that the machine only accepts quarters. Initially, the machine would start in a fresh state, meaning that no quarters (inputs) have been received yet. Then when you put in a quarter the state of the machine would change from the "no money" state to the "one quarter" state. Likewise when you put in the next two quarters the state would change from the "one quarter" state to the "two quarter" state and then to the "three quarter state". If the vending machine had been a combinational circuit, then you would sit there putting in quarters all day, but the machine would never realize that you had put in all three quarters. This is because combinational circuits have no memory, they only see that a quarter was input, they wouldn't see that a "third" quarter or a "second" quarter was ever input. We will come back to the concept of state, but for now I just wanted to illustrate that, unlike combinational circuits, sequential circuits have a memory to them.

The R-S Latch

How do sequential circuits have a memory to them you ask? Well, we use what is referred to as

Figure 1.7: A standard RS Latch



a "latch". A latch is a circuit that can store a piece of information. The book talks about the R-S

latch. The name R-S comes from the fact that the circuit has 2 inputs, reset and set, or R and S. The circuit has one output, which is read from the output of the upper NAND gate. **Note that the value that is output by the latch is the same as the value contained in the latch.** The circuit is illustrated in the figure above. Note that the output of each of the NAND gates are fed into the input of the other gate, creating a circular path for the signal to travel, thus it is preserving a bit inside the latch. The values of R and S are normally held at a 1 by default (this is called quiescent state). This is because of how a NAND gate behaves when one of its inputs is a 1. First consider how the AND gate behaves, if we have a mystery input called Q, where Q can either be a 1 or a 0, Then what is the value of Q AND 1? The answer is always the same as whatever the value of Q is. This concept was covered earlier in the "masking" section of the book. Well a NAND gate is just an AND gate with an inverted output, so the output of Q NAND 1 will always be the opposite value of Q (NOT Q or Q-bar). So if both R and S signals are held at a value of 1, then the value inside the latch will not be affected, it will simply be inverted every time it passes through one of the NAND gates, which is ok. Now consider what happens to a NAND gate when one of its inputs is a 0. If we feed in Q AND 0, the output will always be 0, therefore Q NAND 0 will always have an output of 1. Now consider what will happen to the state of the latch if initially both R and S are at 1, but then we change the value of S to 0. Initially we cannot predict the value of what was contained in the latch, so we will call it Q for now. The upper NAND gate will have inputs of Q NAND 0, which as mentioned results in an output of 1. This means that the value in the latch is now a 1, regardless of whatever value of Q it previously held. If we follow this new 1 down to the lower latch, the two inputs will be 1 NAND 1. Because R is still being held at a 1 (its quiescent value) the output will just pass on the input and invert it. So the output of the lower NAND gate will now be 0. After we have held S at a 0 for an amount of time, we will raise its signal back up to 1. Note that doing this does not affect the value of 1 contained in the latch. This is why the upper signal is called set, because if it is brought to 0 for an amount of time, then the value contained in the latch will change from whatever value it used to be to a 1. The lower signal is referred to as reset and will work similarly, except it will change the value in the latch to 0. Setting both S and R to 0 is invalid and will result in unpredictable behavior. See the truth table below for an overview of the R-S latch operations.

Gated-D Latch
Rather than have to worry about whether to set the S or the R signal, it would be better if we could just specify to store a 0 or a 1 in the latch. This is where a gated D latch comes into play. D and G are inputs into the latch. D is the value you intend to store in the latch and G is the gate signal. If you want to store the value of D in the latch you first need to "open the gate" or set G to a 1. Setting G to a 1 could be referred to as "enabling the latch". Notice if G is a 0 it does not matter what the value of D is, the S-R latch will not change. If G is a 1 and D is a 1, then the S signal will be brought to 0, and a 1 will be stored in the latch. If G is a 1 and D is a 0 then R will be brought to 0 and the state of the latch will change to 0.

1.5 Applications of Logic

Register These gated D latches can be lined up to store a sequence of bits. We can tie a single enable line to all of the G signals. This will form a register with a single enable signal. When the enable signal is brought to a 1 the value is let into the register.

Memory

Figure 1.8: A standard RS Latch truth table

INPUTS				OUTPUTS		OPERATION
S	R	A	B	a	b	
0	0	X	X	1	1	undefined
0	1	X	X	1	0	set state
1	1	1	0	1	0	no state change
1	0	X	X	0	1	reset state
1	1	0	1	0	1	no state change

We can now stack the registers to form multiple locations in which we can store values. These multiple locations each have a specific identifier known as its address. The number of bits in each location is the addressability. Say we stack 8 registers, each register can hold 16 bits, i.e. it contains 16 gated-D latches. We can use the decoder we learned about earlier to select which register we would like to enable. Because there are 8 locations we need a 3-8 decoder. This whole structure is considered to be memory. The 3 lines feeding into the decoder are called the address lines. We can use them to select which memory location we want to refer to. The number of unique locations in a block of memory is called the address space. There are 2 operations we would like to perform on memory locations, write values to them and read them at a later time. To read a value at a certain memory location, we can take the output of each location and AND it with the corresponding output from the decoder. This means that the only values that will be read will come from the location that was selected by the decoder. To write a value at a certain location, we set the inputs to the registers to the value we want to store and AND it with the output of the decoder (which provides where we're going to store the value). Unlike reading a value, however, we must set the WE (write-enable) signal to a 1 before our value is stored inside the register.

1.6 Finite State Machines

Finite state machines tie together all of what we've learned so far. Let's go over this chapter from the beginning: we have transistors, and from transistors we can create logic gates that can perform our logical operations. From the logic gates, we can create combinational and sequential logic circuits. Combinational circuits, such as decoders and muxes are mainly used for selection; similar to a full adder, we can use a decoder and OR gates to create a programmable logic array (PLA) that can do any logical operation we want. Sequential circuits such as R-S latches and gated-D latches (which are just R-S latches with 2 added NAND gates and a write enable signal) are used for storing bits. Line up your gated-D latches and you have a register. Stack up your registers, add some combinational logic (i.e. decoders and muxes), and you've got memory. From reading the above sections, we know it's not as simple as this brief overview, but this is meant to show the flow of

abstraction from transistor to memory. So now that we've got memory, what happens next? With a FSM, we have four parts to understand:

1. Combinational Logic Circuits: "decision" elements
2. Sequential Logic Circuits: "storage" elements
3. States and State diagrams: representation of a machine; describes the machine's different states and shows each state's important values
4. Clock: trigger for the transition from one state to the next We've gone over combinational and sequential logic circuits and how we get memory from the two, so the next order of business is the concept of state and state diagrams.

State Diagrams
Because of the memory, we must now be able to track all possible events/situations that can happen with our machine. We are no longer just keeping track of the current input but past information as well so we use the concept of state and state diagrams to understand and keep track of the system. Otherwise, we'd have different inputs, different outputs, different choices, and a plethora of possibilities to mentally keep track of. It may be important to note how our daily lives highly resemble the work of computers. In a sense a person's life is one giant state machine. Let's take your life, for example. Imagine every important/memorable moment in your life so far. Your memories of those moments are like frozen depictions of important events. You remember particular details about those memories. Those memories would be the different states in your system, and the details of each memory are what you use to describe each one and differentiate one from other memories. Similarly, we need ways of differentiating states; therefore, each state will have a complete set of properties that describes itself. These properties can be stored using various storage devices, like the ones discussed in previous sections of this chapter. Each state connects together with all other states to create a flow of how the system should operate. This is a representation of the finite state machine showing all of the FSM's important events and each event's information that leads to an output or the next event. In this class, we will be focusing on Moore machines, which are FSMs whose outputs depend only on the current state you are in. This may sound confusing so let's take a look at a simple state diagram (see Figure 1.6) of a state machine and see what we can find out about the system. S0, S1, S2, and S3 are the labels of each state and they will each have a binary number that uniquely identifies them. How do we know how many bits each state can be identified as? Because we have 4 states, we know that each state can be described using 2 bits, because 2^2 is 4. Now we know that each state has a corresponding value identifying it, what is that value? Really, we can make it whatever we want. S0 can be 00, 01, 10, or 11. Each state just needs to be uniquely identifiable. In later classes, you will see how assigning certain bit values to certain states can actually optimize the performance of your machine. But for now, we will make it simple by making S0 be 00, S1 be 01, S2 be 10, and S3 be 11. Let's look at a particular state. The number separated by a backslash from S0, in this case '1', is the output of the machine when it is in that state. We put the output inside the state, because since it's a Moore machine, the only thing affecting the output is the state. The numbers corresponding to the arrows are inputs. What does this state mean? When you are in state S0:

1. You have a present output of 1
2. If you get an input of 0, your next state is S0
3. If you get an input of 1, your next state is S2

Figure 1.9: The State Machine referred to above

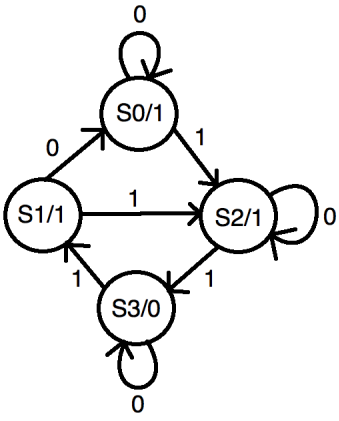
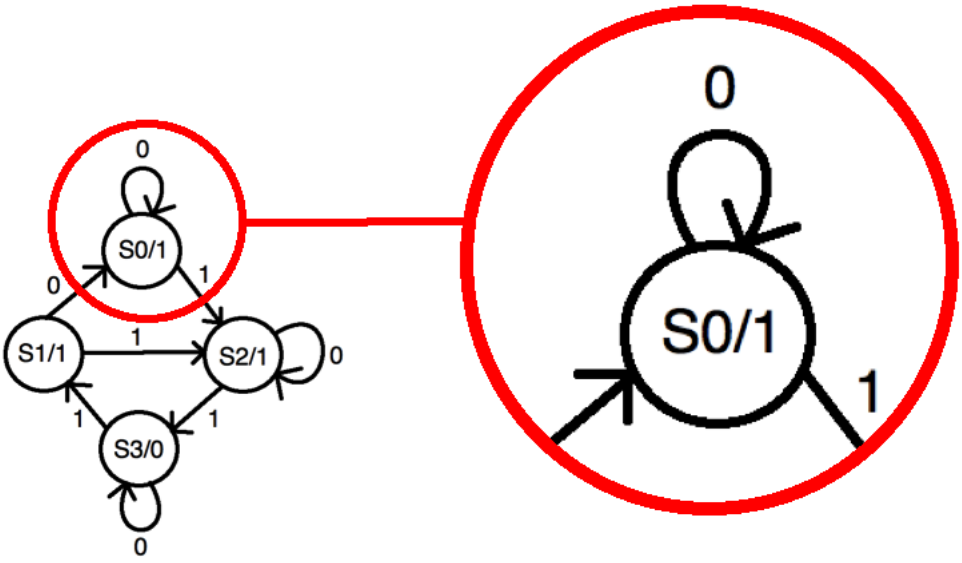


Figure 1.10: An in-depth view of State 0



How do we make sense of this state diagram? When we have a state diagram, we can make a truth table by analyzing its parts. From above we know everything about S0, so we can put that in the table, and we do this till our table is filled out. Then, we can put our bit value assignments of each state into the table. Refer to Figure 1.11 on page 12.

Now that we can comprehend states and state diagrams, how is this actually implemented? The first thing to take into consideration is how you can go from one state to the next.

The Clock

The clock is a signal that alternates from 0 to high voltage (or 0 to logic '1'); see Figure 1.6 below. The state of our system changes at the clock cycle's rising edge, marked by the arrows on Figure 1.6 on page 13.

Figure 1.11: The truth table described previously

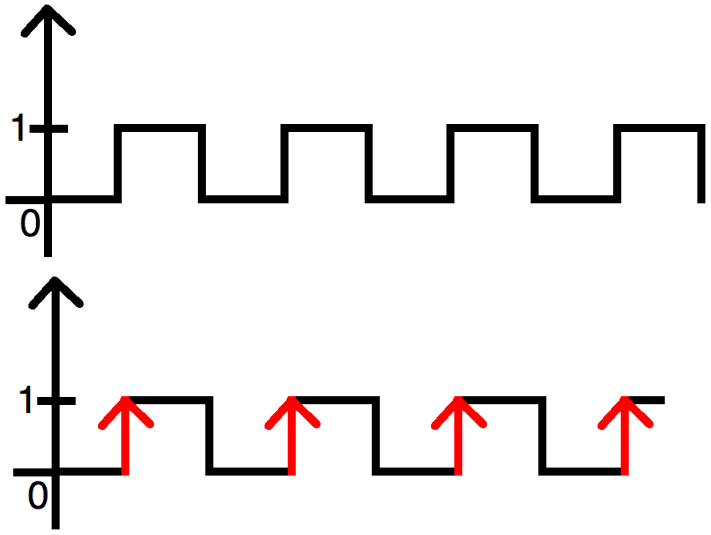
Present State	Next State X = 0	Next State X = 1	Output
S0	S0	S2	1
S1			
S2			
S3			

Present State	Next State X = 0	Next State X = 1	Output
S0	S0	S2	1
S1	S0	S2	1
S2	S2	S3	1
S3	S3	S1	0

Present State	Next State X = 0	Next State X = 1	Output
00	00	10	1
01	00	10	1
10	10	11	1
11	11	01	0

Taking a look at the filled truth table on Figure 1.6 on page 12, you notice that X, the input signal, determines which state the machine will be in next. The clock is a way to synchronize the machine's movements so that even though signal X may be ready to change the machine to the next state, the machine will not change states until the clock cycle is at a rising edge (the moment clock changes from logic 0 to 1). It is important to remember that state change is dependent on both the clock and input, X; the clock must be at a rising edge and X must be present to know which state the machine will actually go to.

