

## Chapter 3

# Digital Logic Structures.

In chapter 1, we stated that computers were constructed out of very large numbers of very simple structures. For example, Intel's Pentium II microprocessor, first offered for sale in 1998, is made up of more than x.x million MOS transistors. The Digital Equipment Corporation Alpha 21264 consists of more than 15 million MOS transistors. In this chapter, we will explain how the MOS transistor works (as a logic element), show how these transistors are connected to form logic gates, and then show how logic gates are interconnected to form larger units that are needed to construct a computer. In chapter 4, we will connect those larger units into a computer.

But first, the transistor.

### 3.1 The transistor

Most computers today, or rather most microprocessors (which form the core of the corresponding computer) are constructed out of MOS transistors. MOS stands for Complementary Metal-oxide Semiconductor. The electrical properties of Metal Oxide Semiconductors is well beyond the scope of what we want to understand in this course. However, it is useful to know that there are two types of MOS transistors: P-type and N-type. They both operate "logically" very similar to the way wall switches works.

Figure 3.1 shows the most basic of electrical circuits: a power supply (in this case, the 120 volts that come into your house), a wall switch, and a lamp (plugged into an outlet in the wall). In order for the lamp to glow, electrons must flow; in order for electrons to flow, there must be a closed circuit from the power supply to the lamp and back to the power supply. The lamp can be turned on and off by simply making or breaking the closed circuit by manipulating the wall switch.

N-type and P-type MOS transistors work the same way. Figure 3.2 shows a schematic rendering of an N-type transistor, (a) by itself, and (b) in a circuit. Note (figure 3.2a) that the transistor has three terminals. If the gate of the transistor is supplied with 2.9 volts, terminals #1 and #2 act like a piece of wire. We say (in the language of electricity)

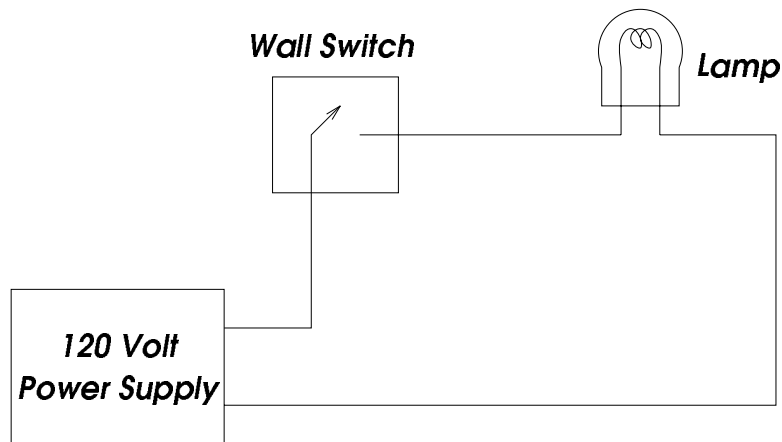


Figure 3.1: A Simple Electric Circuit Showing the Use of a Wall Switch.

that we have a *closed circuit* between terminals #1 and #2. If the gate of the transistor is supplied with 0 volts, terminals #1 and #2 act like a broken connection. We say that between terminals #1 and #2, we have an *open circuit*.

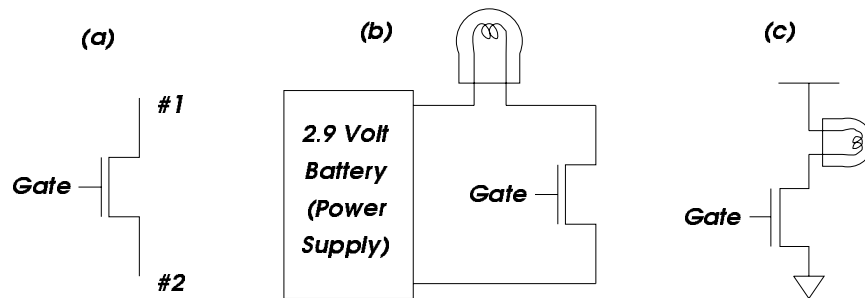


Figure 3.2: The N-type MOS Transistor.

Figure 3.2b shows the N-type transistor in a circuit with a battery and a bulb. When the gate is supplied with 2.9 volts, the transistor acts like a piece of wire, completing the circuit and causing the bulb to glow. When the gate is supplied with 0 volts, the transistor acts like an open, breaking the circuit, and causing the bulb to not glow.

Figure 3.2c is a short-hand notation for describing the circuit of Figure 3.2b. Rather than always show the power supply and the complete circuit, electrical engineers usually show only the terminals of the power supply. The fact that the power supply itself provides the completion of the completed circuit is well understood, and so not usually shown.