Figure 1.12: A clock signal, which oscillates back and forth between logical 0 and 1. State changes occur at a rising edge, marked by the arrows.



1.7 LC-3 FSM & Data Path

The LC-3 (Little Computer 3) is a computer whose data path is shown in Figure 3.33 of the textbook. Notice the Finite State Machine inside? The state diagram displayed on page 568 (Figure C.2) of the textbook represents the routes throughout the data path. It may be intimidating, but imagine how you would go about tracking signals throughout the data path. You can imagine the data path as a city, and a signal as a person. The bus is the subway that people (signals) use to get around to the different places. The state diagram is like a map. Depending on what you want to do, the diagram guides you where to go in the data path and at what time. As we discussed, the diagram represents every possible event, or state, that the machine can be in. You can think of the FSM as the control unit of the whole system. The functionality of the LC-3 will be explained in much greater detail later so don't worry so much that you may not understand everything. Our purpose in this section is to show you how all of the structures within the data path are made from logic structures that you learned in this chapter, how they are synchronized together, and how you can use these tools to specifically analyze each part of a whole machine.

1.8 Logic Practice Problems

Problem 1: Combinational Logic (Programmable Logic Arrays)
Question: Given the truth table in 1.13, can you design a Programmable Logic Array that implements the truth table– obtaining the desired output from each given input combination? Note: A,B,C are each 1-bit inputs.

Figure 1.13: Truth Table for Problem 1

A	B	C	Out
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Problem 1 Solution
If you see a question like this, your first move is to "crack the truth table". Every truth table can be expressed through a truth statement formatted OUT = [expression], where the expression involves your input and any combination AND, OR, XOR & NOT. For instance if you were given the truth table in Figure 1.14, you could tell me that OUT = (A AND B).

Figure 1.14: Another truth table example

A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

Again, you are looking to "crack the truth table" or create a statement that works for all input/output combination. This is the underlying idea of programmable logic arrays, in that you can create one for any truth statement you desire.

So let's look at our truth table. What do you notice? Well, perhaps not much. With three inputs you often want to look at relationships between two inputs, or to catch on to obvious patterns. Here, you can see that anytime the input C is a 1, the output is a 1. What does this tell us? In our truth table we must have some relationship between A and B and then have that result OR'd with C, because anytime C is a 1, out is a 1, but anytime C is 0, the output is contingent on only A and B. What I just said is very important. If we know that, anytime that C is 0 the output is only contingent on A and B. So now we can look at the four cases where C is a 0 and look for a two variable relationship so that we can complete our truth statement. Let's look at what we're left with:

Figure 1.15: Simplified truth table

A	B	Out
0	0	0
0	1	1
1	0	0
1	1	0

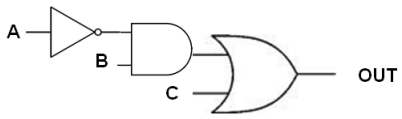
We expect some kind of AND relationship, given OUT is 0, except for 1 case. (A AND B) makes sense but only if cases (A=0, B=1) and (A=1, B=1) was reversed. Let's try using the NOT modifier. If we try (A and notB), on case 2, we see it doesn't work as 0 AND 0 is not 1. If we try (notA and B), we see that all 4 input combinations work for the desired output.

To put it all together, we can now view the truth table as a whole, remembering what we said about C to arrive at the following statement:

OUT = [(A AND B) OR C]

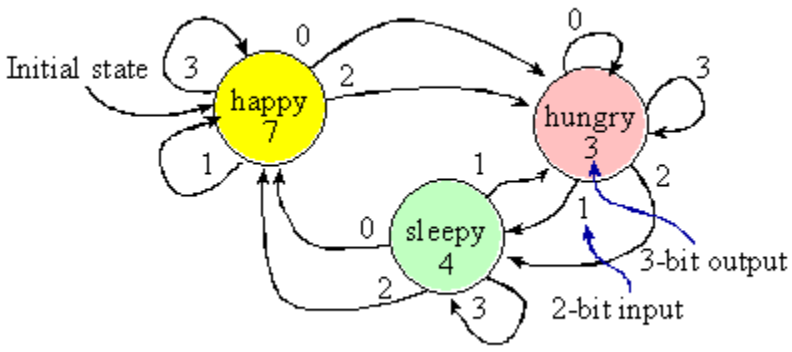
Cracking the truth table is definitely the hardest part of the problem. In order to draw the PLA, we literally just draw what the truth statement is saying using the gate symbols that we are familiar with:

Figure 1.16: Logic Gate representation for problem 1



Problem 2: The Finite State Machines
The combination of evaluating inputs and outputs results in the construction of a finite state machine. More importantly, it should be known that there are a specific number of inputs, outputs and states in a FSM. In a diagram of this system, bubbles represent states. Typically, these bubbles will have labels in them, which indicates the name of the state. Also, linking these individual states are arrows. These arrows indicate the inputs that can be taken to arrive at the next state, so for this reason they can be thought of as pathways between states. For Moore machines, the output is displayed inside each state, and in our example they are underneath the name of the state.

Figure 1.17: FSM for Problem 2



Let's take a look at this example. If you arrive to the happy state, you will always output 7, if you arrive to the hungry state you will always output 3, and if you arrive to the sleepy state you will always output 4. Try to fill in the data for all the combinations in the Moore machine above. (Hint: there are 12 combinations. One is done for you.)

Current State:	Input:	Next State:	Output:
Happy	**0**	**Hungry**	**7**

Problem 2 Solution
Explanation/Summary – The "input" refers to the numbers corresponding to all outgoing arrows

Moore Machine:

Current State:	Input:	Next State:	Output:
Happy	**0**	**Hungry**	**7**
Happy	1	Happy	7
Happy	2	Hungry	7
Happy	3	Happy	7
Hungry	0	Hungry	3
Hungry	1	Sleepy	3
Hungry	2	Sleepy	3
Hungry	3	Hungry	3
Sleepy	0	Happy	4
Sleepy	1	Hungry	4
Sleepy	2	Happy	4
Sleepy	3	Sleepy	4

from the current state you are in, which means it is the input to the next state. The "next state" means what its name suggests. It is the next state given the input received. The "output" refers to the output of the current state, which, because this is a Moore machine, is the number corresponding to the current state. Note: The output does not refer to the next state's output. You have not reached that state yet to output its value.

Problem 3: Finite State Machine
American Airlines flight 306 is utilizing a finite state machine to control the actions of the plane. Duplicate the FSM according to the description provided as follows: The plane is always in one of four states: TAKEOFF, CRUISE, EMERGENCY, and LANDING. The plane is initially in TAKEOFF mode, and it remains in this mode as long as it has not reached its optimum altitude. In TAKEOFF mode, the seat belt sign is lit and the safety video is played. Once at the optimum altitude, the state switches to CRUISE mode, where the flight attendants serve drinks. While in CRUISE mode, the flight can drop below the optimum altitude, in which case the state will switch back to TAKEOFF mode. If the plane sees an obstruction at any time in its path, it will engage an EMERGENCY sequence, and it will remain there as long as it observes the obstruction in its path. In this sequence, oxygen masks will eject from the ceiling. Once the obstruction has cleared, it will re-enter into CRUISE mode. The gradual LANDING sequence is initiated if the destination is in sight, or if the plane is on a collision course. This sequence necessitates the seatbelt signs to be lit, and the cabin lights to be dimmed. A plane is declared to be on a collision course if it has previously been in an EMERGENCY mode, and the obstruction is too close to be avoided. Note that if the plane is in the LANDING sequence and it notices an obstruction, it will continue to land, but away from the obstruction. Take Note: the final state is LANDING, the first state is TAKEOFF.

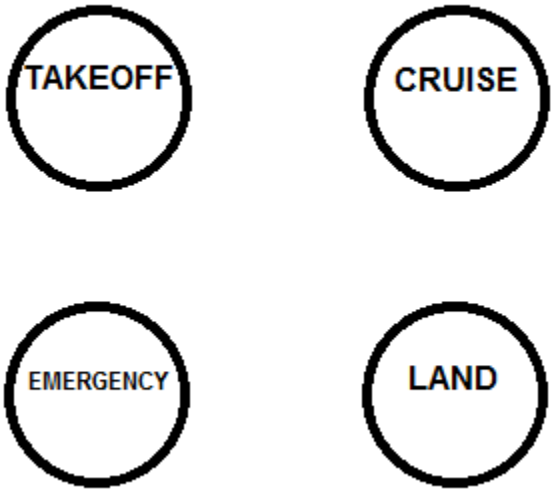
1. Assign attributes to the following 3-bit inputs. All may or may not be used.
   000 - collision course
   001 -
   010 -
   011 -
   100 -
   101 -
   110 -
   111 -

2. Assign attributes to the following 2-bit outputs, and then fill them in correspondingly in the diagram below. Note that seatbelts and the safety video are grouped into one output, and seatbelts and dimmed lights are grouped into another output.
   00 -
   01 -
   10 -
   11 -

3. Using the inputs assigned in part a, draw arrows on the diagram in part-b corresponding to the correct change in states

4. Complete the truth table given below containing five scenarios of the correctly implemented FSM.

Figure 1.18: Problem 3.3 State Diagram



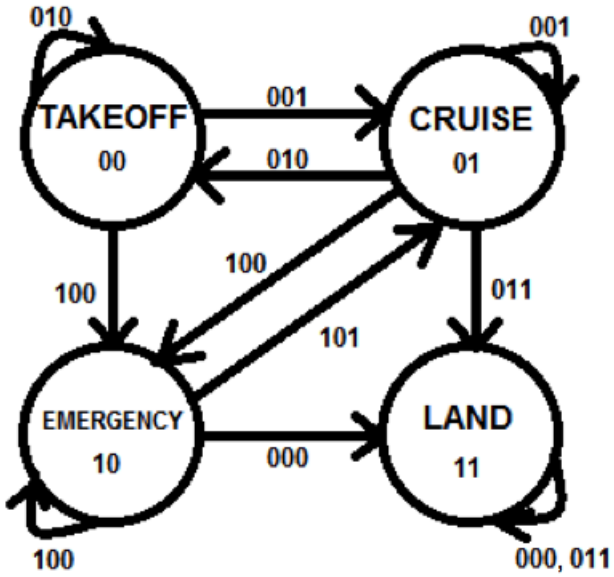
CURRENT STATE	INPUT	NEXT STATE	OUTPUT
TAKEOFF			SEATBELTS + VIDEO
		EMERGENCY	
	COLLISION COURSE		
			DRINKS SERVED
CRUISE		CRUISE	

Problem 3 Solution

1. 000 Collision Course
   001 Optimum Altitude
   010 Not at Optimum Altitude
   011 Destination in Sight
   100 Obstruction
   101 Obstruction Cleared
   110 [n/a]
   111 [n/a]

2. 00 Seatbelts + Safety Video
   01 Drinks Served
   10 Oxygen Masks
   11 Seatbelts + Dimmed Lights

Figure 1.19: Solution for Problem 3.3



3.

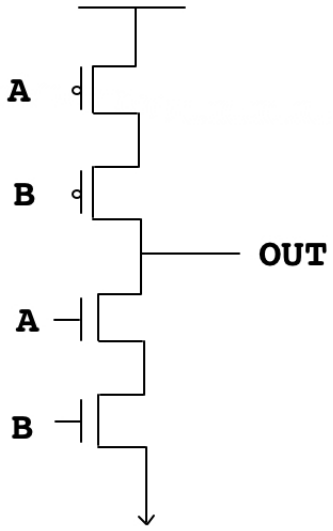
CURRENT STATE	INPUT	NEXT STATE	OUTPUT
TAKEOFF	[001]	CRUISE	SEATBELTS + VIDEO
TAKEOFF	[100]	EMERGENCY	[00]
EMERGENCY	[COLLISION COURSE]	LANDING	[10]
CRUISE	[100]	EMERGENCY	DRINKS SERVED
CRUISE	[001]	CRUISE	[01]

Problem 4
Consider the traditional 2^2 - by - 3-bit Memory module previously discussed. Please refer to Figure 3.21 of the Introduction to Computing Systems textbook. What is an unintentional consequence of replacing the 4 gates, which AND the WE bit with the output of the address decoder together, with a direct signal from the WE bit? (Explain in 15 words or less)

Problem 4 Solution
In the case where WE is 1, all other memory locations are also written over.
Explanation: This is because although the signal would be acceptable for the desired address location, it would have effects on the other locations where we require a 0 to prevent reading or writing to occur. An example of this can be seen when we have an A[1:0] of 11 and a WE of 1. The bottom location will have correct results because the WE is a 1 and the ANDed result of the WE and the output of the decoder is also a 1. The other three locations though will also receive 1s, which means those registers will also output their values. This creates falsified results. The decoder output is necessary for the memory to determine which location to write or read from.

Problem 5
Determine whether the circuit in Figure 1.20 is correctly designed- that is, does it work for all possible input combinations of A and B?

Figure 1.20: The circuit for Problem 5



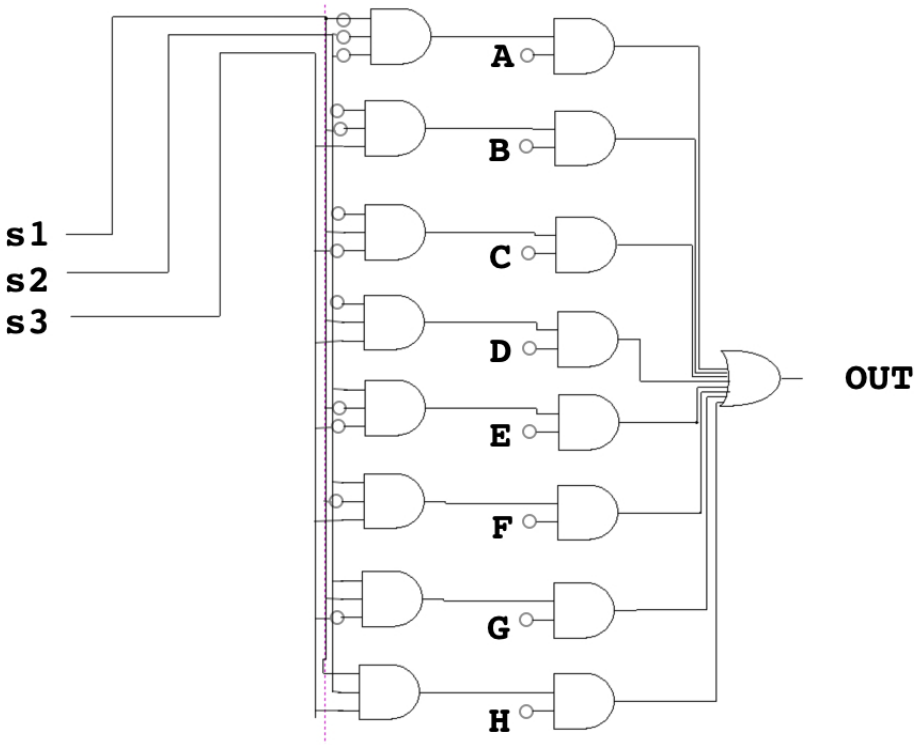
Problem 5 Solution
The idea here is that when testing the validity of a transistor circuit, you are really looking to avoid two things: short circuits and floating outputs. A short circuit exists when there is a direct path from power to ground. A floating output is when the output is connected to neither the ground nor the power. We say that we want NO path to ground when our output is high, but a path to ground when output is low. Because voltage is defined as a potential difference between two points, it is imperative that when our output is high, ground is 0 V so that there is indeed a potential difference. If ground is +5V as well as output, then there is no potential difference, thus output isn't truly high. If output is low, it needs to be at 0V, and ground needs to be at 5V so that there is a potential difference.

You see when A = 0 and B = 0, OUT = 1, and there is no path to ground, as the n-type transistors out the bottom are low. When A = 1 and B = 1, OUT = 0, and there is a path to ground. However, we run into a problem when A = 0 and B = 1, or A = 1 and B = 0. In the first case, we see that on top, output = 0, so we would expect there to be a path to ground, but there is not! In this case the output floats. In the second case, the same is true therefore output floats. We are able to conclude that this circuit is not properly built and does not work for all input combinations.

Problem 6
Eight students were asked a true or false question. It read: "True or False: The capital of Texas is Houston". Clearly, the answer is false.

They were instructed to write their name on a sheet of paper and their answer- either a 1 or a 0, to represent T/F respectively. We will use A-H to represent each student's answer. Your job is to build a mux such that you can effectively select a name and then look to the output to see if the student answered the question correctly. An output of 1 means that the student answered the question correctly, while an output of 0 means an incorrect answer.

Problem 6 Solution
There are three select lines for the eight inputs (n select lines for 2n̂ inputs). The select lines are fed into an AND gate that is a 1 precisely when that corresponding input is selected (What logical function that does represent?). The input coming from each letter is the student's answer. The students answer is fed into an AND gate but is inverted so that, when it is selected, the second AND evaluates to a 1 (if the student answered the question correctly). Each AND gate is then fed into an OR gate, as is the case with any mux. With this design, the output is a 1 precisely when the selected student answers the question correctly.



Chapter 2

The Data Path

2.1 Vocabulary

Active High The signal is asserted with a logical '1'. When we simply say a signal is "asserted" without providing its assertion value, then we usually mean an Active High assertion.

Active Low The signal is asserted with a logical '0'.

Address Control Logic Controls the input and output device registers. There are three inputs: MIO.EN, R.W, and MAR.

1. MIO.EN indicates whether a data movement from/to memory or I/O will occur.
2. R.W indicates load or store to memory location or I/O device
3. MAR: reference "Memory Address" Register below

Assertion The signal is "active", though this does not necessarily mean the signal is a '1' or a '0'; depending on the control logic, it can be either. An example of this is the R.W signal which is '0' when the program specifies a read from memory and '1' when a write is requested. In either case, we can say that the R.W signal is "asserted" since we care about its value.

Bus A bundle of wires that is used to transfer data between units of the computer. What makes the bus unique from other 16-wide wires is that the bus can have different sources and destinations depending on the state of the machine. The bus in the LC-3 can only hold one 16-bit value in any single clock cycle, therefore only one tristate buffer gate may pass data to the bus in any given clock cycle.

Clock Cycle The unit of time that a computer remains in any state. The end of a clock cycle triggers the transition from one state to the next in the LC-3's Finite State Machine (FSM). At the point of this state transition, the clock signal will move from its low state to its high state, and this will enable new data to enter any register of the machine so long as the control signals enable each specific register to be loaded. The clock signal is important because it is a global signal that is singularly responsible for keeping the entire machine synchronized.

Control Unit The control unit is the driver for the LC-3's state machine. At the beginning of each clock cycle, the control unit provides all of the control signals to the entire machine. The unique combination of control signals is completely dependent on the state of the machine. For instance, when the computer is in state 18 (during the fetch phase), the control unit is providing the same set of signals as it will any other time the computer is in state 18.

Control Signals Control signals have a variety of purposes. They can select an input through a mux, control whether or not a register gets modified, determine what data gets gated onto the bus, or specify what operation the logic block is to perform. Control signals are provided by the control unit and drive all important events that occur in the LC-3.

Data Path The data path represents how all work gets done in the LC-3 at a low level of abstraction. Specifically, it lays out how signals and data move through the machine. In order to achieve a firm understanding of the data path, it is recommended that you trace all events in each state. It would be good to have the state machine and data path documents handy when you do this.

Finite State Machine A system that consists of a finite set of states, including the inputs, outputs, and transitions for each of these states. The LC-3 has 49 states, each having a set of operations that the machine executes during that state. There is no difference in what the computer does between two instances of the machine being in the same state, except for the case of a conditional branch.

Gate Controls when data is allowed onto the bus In the LC3, gates are represented by tri-state buffers that control the output from a module to the bus. When the gate is "asserted" the contents of the output module is loaded onto the bus. At any given time, only one gate can be asserted. If more than one gate is asserted the value on the bus would be invalid.

Input/Output (I/O) External hardware is controlled by memory locations mapped to each device (memory bound I/O). This allows the programmer to read or write information to external hardware by reading or writing to the memory address associated with that hardware. For example, keyboard data can be read by interrogating the memory location that is mapped to the keyboard inputs.

Instruction Cycle Step by step sequence that are processed in the distinct phases of each Instruction (for detailed information on the Instruction Cycle phases, reference page 104 of Patt's book).

"Interrogate" (in context of memory I/O)To read or write to or from the memory by putting relevant values in the MAR or MDR and telling the memory subsystem to appropriately act upon the data (depending on the asserted value of R.W).

Instruction Register Holds the actual instruction being executed. In the LC3 it is a 16 bit register that contains the 4 bit op-code and 12 bit input that specify the source registers, destination registers, and immediate values.

Interrupt An interrupt is an external event that stimulates the computer. This requires additional hardware signals to interrupt the processor and supply it with a interrupt vector that provides an entry point in memory that will handle the interrupt. When an interrupt occurs, the Interrupt Enable bit (IE) is asserted. Each interrupt is given priority, which corresponds to the necessity of the event being dealt with in a timely manner. This means that when an external event occurs its priority is compared to that of the current process, only if the interrupt has a higher priority will it be executed. Note that this processes could also be an earlier interrupt triggered service routine

Program Counter Is a register which holds the memory address of the next instruction.

Memory Address Register (MAR) This register is loaded with the address of the memory location in which the next memory access is going to touch. During reads, this register will specify the address of the data that is being accessed. During writes, it will contain the address of the memory location in which the data is to be written to.

Memory Data Register (MDR) This register stores a value to be stored in memory or that has been retrieved from memory. The LD.MDR signal controls when data can enter the MDR. During a ST instruction, the MIO.EN signal specifies MDR to be loaded from the bus. During a LD instruction the MIO.EN signal specifies the MDR to be loaded from memory. Note: when MDR is loaded from the bus, it only requires 1 cycle. However, when MDR loads from memory, it must wait until the READY bit is set. (Generally this takes 5 cycles)

Multiplexer (Mux) Selects one signal from many governed by the bits in the selector line(s). For example, a 2 bit mux can select 1 of 4 lines, in general, the number of alternatives that can be chosen is equal to $2^{\text{number of bits in the selector line(s)}}$.

Opcode A sequence of bits (four on the LC3) which specify which instruction to carry out. Instructions are stored in the non-volatile portion of computer memory (ROM) and are decoded by the microcode in the structure labeled CONTROL. Most of the work the Decode structure does is out of the scope of EE306.

Operand The twelve bits that follow the Opcode which determine what data to act upon (which register or immediate value to store, load, add, etc).

Processing Unit A component of a computer architecture which processes data controlled by control signals. A simple example of a processing unit in the von Neumann model is the ALU. In the LC3 the ALU recieves input signal from the control store that designate ADD, AND, and NOT operations.

Steering Bit Sometimes instructions are specified with different modes which cause the instruction to act differently according to the value of the steering bit. In the case of the LC-3, steering bits are used to specify what operating mode the operands will take in binary operations such as ADD and AND.

von Neumann Machine A computer architecture which consists of five components: memory, a processing unit, input, output, and a control unit. Programs and Instructions are both stored in memory. The design of the LC-3 is based on the von Neumann architecture. Note: The only way to turn on the computer after it has received a HALT instruction is via external control (such as a power button).

2.2 Introduction to the Data Path

2.3 The Data Path Explained

The Data Path diagram is a top-down view of all of the LC-3's components. Understanding the data path requires an understanding of exactly how each instruction is carried to execution. The best way to do this is to trace the movement of data at each state of the machine. You may be asked to determine a set of control signals that are asserted at any state of the machine. It is important that you keep in mind what control signals must be asserted in each state in order for the event

specified by the stat machine to occur. For instance, if you see that the PC is loaded in a certain state, it is evident that the LD.PC signal must be asserted. Before we delve into the specifics of the state machine, it is important that you understand one critical distinction: combinational and sequential logic.

Combinational logic consists of all gate paths that do not take the clock signal into consideration. In terms of the LC-3 datapath, combinational logic consists of everything that is not a register. At the beginning of each clock cycle, new values that have just been applied to the system registers (PC, MAR, and the general purpose registers) stimulate the combinational circuitry of the machine, regardless of whether or not the result of that logic is important to the execution of the instruction. For instance, in a state where the PC is being gated onto the bus and sent to the MAR, the ALU is still taking inputs and generating an output. These inputs may be garbage values that add nothing to the execution of the load instruction. It doesn't matter though since the value of the ALU in this instruction is thrown away when the gateALU signal restricts the output from being put on the bus. In a way, combinational logic is constantly producing outputs that may or may not be used to update the state of the machine.

Sequential logic in the LC-3 is primarily comprised of the system registers and a few constructs in the Control Block. Sequential logic is responsible for all state transitions in the machine and it is all tightly coupled with the clock signal. Unlike combinational logic which gets updated nearly instantaneously when inputs change, sequential logic is only updated at a clock cycle transition. The point is that the LC-3 has a very specific set of tasks to do each cycle, and this is driven by the combinational and sequential circuitry working in lockstep. At the beginning of each cycle, new values get latched into registers and the control block provides a new set of control signals based on the new state of the machine (sequential logic), then everything else in the machine evaluates these new inputs and dumps the result onto a wire that goes into a register. However, that output must wait for the end of the current clock cycle before it can update the register and start the process over again.

2.4 Highlights of the Data Path

Each latched element only latches in new data at the end of the clock cycle. This has some very profound effects which, if not understood, can lead to much confusion. For example, during the Pre-Instruction phase (basically Fetch and Decode; the first three stages on the LC3's Finite State Machine diagram), the PC is gated onto the bus but the LD.PC signal is asserted and the PCMUX is set to allow PC <- PC+1 (see Figure 2.1).

Which value of the PC appears on the bus? The answer has to do with the nature of a latch itself. Latches only store a new value at the end of a clock cycle. Therefore, the PC pushes its current value on the bus and adds one to value on the PC input wire at the same time, but the incremented value of the PC is only stored at the positive edge of the clock. All the other elements receive the original value of the PC until the next cycle.

Remember that during each instruction cycle, all of the data paths are always active even though our drawings may not explicitly illustrate every one of them. The control signals govern the propagation of these signals, preventing them from interfering with the data that we care about. In short, the arrows with black heads on the data path illustrate paths that are always active; the small arrows with white heads illustrate control signals that change the flow of data through the multiplexers (these are also always active, but their signals are usually blocked off by a gate to prevent them from interfering with the data we care about); and the large equilateral white triangles illustrate gates (tri-state buffers) which block signals from passing through unless they have a

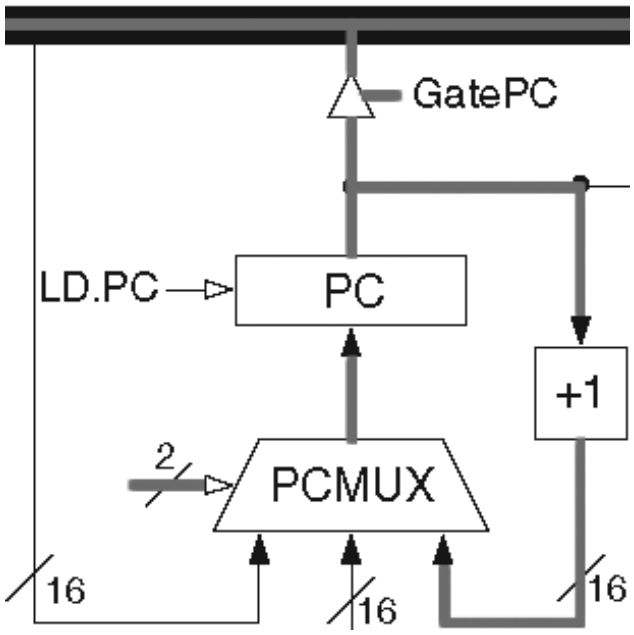


Figure 2.1: Example of a potentially confusing situation which illustrates the importance of the understanding of latches

signal to explicitly allow the transfer of signals. The values of these gates always matter, for they restrict access to the bus which can only handle one 16-bit value in each clock cycle. There will be situations where we do not care if data is allowed to flow through control structures (the small white triangles), but we always care about what is allowed on the bus by the gates (the large whie equilateral triangles).

2.5 LC-3 Instructions

2.5.1 The Pre-Instruction: Fetch and Decode

State 18 Signals Asserted:

- GatePC
- LD.MAR
- LD.PC
- PCMUX (Select PC+1)

Explanation: The value of PC does several things at once: The current value of the PC is gated onto the Bus while the LD.PC signal is asserted and the PCMUX is set to load PC+1 into the PC. All this happens at once, but the value of PC is not changed until the positive edge of the clock cycle, meaning that the Bus is given PC during the entire clock cycle but at the end of the cycle, PC+1 is eventually latched into the PC. When the value of the PC is on the Bus, it becomes latched in to the MAR since the LD.MAR signal is asserted. This implies that we're going to load from memory the address that the PC contains.

State 33 Signals Asserted:

- MIO.EN
- INMUX (Selects the line from MEMORY)
- LD.MDR

Explanation: Now that the MAR has the value of the block of memory that we want to load, we do the relevant processes to load from the memory: The value from the MAR flows into the ADDRESS CONTROL LOGIC and the MEMORY blocks. The MIO.EN, MEM.EN, and R.W signals are asserted to enable memory access and read/write capabilities. Notice that the R.W line goes into both the ADDRESS CONTROL LOGIC and the MEMORY units. Memory can take multiple CPU cycles to retrieve the data, so we loop in this state until we detect the R signal asserted from the MEMORY unit (this is indicated on the LC3's FSM as a loop with R-bar, indicating that we want to loop as long as R is not asserted). As long as the MIO.EN and LD.MDR signals are still asserted during these loops, we can be assured that we'll get the value from the memory into the MDR eventually since this forces the data from the memory to flow into the MDR during the loops.

State 35 Signals Asserted:

- GateMDR
- LD.IR

Explanation: Now that we finally have the value of MEM[PC] into the MDR, we now have to push it into the IR to actually execute any instruction that is contained in the MDR. We activate the GatePC, which pushes the value of the MDR onto the Bus. Since LD.IR is also asserted, the value of the MDR (which is currently on the bus) is pushed into the IR and is finally latched in.

State 32 Explanation: This is the Decode instruction where the machine moves to instruction-specific states based on the 'J' bits in the control logic. For more information about the 'J' bits, refer to page 473 of Patt's textbook (Appendix C.4). The structure of the control logic is outside the scope of this course, and the signals that are asserted on the FSM do not need to be scrutinized due to this consideration (the BEN signal, for example, is listed on the State Diagram but does not appear on Data Path; don't worry about this). For the remainder of this document, we will refer to the instructions specified by the state branched off from this state (state 32). For example, if we encounter an ADD instruction, then the next state will be state 1 branched off from state 32.

2.5.2 The ADD Instruction

The ADD instruction has two types of operation modes. The first operation mode takes two source registers and stores the value to a destination register. The second operation mode takes one source register and one immediate sign-extended value taken from IR[4:0]. The source registers provide the data to the ALU the output from the ALU is stored in the destination register. Both values are sent to the ALU and the result is stored in the destination register. To distinguish between the two modes, IR[5] is used as a steering bit: If IR[5] is 1 then the ALU uses one source register and a sign extended immediate value, otherwise the instruction uses two source registers.

State 1

Signals Asserted with IR[5]=1

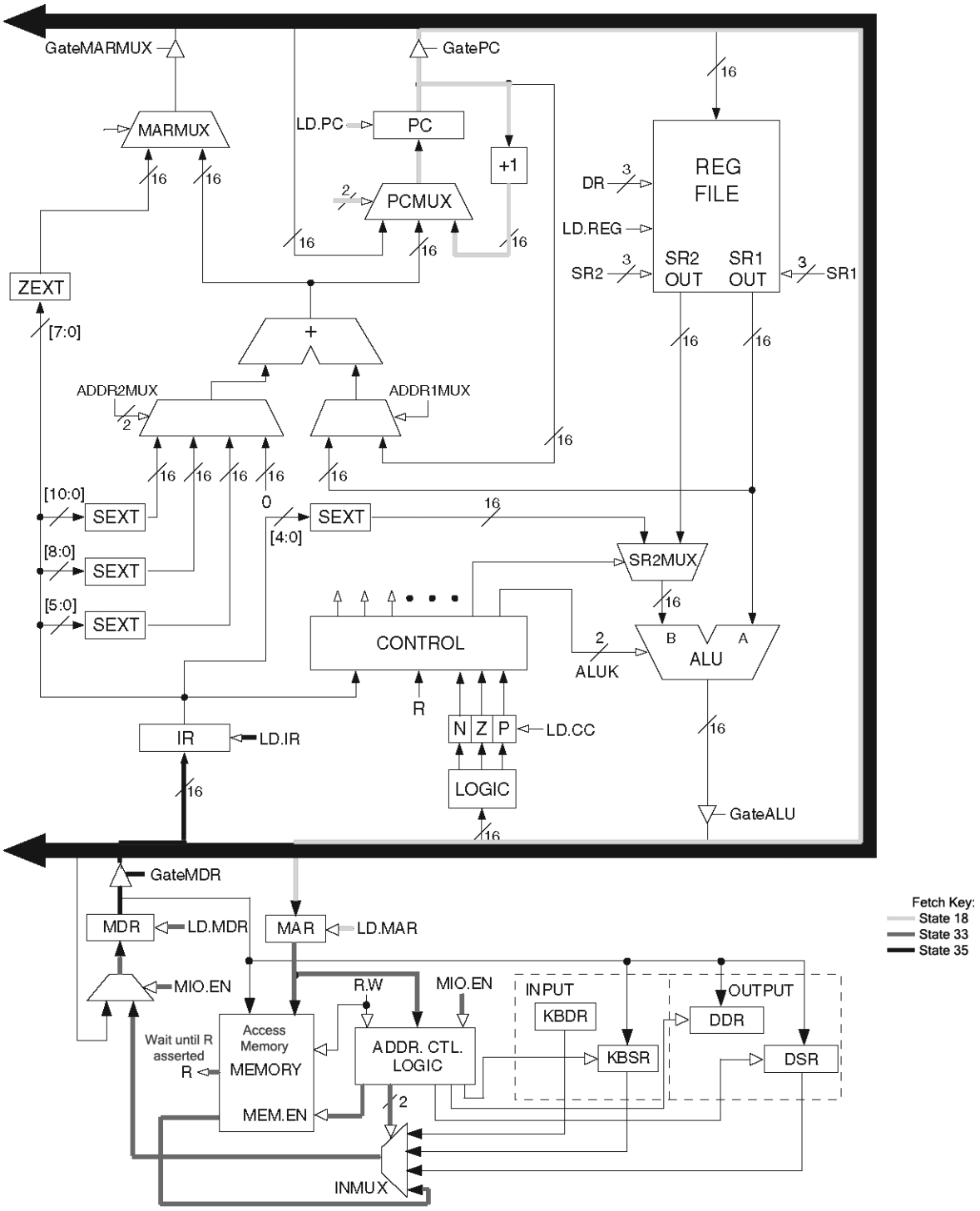


Figure 2.2: Data flow of the PreInstruction

- SR2Mux
- ALUK
- GateALU

Signals Asserted with IR[5]=0

- SR2Mux
- ALUK
- GateALU

Explanation: The SR1 value is selected and sent to ALU A and SR2 mux selects for either the immediate value or SR2 and sends it to the ALU B. ALUK is then set to 00 (00 is ADD, 01 is AND, 10 is NOT, 11 is pass value). Then, GateALU is asserted which allows the result from the ALU onto the system bus. Finally, the DR selects from the IR bits that specify the destination register and the value is taken from the bus and put into the register file.

2.5.3 The JMP Instruction

The JMP instruction allows the programmer to change the PC so that in the next cycle, the program will continue at the address specified. The JMP instruction can be implemented two different ways; it is possible that one path is more efficient than the other. The object of this instruction is to load the PC with the value in a specified base register (IR[8:6]). One possible way to achieve this is to drive the base register value out of SR1 onto the bus by setting the ALUK to pass the value in "A". Now that the bus contains the value of the base register, the PCMUX would accept the value from the bus and the LD.PC would be asserted to accept the new address. As an alternative, the base register would come out of SR1 and be selected by the ADDR1MUX. ADDR2MUX can be selected to choose one of two things: it can pass the value '0' or it could pass the sign extended bits of IR[5:0] which, if you will notice, will always be 0 for the encoding of this instruction. The adder would add the base register with 0, leaving its value unchanged, and would pass it to be selected by the PCMUX. Similar to the previous path, the LD.PC signal would assert and load the new PC value.

State 12:

Signals Asserted: The ALU Path

- LD.PC
- ALUK
- GateALU
- PCMUX
- SR1MUX[1] (Selects IR[8:6])

Signals Asserted: The ADDR1MUX Path

- SR2MUX
- SR1MUX[1] (Selects IR[8:6])
- ADDR1MUX
- ADDR2MUX
- LD.PC

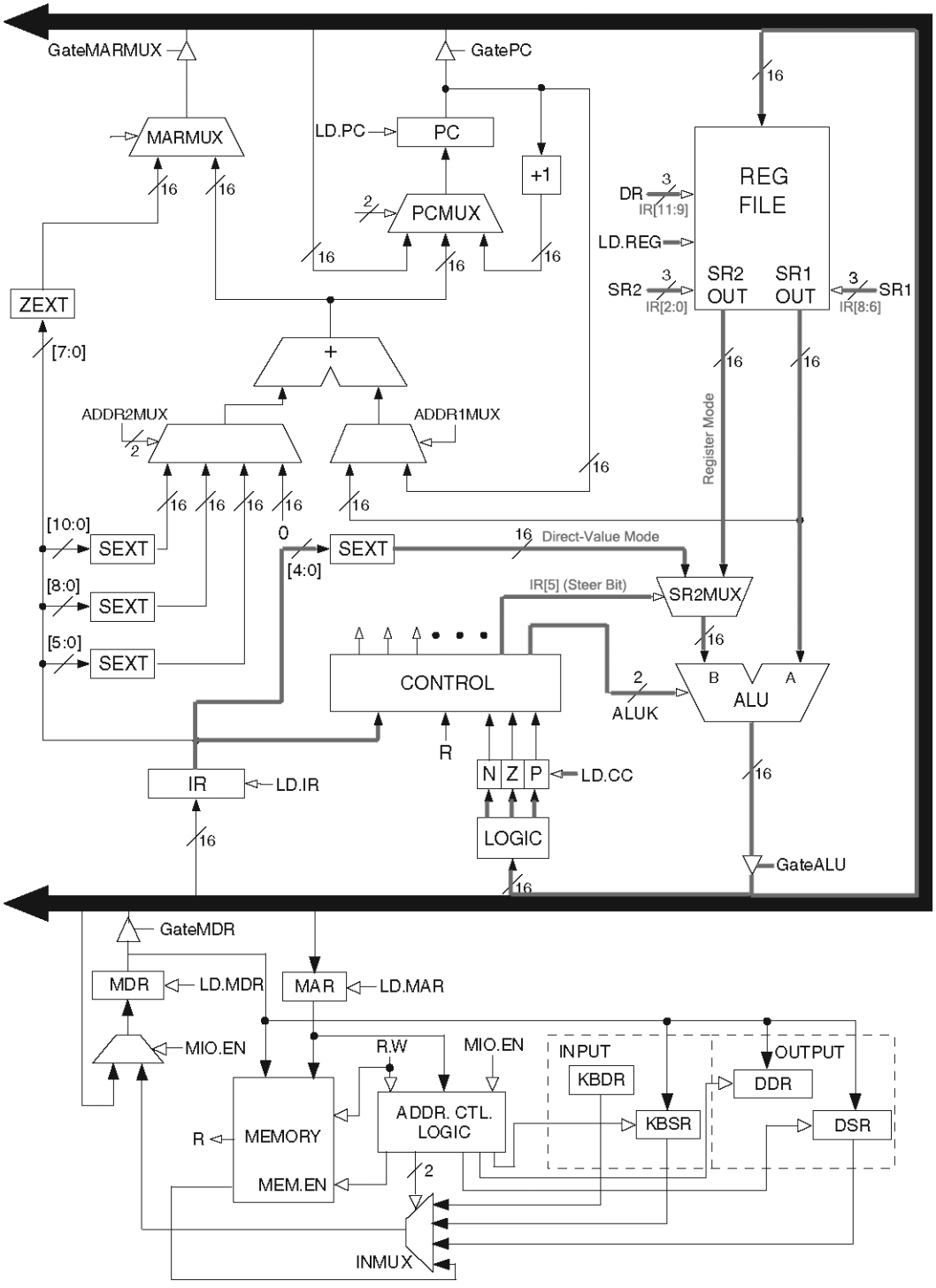


Figure 2.3: Data flow of the ADD instruction

- GateMARMUX
- PCMUX

Explanation: The SR1MUX[1] selects the register for the base address and the ADDR1MUX selects the SR1 line. The ADDR2MUX chooses the imemdiate 16 bit zero value or the SEXT (Sign Extended) IR[5:0] value. Finally, the GateMARMUX asserts and the value is loaded onto the bus. After the value is on the bus, the LD.PC asserts and the PC is loaded with the location that the JMP instruction wants to go to. In the ALU Path the ALU is set to PASS VALUE and the base register is propagated to the GateALU which asserts and puts the value onto the bus. From there just like through the ADDR1MUX path, the PCMUX selects for the bus line and the LD.PC allows the PC value to change to the JMP location.

2.5.4 The STI Instruction

The STI instruction is a store instruction that accesses memory twice. In fact, it is the only store instruction that actually performs a load from memory (in addition to a write to memory). The following will break down the STI instruction into three parts. First, an initial address will be calculated by adding an offset to the PC. This first step is like a regular LD; memory is interrogated and a value retrieved. However, the value retrieved is an address where the value in SR will be stored to. The next step is to return this value fetched out of the MDR into the MAR. Finally, the desired value is placed in the MDR, the memory access is initiated, and ithe value is stored at M[MAR].

State 11 Signals Asserted:

- MARMUX (Selects the output of the "+" structure)
- GateMARMUX
- ADDR1MUX (Selects PC)
- ADDR2MUX (Selects SEXT of IR[8:0])
- LD.MAR

Explanation: The value of the PC and the sign-extended value of IR[8:0] (PCoffset9) are first added and pushed on the bus where it is loaded into the MAR. This will prepare the machine to load the value contained within PC + PCoffset9 from memory.

State 29 Signals Asserted:

- R.W (Active Low[2])
- MIO.EN
- MEM.EN
- INMUX (Selects the line from MEMORY)
- LD.MDR

[1]This structure does not exist on the Data Path, but within the LC-3 it selects between providing IR[11:9] or IR[8:6] to the SR1 line. This is normally only used on instructions which provide a 9-bit offset to the instruction so that the machine can load DR, which only exists in IR[11:9], and send it to the ALU. This contrasts most instructions which usually have SR1, which is normally located in IR[8:6], load and flow into the ALU. Don't stress upon this structure, as it is one of the small internal details of the LC-3 which need not be overly scrutinized.

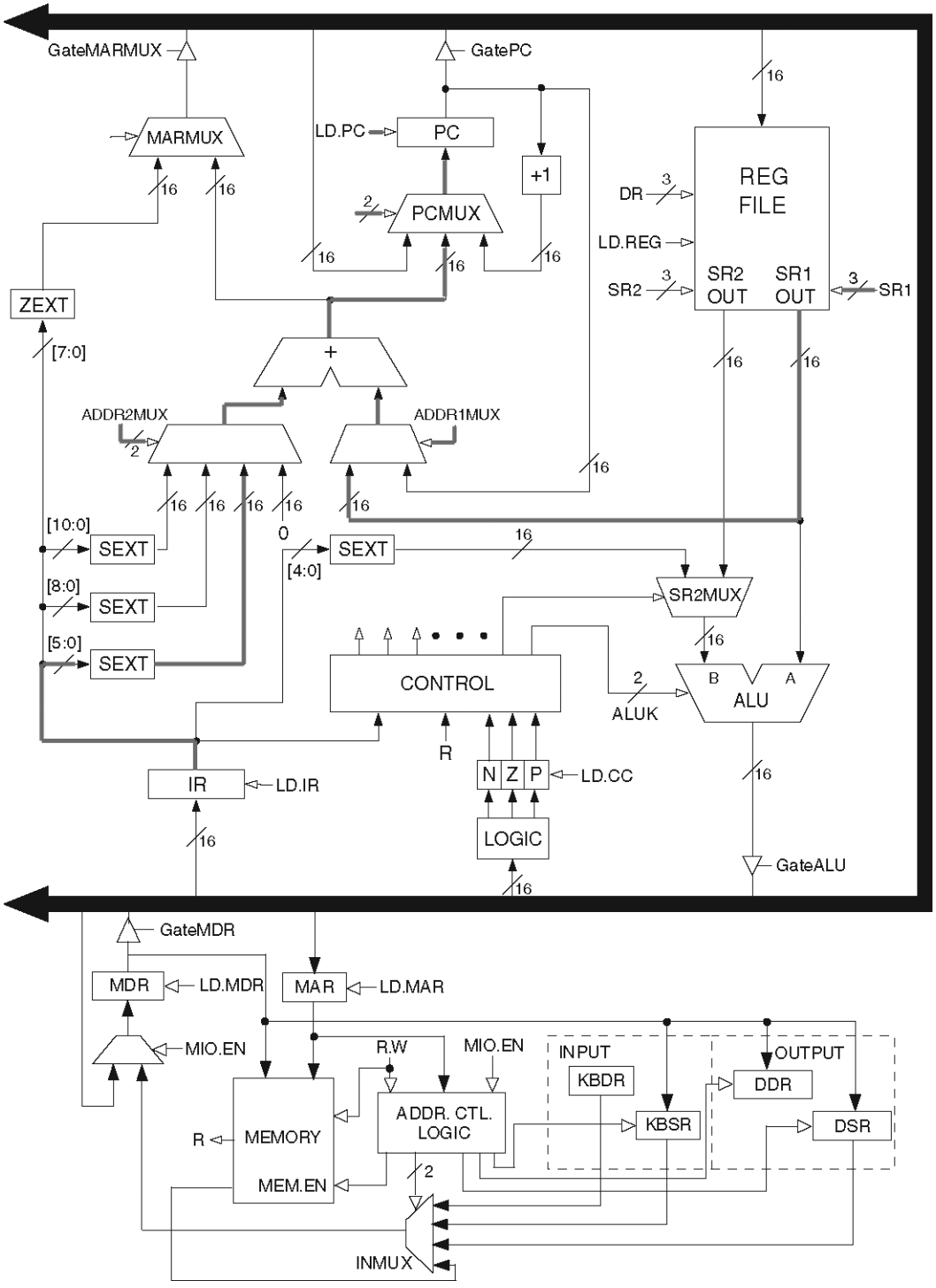


Figure 2.4: Data flow of the ADDR1MUX path of the JMP instruction

- Wait for R (asserted by memory when the memory read is finished)

Explanation: At this point we are loading from memory. The MAR, MIO.EN and MEM.EN control structures are asserted, and R.W is set to Active Low to specify a memory read[2]. The Address Control Logic block tells the INMUX to select the value coming from memory and push it into MDR by way of the MIOMUX[3], and the LD.MDR signal is asserted to allow the incoming value to overwrite what is currently in the MDR. Since memory access can take many cycles, the machine waits until the R signal is asserted to continue to the next state.

State 31 Signals Asserted:

- GateMDR
- LD.MAR

Explanation: This is a very simple step: The machine moves the value within the MDR into the MAR to read from the address specified by the data which was just read from memory.

State 23 Signals Asserted:

- GateALU
- MIO.EN (Active Low; Selects the value on the bus)
- LD.MDR
- SR1MUX (Selects IR[11:9])

Explanation: The machine moves the value within SR onto the bus by asserting a "Pass-thru" signal in the 2-bit wire labeled "ALUK". This allows the value within SR to be gated directly on the bus. MIO.EN is asserted to Active Low and LD.MDR is asserted to allow the value on the bus to be pushed on the MDR. The machine is now ready to store the value within SR (now within MDR) into the address specified by the MAR (PC + PCoff9).

State 16 Signals Asserted:

- MIO.EN
- MEM.EN
- R.W (Active High[2])
- Wait for R

Explanation: At this point the contents of the MDR (SR) go into the memory address specified by the MAR (PC+PCoff9). The logic knows to do this because R.W is set to an Active High signal, specifying a write to memory. The machine waits until R is asserted to indicate that the write was successful and the memory subsystem is ready for another operation.

[2]R.W Active High specifies a Write while R.W Active Low specifies a Read

[3]MIOMUX doesn't actually have a label, but it is just below the MDR and has a MIO.EN signal to pass either the value on the bus or the output from INMUX into the MDR.

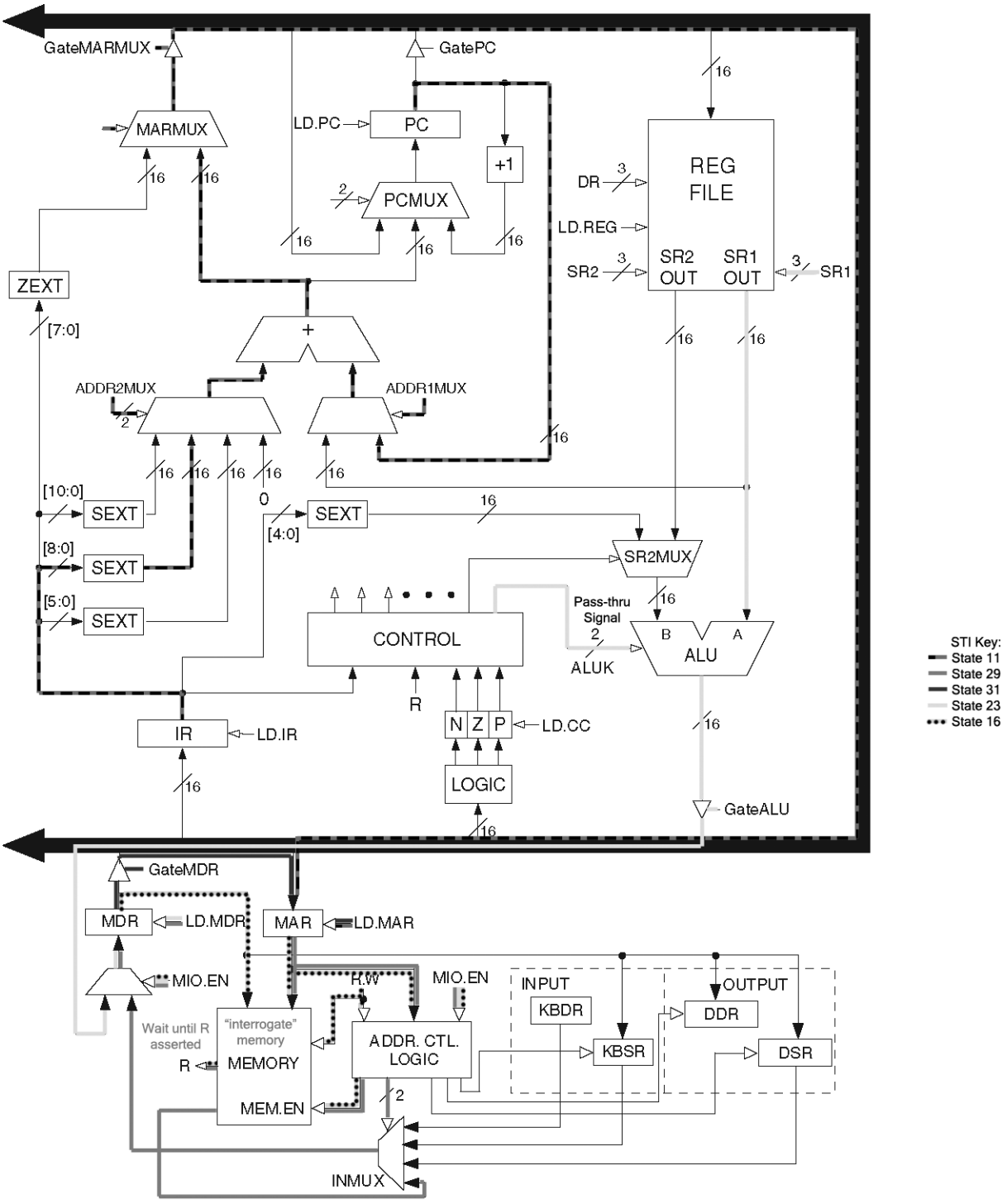


Figure 2.5: Data flow of the ADDR1MUX path of the JMP instruction

2.5.5 The TRAP Instruction

The TRAP Instruction performs service routines for specialized memory operations. The TRAP Vector table in your book will specify each value and its function. In general, the TRAP instruction controls input and output from peripheral devices (namely the keyboard and console for the LC-3) and also services the "HALT" instruction.

State 1 Signals Asserted:

- MARMUX (Select the ZEXT of IR[7:0])
- GateMARMUX
- LD.MAR

Explanation: First, the value from the IR is zero extended (ZEXT), passed through MARMUX and allowed on the bus by the GateMARMUX. Then, the LD.MAR signal is asserted allowing MAR to load the contents off the bus. The contents specify the TRAP Vector that corresponds to a specific system call subroutine.

State 29 Signals Asserted:

- R.W (Active Low[2])
- M.EN
- INMUX (Selects the line from MEMORY)
- LD.MDR = 1
- Wait for R

Explanation: Many operations happen at the same time in this state: The contents of the MAR specifies the address corresponding to the particular TRAP Vector. R.W, MIO.EN are asserted and the Address Control Logic unit produces a 2 bit value for the INMUX to select the output line from MEMORY. Memory[MAR] is read and waits until the R signal is asserted. After the memory has been read and R is asserted then the data at the memory location is loaded to MDR. At the same time the GatePC is asserted, pushing the PC value to the bus. The DR signal loads the PC value to R7 which will specify the return address after the system call is serviced.

State 30 Signals Asserted:

- GateMDR
- PCMUX (Selects the line from the bus)
- LD.PC

Explanation: To finally execute the service call, the GateMDR is asserted, allowing the data from MDR, which specifies the address of the service call, to be loaded onto the system bus. The PCMUX selects the bus line and the LD.PC is asserted so that PC contains the address of the service call from the bus. In the next state, the program will run the service routine specified by the address in the TRAP Vector table, which is now in the PC.

Chapter 3

Programming Primer

3.1 Introduction

A significant part of EE306 and Computer Engineering is programming - being able to dissect a problem statement and then teach a computer how to solve it. In the following section, we go through several LC3 example programs to illustrate the proper mindset while programming, both in general and in assembly in particular. We also guide you through several common types of exam problems that relate to programming - the "What Does This Program Do" and the "Fill In The Blanks" problems.

The important thing to note here is that these specific problems and solutions don't matter much - the thought process necessary to obtain the answer is key. Furthermore, it's impossible to learn how to program just by reading. If you are struggling, or even if you think you aren't, the only way to get better is to practice.

3.2 Program 1

Learning how to program is best done with examples. Thus, let's begin by describing a general outline that all programs follow in the context of an example. Take a simple program that takes two numerical inputs from memory, adds them, multiplies the sum by 3, and stores the result somewhere back into memory.

This specification can be described by a simple algorithm: Output = (Input1 + Input2) * 3. **All** programs have an underlying algorithm, though not all can be described as mathematically as this one. For example, the algorithm for the Character Count program described in the book is: Count the number of instances of INPUT1 (a character) in INPUT2 (an array of characters). Output = Instances of INPUT1 in INPUT2. Though this step may seem trivial, as we are simply writing down the problem's words in a more mathematical notation, it's an important step to resolve any ambiguities and to clearly see what you are asked to do.

In addition, all programs contain the following parts:

1. Initialization (Get ready to do the important stuff)
2. Processing (Do the important stuff, the equation)
3. Finish (Clean up loose ends)

Initialization

The initialization phase has two key tasks that it must complete:

1. Receive any user inputs
2. Place any necessary data (either user inputs or predetermined values) in either registers or in memory, whatever is easiest for the processing phase to retrieve.

Examples of things to complete during initialization include creating counters for loops, loading addresses of where values are stored in memory, and zeroing out the register in which the output will be stored. The initialization phase can be broken down further. Let's examine this process in the context of our sample program.

First, define the data that you need in this program, including intermediate values. For our program, you will need the following values:

1. INPUT1
2. INPUT2
3. Sum
4. Output

Knowing what values you need is not enough. The next step is to initialize registers for these values. We do these by loading registers with the appropriate values.

1. Load INPUT1 into Register 0
2. Load INPUT2 into Register 1
3. Load 0 into Register 2 as the initial value for the sum
4. Load 0 into Register 3 as the initial value for the output

Luckily, these operations can each be carried out with a single LC3 instruction. Assume that INPUT1 and INPUT2 are stored in memory locations with the appropriate labels. Here is the LC3 code for this section:

```
.ORIG x4000

LD R0, INPUT1
LD R1, INPUT2
AND R2, R2, #0
AND R3, R2, #0
```

Here, we assumed that the inputs are stored in memory locations. In general, the initialization step may need to be decomposed further. For example, to load from the keyboard, you need to convert from ASCII to decimal representation. To load from other input devices, you may need to do something else. That concludes the initialization phase.

Processing

The processing phase is the most varying part of programs. It takes what you set up in the initialization phase and acts on it to meet the specifications of the problem statement. The key is to follow the algorithm you set up.

```
Output = (Input1 +  Input2) * 3
```

First break up this algorithm into its logical parts, as you would if you were calculating the answer by hand. The first part can be described by:

```
Sum = (Input1 + Input2)
```

Now, replace these abstract values by their representations in the LC3:

```
Register 2 = Register 0 + Register 1.
```

Luckily, there is an opcode that does this exact thing.

```
ADD R2, R0, R1
```

Once you debug this addition, you no longer have to worry about Input1 and Input2. From now on, you can directly work with Sum. Though this step is trivial for this program, *modular programming* (programming in parts, with each part being debugged separately) will be important for any significant program. Now, take the next part of the algorithm :

```
Output = Sum * 3
```

Again, replace this algorithm by its representation in the LC3:

```
Register 3 = (Register 2) * 3
```

However, now we run into a problem. The LC3 does not have a multiply opcode. You must therefore take a step back and break down the algorithm further.

```
Output = Sum + Sum + Sum.
```

Unfortunately, no opcode exists to add 3 values at once. So break down the algorithm further, defining new intermediate values where needed.

```
Sum_intermediate = Sum + Sum
Output = Sum_Intermediate + Sum
```

Both these steps can be carried out with the available opcodes. Now, represent them using registers. It is OK to use Register 3 to temporarily hold SumIntermediate.

```
Register 3 = Register 2 + Register 2
Register 3 = Register 3 + Register 2
```

Translating to LC3 code is now trivial:

```
ADD R3,R2,R2
ADD R3,R3,R2
```

Register 3 now contains the output of the program, as we wanted all along. That concludes the processing phase.

Finish

Now to write the finish phase.

In this program, we store the output in the memory location labeled 'OUTPUT.' For this, we allocate 1 memory location for the output below our code, and label it 'OUTPUT.' Any program that wants to read our solution and do something with it can read that memory location.

Storing a value in a memory location can be done in one line:

```
ST R3, OUTPUT
```

We are now done with this program. Here is the complete program:

```
.ORIG x4000
LD R0, INPUT1
LD R1, INPUT2
AND R2, R2, #0
AND R3, R2, #0
ADD R2, R0, R1
ADD R3,R2,R2
ADD R3,R3,R2
ST R3, OUTPUT
HALT

INPUT1 .FILL #5 ;the input for this program is trivial. In reality, input will rarely
    be hardcoded
INPUT2 .FILL #20
OUTPUT .BLKW 1

.END
```

With those inputs, the output is 'K.' Looking up 'K' in the ASCII table, we see that it corresponds to the decimal value 75. Test the program with some other values to verify its correctness. Congrats, you have finished this (admittedly easy) program. The key is to remember that you can and should follow this same method for every program you write.

Now, let's go through the same process with a harder program.

3.3 Program 2

Given a word stored in memory location x3000 swap each nibble of the high byte and the low byte and store the swapped number into memory at x3001. For example if x3000 contained x3CF5, x3001 should contain xC35F.

In this section, we will give you a higher level description of each step in the algorithm. You should practice breaking down these descriptions into manageable bites that you can directly translate into code.

We will use the following algorithm to solve this problem. If you can think of a different one, great! Write the program, then compare outputs with our version.

1. Load your number from memory into two registers (initialization)
2. Isolate the low nibbles of high and low bytes of your number in one register using a mask
3. Shift left four times by adding number to itself four times keep in register
4. Isolate high nibbles of high and low bytes of your number in the other register using another mask
5. Shift right four times by dividing number by two four times by using the counting method
6. Add the two numbers together and store back into memory

If you didn't understand the above algorithm, don't worry. Let's look at each part individually, as we did for the previous program.

Initialization

Our first step, initialization, consists of receiving any user inputs and placing any necessary data (either user inputs or predetermined values) in either registers or in memory, whatever is easiest for the processing phase to retrieve.

Seeing that we have no need for user input our initialization will consist of placing our necessary data into registers and memory. But what is the necessary data? To find that out let's take a look at our problem and our algorithm.

We will need to reserve two registers to hold our final shifted numbers. And we will need a counter of four to keep track of how many times we have shifted we will also need the masks (x0F0F and xF0F0) to isolate the low and high nibbles of each byte and a register in which to store them. Also, since our method of shifting left destroys our original value we will need a register to hold our number each time we shift it left.

Here is the initialization:

```
.ORIG x4000
LDI R0,LOCATION ;initialize two registers to hold shifted values
LDI R1,LOCATION
LD R3, COUNT ;R3 has counter for shifting right
AND R4,R4,#0 ;clear register to hold new shifted right value
```

And at the bottom of our code we have our masks, counter, and locations we want to get and store our data into:

```
LOCATION .FILL x3000
FINISHED .FILL x3001
COUNT .FILL x0004
MASK1 .FILL x0F0F
MASK2 .FILL xF0F0
```

Processing

Our second step processing comprises the bulk of our algorithm:

1. Load your mask
2. Isolate the low nibbles of high and low bytes of your number in one register by ANDing your number and the mask
3. Shift left four times by adding the number to itself four times keep in register

So we know that shifting left is the same as multiplying by 2 and when we multiply a number by two we are essentially adding the number to itself so shift left four times becomes:

1. Add number to itself
2. Add number to itself
3. Add number to itself
4. Add number to itself

In LC3 Assembly, the first part of the processing is represented as:

```
LD R2,MASK1
AND R0,R0,R2 ;mask and shift left four times
ADD R0,R0,R0
ADD R0,R0,R0
ADD R0,R0,R0
ADD R0,R0,R0 ;R0 has shifted left four times number
```

Following our pattern of breaking down each step into smaller pieces, let's break down the second part of the processing phase. As a reminder, here is what we want to do in this part:

1. Load your other mask
2. Isolate high nibbles of high and low bytes of your number in the other register by anding your number and the mask
3. Shift right four times by dividing number by two four times using the counting method

At this point, you should rightfully be asking yourself what the counting method for right shifting is. Let's try to understand that now.

So if shifting left is multiplying by 2 then shifting right can be represented as dividing by 2. This gets a little trickier, so let's think what happens when we divide a number by two. One can think of dividing a number by two as counting how many 2's there are in the number or how many times you can subtract two from the number before you get to 0 or negative. So for instance 8/2 can be computed by:

```
8-2=6 (count=1) 6-2=4 (count=2) 4-2=2 (count=3) 2-2=0 (count=4)
Thus 8/2=4. Or in binary 1000 (8) shifted right is 0100 (4).
```

What does this look like in Assembly? At this point, try writing the code yourself before looking at the answer. Once you're done, come back.

```
LD R2, MASK2
AND R1,R1,R2 ;mask
BRZ DONE2 ;if the masked number to start with is x0000 then you are done
SUB ADD R1,R1,#-2 ;shifting right
BRZ DONE ;if number is x0000 the you have shifted right once. You have subtracted 2 X
    number of times. X is thus your initial value divided by 2 and is stored in R4 (
    actually, X-1 is stored in R4. Why?)
ADD R4,R4,#1 ;if your number is not 0 then you haven't finished the shifting process
BRNZP SUB ;continue subtracting
DONE ADD R4,R4,#1 ;you branched before you counted the last time you subtracted so you
     need to add one to R4. Now, R4 is equal to X.
ADD R1,R4,R1 ;R1 has the shifted value now
AND R4,R4,#0 ;clear R4 to hold next shifted value
ADD R3,R3,#-1 ;decrement counter because we need to shift right four times
BRP SUB ;branch if your counter is not yet at zero
DONE2 ADD R0,R1,R ;if you are done shifting right add R0 (nibbles shifted left) and R1
;(nibbles shifted right) R0 has result
```

Finish

Now that you have your swapped number in R0 all that there is left to do is store that number back into memory at x3001!

```
1
2 STI R0,FINISHED
3 HALT
```

We are now ready to combine all of our pieces:

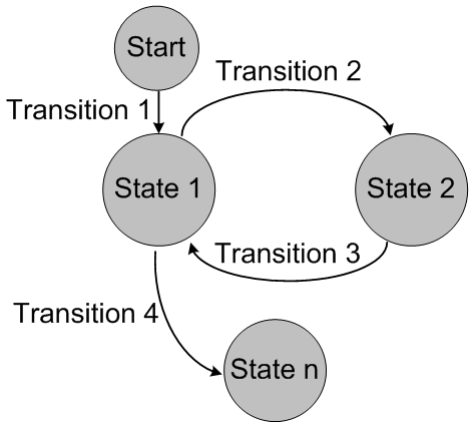
```
1
2 .ORIG x4000
3 LDI R0,LOCATION ;initialize two registers to hold shifted values
4 LDI R1,LOCATION
5 LD R3, COUNT ;R3 has counter for shifting right
6 AND R4,R4,#0 ;clear register to hold new shifted right value
7 LD R2,MASK1
8 AND R0,R0,R2 ;mask and shift left four times
9 ADD R0,R0,R0
10 ADD R0,R0,R0
11 ADD R0,R0,R0
12 ADD R0,R0,R0 ;R0 has shifted left four times number
13 LD R2, MASK2
14 AND R1,R1,R2 ;mask
15 BRZ DONE2 ;if the masked number to start with is x0000 then you are done
16 SUB ADD R1,R1,#-2 ;shifting right
17 BRZ DONE ;if number is x0000 the you have shifted right once. You have subtracted 2 X
    number of times. X is thus your initial value divided by 2 and is stored in R4 (
    actually, X-1 is stored in R4. Why?)
18 ADD R4,R4,#1 ;if your number is not 0 then you haven't finished the shifting process
19 BRNZP SUB ;continue subtracting
20 DONE ADD R4,R4,#1 ;you branched before you counted the last time you subtracted so you
     need to add one to R4. Now, R4 is equal to X.
21 ADD R1,R4,R1 ;R1 has the shifted value now
22 AND R4,R4,#0 ;clear R4 to hold next shifted value
23 ADD R3,R3,#-1 ;decrement counter because we need to shift right four times
24 BRP SUB ;branch if your counter is not yet at zero
25 DONE2 ADD R0,R1,R ;if you are done shifting right add R0 (nibbles shifted left) and R1
26 ;(nibbles shifted right) R0 has result
27 STI R0,FINISHED
28 HALT
29
30 LOCATION .FILL x3000
31 FINISHED .FILL x3001
32 COUNT .FILL x0004
33 MASK1 .FILL x0F0F
34 MASK2 .FILL xF0F0
35 .END
```

3.4 Program 3: The State Machine

Another common type of program is the implementation of a finite state machine. You can find the details of and uses for a finite state machine in your textbook or online. Most basically, a FSM is defined by a group of *states* and textittransitions between those states.

You should have already been introduced to a FSM as a hardware design tool, as in the state machine for the LC3. A state machine is also a good algorithm design tool that can be implemented

Figure 3.1: A basic State Machine



in software to design a real world system with states and transitions between those states.

An FSM software controller is an elegant solution with no branch statements and comparisons. We do not expect you to come up with it yourself, so we provide the solution below. Your job is to understand it and be able to implement it for any given system.

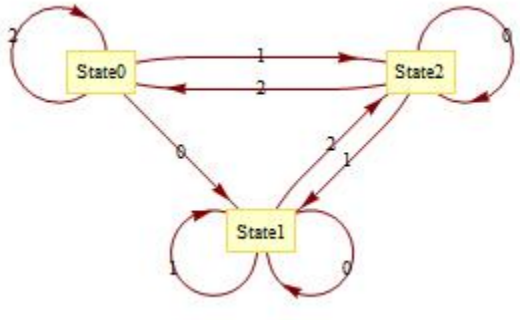
The thought process here is that it's better to write and understand this controller solution once than to rewrite a program each time your state machine design changes slightly. Furthermore, staying away from branch statements is great for several reasons. First, it increases the runtime performance of the program. In a real world system, like a traffic light system, performance is everything. Furthermore, the fewer branch statements that a program has, the easier it is to verify that it is correct.

A program that implements a FSM is easy to understand and extend and always follows the same pattern. In the following example, we will implement a Moore Machine, in which the output is a function of only the current state. (A Mealy machine's output is a function of the current state and current inputs, and is not used in this course. You will learn more about these machines in future courses).

A state machine program has two main parts: a data structure that defines the state machine and a controller. Each iteration of the controller loop represents the handling of a single state. The controller outputs the current state's output, inputs data, and transitions to the next state. We'll go through the state machine code using the following state machine diagram: Here is some LC3 code that implements the state machine in Figure **??**. Remember, the solution below is not the only way to design the system, but it is one of the best.

```
1
2  .ORIG X4000
3            ;initialization code - preparation for statemachine loop
4    LEA R2,STATE0   ;loads address of initial state into R2.
5  FSMController
6    LDR R0, R2, #0    ; loads into R0 the output of the current state
7    LD R3, digitasciiadd
8    ADD R0,R0, R3   ; integer to ascii conversion
9    OUT       ;output what is in R0
10   IN       ;load input
11   LD R3, digitasciineg
12   ADD R0,R0, R3   ; ascii to integer conversion
13   ADD R2,R2,#1    ;add input to R2 (also add offset to next state array)
```

Figure 3.2: Another basic state machine




```
  ADD R2,R2,R0     ;R2 is now a pointer to the address of the next state
  LDR R2,R2, #0    ;R2 now points to next state

        ;loop to FSMcontroller
  BRNZP FSMController
  HALT


digitasciineg .FILL X-30  ; negative offset for ASCII value for 0
digitasciiadd .FILL X30 ;positive offset for ASCII value for 0

;;;;;;;;;;;;;;;;;;; State Data Structure
STATE0  .FILL #0  ;output for state0
    .FILL STATE1 ;next state for input = 0
    .FILL STATE2 ;next state for input = 1
    .FILL STATE0 ;next state for input = 2
STATE1  .FILL #1  ;output for state1
    .FILL STATE1 ;next state for input = 0
    .FILL STATE1 ;next state for input = 1
    .FILL STATE2 ;next state for input = 2
STATE2  .FILL #2  ;output for state2
    .FILL STATE2 ;next state for input = 0
    .FILL STATE1 ;next state for input = 1
    .FILL STATE0 ;next state for input = 2
.END
```

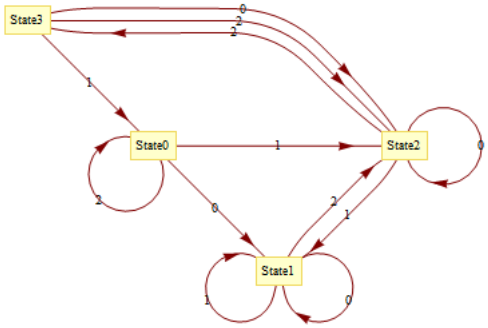
Let's start breaking this program from the end. The data structure defines the FSM. This FSM has 3 states.

Each state, in this example, has 2 parts: an output value and an array of pointers to the next state. For a Mealy machine implementation, simply change the output value into an array, with 1 output value corresponding to each input value.

To add another state, just add another State structure to the end. For example, consider the challenge of changing our structure to match the following diagram: Change the State2 structure to:

```
STATE2  .FILL #2  ;output for state2
    .FILL STATE2 ;next state for input = 0
    .FILL STATE1 ;next state for input = 1
    .FILL STATE3 ;next state for input = 2
```

Figure 3.3: The state machine in **??** modified slightly



And add State3:

```
STATE3  .FILL #5  ;output for state3
    .FILL STATE2 ;next state for input = 0
    .FILL STATE0 ;next state for input = 1
    .FILL STATE2 ;next state for input = 2
```

That is the magic of a FSM implemetation: To represent a different FSM, there is no need to rewrite the controller code. One just has to change the State data structure.

Next, let's break down the code that runs the machine.

```
ORIG X4000  ;initialization code - preparation for statemachine loop
LEA R2, STATE0  ;loads address of initial state into R2. R2 will keep track of the
    current state.
```

We'll need to keep track of the current state each iteration of the controller code. Let's use a register to hold the address of that state. Throughout the program, R2 holds the address of the current state. The above code initializes State0 as the first state.

Let's examine the first part of the FSM controller:

```
FSMController
  LDR R0, R2, #0    ; loads into R0 the output of the current state
  LD R3, digitasciiadd
  ADD R0,R0, R3    ; integer to ascii conversion
  OUT          ;output what is in R0
```

The #0 is the offset for the output of the current state from the beginning of the current state. We need an offset because R2 holds the address of the current state. This state has multiple parts: an output and an array of addresses to the next state. The offsets are the difference between the beginning of the state and the beginning of each part. If the output in the structure was located

after the pointer array, the offset would be changed to #3. Usually, the offsets are defined using variables, but we directly include it here for simplicity.

The rest of the code outputs the output for the current state onto the screen and waits a specified amount of time. Let's finish with the 2nd half of the controller:

```
IN          ;load input
LD R3, digitasciineg
ADD R0,R0, R3    ; ascii to integer conversion
ADD R2,R2,#1     ;add input to R2 (also add offset to next state array)
ADD R2,R2,R0     ;R2 is now holds the address of the address of the next state
LDR R2,R2, #0    ;R2 now points to next state
        ;loop to FSMcontroller
BRNZP FSMController
```

We first load the input from the keyboard. This input will determine the next state as defined by the state diagram. We add the offset (#1) to state array from the beginning of the state. R2 now holds the address of the beginning of the next state array for the current state. We then add the input to this value. R2 now contains the address to the correct index of the state pointer array. The next opcode, LDR, loads the value at that address, which is itself the address of the next state.

After understanding the program, run it in the simulator. Observe that the first output is 0, the output of our initial state, State0. Now, type in numbers and see how the next output corresponds to the correct state in the state machine, defined by the old state and the input.

Note that there are various ways to extend this program, like ensuring that the input is valid (there is a transition defined for that state and input), and changing the state diagram. You should try these out to gain a better understanding of software state machines.

3.5 Fill in the Missing Instructions Type Problems

One type of programming question you're sure to encounter is the fill-in-the-blank question. These questions will give you a program in which certain lines or parts of lines are left blank for you to fill in based on the purpose of the program. Here's an example from an old exam:

In Problem set 5, you were asked to write the subroutine BIN_GET which inputs a binary number entered from the keyboard and stores the zero extended value into R0. The program below calls the subroutine MOD_BIN_GET which in addition to storing the zero extended value in R0, also stores the total number of bits entered into R1. For example, if the user types 10010, R0 would contain 0000000000010010, and R1 would contain the value 5. Assume the user types from 1 to 16 binary digits.

After calling the MOD_BIN_GET subroutine, the program below sign extends the value that the user entered from the keyboard and stores the result in R0. However, a few instructions have been left out. Your job: complete the program. Note: Each box corresponds to 1 missing instruction.

```
        .ORIG  x3000
        JSR    MOD_BIN_GET
               [                          ]
        ADD    R2, R2, #1 ; R2 = 0000000000000001
        NOT    R3, R2     ; R3 = 1111111111111110
LOOP    ADD    R1, R1, #-1
        BRz    DONE
        ADD    R2, R2, R2
               [                          ]
        BRnzp  LOOP
DONE    [                          ]
        BRz    SKIP
        ADD    R0, R0, R3
SKIP    HALT
        .END
```

The first step is to read the problem description and locate the purpose of the program. Here, we learn that we want to store a sign-extended keyboard input into R0. Our program calls MOD_BIN_GET which takes an input – a binary number – from the keyboard and does two things: 1) stores the zero-extended input into R0; 2) stores the number of bits in the input in R1. As the problem tells us, we may assume the user types 1 to 16 binary digits, relieving us from having to deal with too many bits or an incorrect input.

The next step is to look at the program and figure out what it's already doing without the lines we have to fill in. We see that R2 contains x0001 and R3 contains xFFFE. Since the third line of our program adds 1 to R2 with the result being R2 containing x01, we know that R2 must contain x00 before this line. Remember, you cannot assume the initial value of R2 when the program starts. It most likely contains junk content. Thus we know we must clear R2 in the first blank:

```
AND R2,R2,#0
```

We now get to the main LOOP of our program. We notice that we exit LOOP when R1 becomes 0, and we know R1 contains the number of bits in our binary input, so we'll run through LOOP as many times as there are bits in our input. We then left-shift R2 (Adding a number to itself is the same as left-shifting it. There are many such patterns/instructions that correspond to common programming necessities. Practice is the key of recognizing these patterns). Then we do something, and then go back through LOOP. What do we do after left-shifting R2? At this point, it's not clear what we do with R2, but we should move on and hope that it makes more sense later on.

After finishing with LOOP, we do something; if that something results in anything except a zero, our program is finished. However, if that something yields a zero, then we add R3 to R0 and store the result in R0. Remember, the goal of the program is to leave R0, originally containing the *zero-extended* input, with the *sign-extended* input. And remember that R3 originally contains 1111111111111110. Zero-extension means adding all zeros to the front end of a number, regardless of its sign (high bit). Sign-extension means adding 1s to the front end of a negative number (high bit is 1) and 0s to the front end of a positive number (high bit is 0. If we want to sign-extend a zero-extended positive number, we don't need to change anything; if we want to sign-extend a zero-extended negative number, we only need to change all the leading 0s into 1s. This can be accomplished through adding, which is exactly what this program does. That second-to-last line – ADD R0,R0,R3 – makes a zero-extended negative number into a sign-extended negative number. Thus it is clear that the blank with label DONE checks whether our number is positive or negative. This check yields a zero if our result is positive and something nonzero if it is negative.

If the input is indeed negative, we need to add the correct amount of 1s and make sure we don't change the number itself. We can do this by left-shifting R3 by the number of bits in the input. We do this in the second blank:

```
ADD R3,R3,R3
```

Now all we have left to do is the check. We need to see if our original input was negative or positive. To do this we use R2. Remember, R2 originally contained x01 and was left-shifted by the number of bits in our input. This means we have a 1 in R2 in the same place as the highest bit of our input and a 0 everywhere else – a bit mask. If we AND R2 with R0, we isolate the highest bit of our input, telling us whether it's positive (isolated bit is 0) or negative (isolated bit is 1). We do this in the final blank:

```
AND R2,R2,R0
```

We're now done with the problem!

Here's another fill-in-the-blank problem:

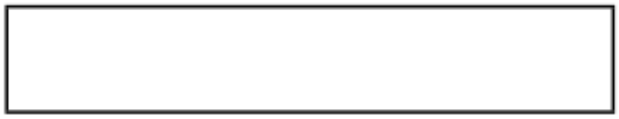
Problem 4. (10 points): One algorithm for dividing a positive (non-zero) **even** number by 2 is to load the even numb into one register, load a second register with 0, and then continually decrement the first and increment the second, un you have the same value in both registers. That value is your original even number divided by 2.

Example: Take the value 10: (10,0) → (9,1) → (8,2) → (7,3) → (6,4) → (5,5). Hooray!

The subroutine shown below, with the two missing instructions, performs this algorithm.

Your job: Insert the two missing instructions.

```
        LDI   R0, INPUT
        AND   R1, R1, #0
AGAIN   ADD   R0, R0, #-1
        ADD   R1, R1, #1
        NOT   R2, R0
```



```
        ADD   R2, R2, R1
```

```
        STI   R0, OUTPUT
        RET
INPUT   .FILL x3100
OUTPUT  .FILL x3101
```

The first thing we should do is understand the purpose of the program. We want to divide a number by 2 by finding the halfway point between the number and 0.

As setup steps, the program first loads R0 with the input and then clears R1. We then enter a loop. We can guess that this label corresponds to a loop because there are limited reasons for a label. Labels are used to mark the beginning of a loop (so that we can branch back to it), to access a memory location that stores some required value, and to mark functions. It does not make sense to have a variable stored in the middle of our code, and the code isn't separated from the main code, like a function normally is. Thus, the label marks the beginning of a loop. Since there are

no branch instructions given in the program, we may assume that one of the two blanks will be a branch back to AGAIN.

Our loop starts by decrementing the input (in R0) and incrementing R1. It then NOTs R0 into R2. We want to exit the loop when R0 and R1 are equal; a comparison can be done by subtracting one from the other and then branching based on that result. Since the LC-3 doesn't have a subtract instruction, the instructions that follow are a common way to implement a subtract. Start associating these steps with a subtraction/comparison. The NOT step starts us off with this action. If we want to subtract a number we actually have to add its negative. We can negate a number by NOTing it and adding 1. This is exactly what we do in the first blank:

```
ADD R2,R2,#1
```

The program shows that we then ADD R2 and R1. Since R2 now contains the negative of R0, we're really doing R0 – R1. Remember, every time we go through the loop we increment R1 and decrement R0, and we want to exit the loop when the two are equal: i.e., when R0 – R1 = 0. This means we want to stay in the loop if the result isn't 0, i.e. if it's negative or positive; thus we know the second blank as well:

```
BRnp AGAIN
```

And with that, we finish the problem! However, we have only brushed the surface of missing instruction problems. This type of problem can get very complex quickly. Now go and solve some of these problems from past exams. We can guarantee that you will have at least 1 of these problems on a EE306 test, or your money back (disclaimer: You will not receive your money back).

3.6 What Does This Program Do? Type Problems

Another common exam question is of the type Ẅhat does this program do?Äs a computer/software engineer, a large part of your job will be to decipher code, either your own (which is why comments are always great) or that of other people, so that you can work with that code. These types of problems test that skills. Our goal here is to show you the thought process that goes in behind solving one of these problems. Let's dive right in with an example.

Problem 1

```
        .ORIG  X4000
         LDI  R0,DATA
         ADD  R0,R0,R0
         STI  R0,RESULT
         HALT

DATA     .FILL  X4001
RESULT   .FILL  X4003
         .END
```

This program looks pretty straightforward and simple but be careful not to jump to conclusions as many times when asked one of these types of problems the correct answer is not the straightforward one!

For instance lazy student Dan might have looked at this and said this is easy it multiplies a value data by two. Well lazy Dan if you weren't so lazy maybe you'd realize that you are completely WRONG

So lets learn a lesson from our lazy friend Dan and begin by looking at the first instruction ldi R0,data simple enough right?

WRONG!

What is in data?

Oh its x4001... wait a second so I am indirectly loading a memory location that I am writing too thats odd!

Yes student it is odd so what are you going to do about it?

Well I am going to see what I am really loading into R0... Oh I am loading x1000 the opcode for the following instruction (ADD R0,R0,R0) into R0. Isn't that tricky!

Okay so big whoop you figured that out so now you've got x1000 in R0 and after adding it to itself you have x2000 in R0, but what happens when you execute your next instruction STI R0,result? Let's let Dan answer this one....

Well that was easy I might have messed up at first but anyone can see that now you just store R0 into your result and then you're finished.

WRONG

Wow lazy Dan have you heard of learning from your mistakes... Apparently not... So lets take a look and see where Danny boy went wrong. What exactly is in result?

Well its x4003... the address of our halt instruction and since its an sti we are actually rewriting this instruction... oh no so I guess we never stop so I guess this program crashes your computer.

GREAT JOB STUDENT! You figured it out and saved your computer from crashing!!!! Maybe now that you are a computer whiz you can help your lazy friend Dan fix his crashed computer!

Problem 2

```
1
2                .ORIG  X4000
3                AND R2,R2,#0
4                ADD R2,R2,#10
5                AND R3,R3,#0
6                LEA  R0,DATA
7   LOOP         LDR  R1,R0,#0
8                AND R1,R1,#01
9                BRZ  SKIP
10               ADD R3,R3,#1
11  SKIP         ADD R0,R0,#1
12               ADD R2,R2,#-1
13               BRP  LOOP
14               HALT
15
16  DATA      .BLKW  #10              ;ASSUME THAT THE 10 VALUES HAVE ALREADY BEEN INITIALIZED
        TO CONTAIN NUMBERS
17               .END
```

So what does this program do? Well lets first see what they give us in that one comment. So we can assume that there are 10 elements in memory that we don't know. Okay that helps us out a little but we still don't know what this program does with the elements. Let us look at our first two instructions we clear R2, okay simple enough, then we add 10 to it... Wait a second we have an array of 10 elements and we are loading R2 with 10 I am going go out on a limb and say that R2 is a counter for which element you are accessing, but let's continue. Then we clear R3, fair enough it's

probably a counter too, and then we load the address of data into R0... Woah there we are loading R0 with the address of data so right now I am pretty sure this program is definitely accessing and doing something with the data in our array. Let's find out what!

So we load R1 with our first element fair enough and then we AND it with #01 and check to see if that bit is set and if it is we increment R3 and if it isn't then we don't and then we increment R0 (the address of the next element in our array) and decrement R2 or what is most likely our element counter and check if its 0 (no more elements!).

So that was weird why did we and our number with #01. Well if you remember if a number is odd it will always have that bit set and if it is even then it will always have that bit clear so we our checking to see if our array element is odd or even easy enough. And since we are incrementing a counter we are not only checking but we are counting how many odds are in this array!

Now you know how to count the odds (and evens) in an array, a very impractical and not very useful skill but nonetheless it's probably something you should know.

Again, and we love to mention this repeatedly, you won't learn anything by just reading. Go out and practice.

3.7 Debugging

The last part of our programming guide is a quick introduction to debugging. Debugging, just like writing programs, is a skill only refined with practice, but we can get you started.

We are not going to go too in depth in testing programs. The key to remember in testing is to ALWAYS TEST THE CORNER CASES, after you test the normal cases. For example, in the state machine example above, you should test inputs outside the state machine diagram (because of the elegance of the solution, there are not that many corner cases besides those inputs). In the first program, in which we added two numbers and then multiplied by 3, the corner cases would be negative numbers or numbers close to the maximum value of an integer on the LC3. Handling these corner cases and making your program robust are what separate complete and industrial quality programs from incomplete ones.

However, let's get back to debugging errors that occur. Luckily, there are many tools in most environments in which you will write programs. Most environments, (and the LC3 simulator is no exception), allow you to step through the program and examine memory locations. You should use these tools to figure out where the problem is. The key is to write modular programs, and debug each module separately. All industrial code projects are too big to write in one giant sitting, and then ran. Projects split up by functions or modules, and are often written by different programs. These modules should continuously be debugged. Once you are certain that there are no errors in one module, you can focus on the next. In general, debugging should take you at least as long as it took to write the code, and is never actually a done process in any significant project.

The hardest, and first, part of debugging is finding the problem and narrowing down where in the program it occurs. From there, fixing the problem should be (mostly) trivial. After you run the program, let's say that there is some sort of error. For our purposes, it doesn't matter whether the error is logical or actually leads to the program crashing. Start stepping through the program. However, before you do, write down what you expect each relevant memory location to contain after each instruction. This step has two purposes. First, you will quickly see if you have any logical bugs in your program - if you missed a step while decomposing the problem. Second, you will be able to see if the instruction or set of instructions does what you want it to do. If the module that you are currently testing does exactly what you want it to do, then you no longer have to worry about it. If you run the program again, simply set a breakpoint past that set of instructions and

debug the next module.

3.8 Conclusion

That concludes the programming part of the packet. Now take a break from reading and go write some programs. The only way to get better at programming is to PRACTICE.