The P-type transistor works exactly the opposite of the N-type transistor. Figure 3.3 shows the schematic representation of a P-type transistor. When the gate is supplied with 0 volts, the P-type transistor acts (more or less) like a piece of wire, closing the circuit. When the gate is supplied with 2.9 volts, the P-type transistor acts like an open circuit. Because the P-type and N-type transistors act in this complementary way, we refer to circuits that contain both P-type and N-type transistors as CMOS, for *complementary metal oxide semiconductor*.

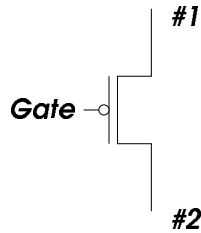


Figure 3.3: A P-type MOS Transistor.

## 3.2 Logic Gates.

One step up from the transistor is the logic gate. That is, we construct basic logic structures out of individual MOS transistors. In Chapter 2 we studied the behavior of the AND, the OR, and the NOT functions. In this chapter we construct transistor circuits that implement these functions. These circuits are called AND, OR, and NOT gates.

### 3.2.1 The NOT gate (or, inverter).

Figure 3.4 shows the simplest logic structure that exists in a computer. It is constructed from two MOS transistors, one P-type and one N-type. Figure 3.4a is the schematic representation of that circuit. Figure 3.4b shows the behavior of the circuit if the input is supplied with 0 volts. Note that the P-type transistor conducts and the N-type transistor does not conduct. The output is, therefore connected to 2.9 volts. On the other hand, if the input is supplied with 2.9 volts, the P-type transistor does not conduct, but the N-type transistor does conduct. The output in this case is connected to ground (i.e., 0 volts). The complete behavior of the circuit can be described by means of a table, as shown in Figure 3.4c. If we replace 0 volts by the symbol 0 and 2.9 volts by the symbol 1, we have the truth table (Figure 3.4d) for the complement or NOT function, which we discussed in Chapter 2.

In other words, we have just shown how to construct an electronic circuit that implements the NOT logic function discussed in Chapter 2. We call this circuit a NOT gate, or an *inverter*.

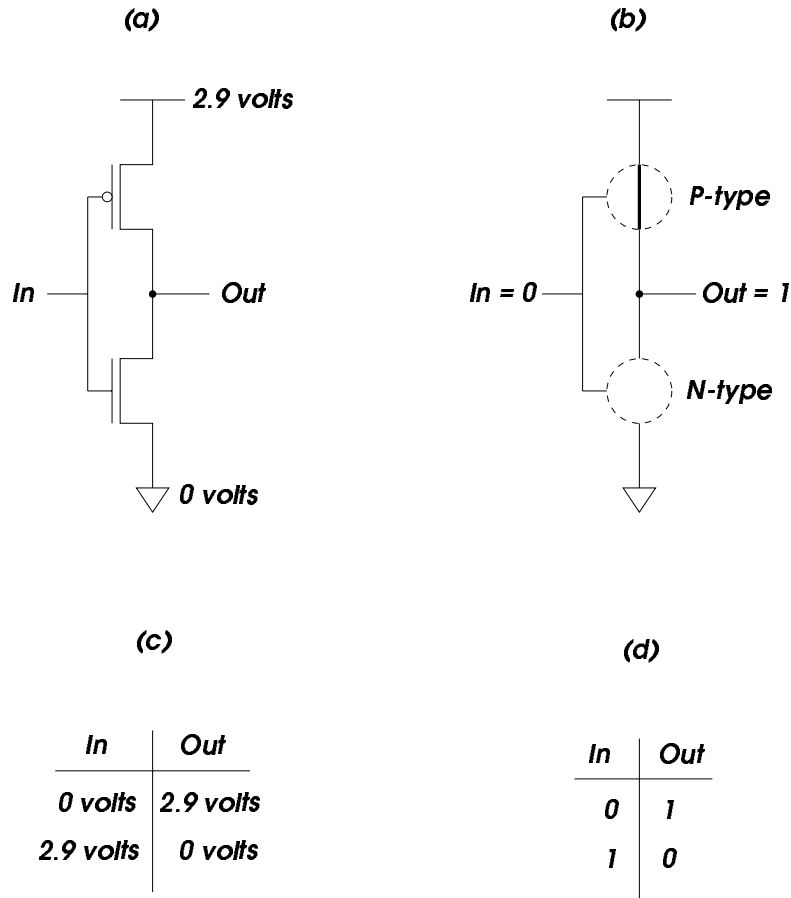


Figure 3.4: A CMOS Inverter.

### 3.2.2 OR and NOR gates.

First examine Figure 3.5. Figure 5a is a schematic containing two P-type and two N-type transistors.

Figure 3.5b shows the behavior of the circuit if *A* is supplied with 0 volts and *B* is supplied with 2.9 volts. In this case, the lower of the two P-type transistors produces an open circuit, and the output *C* is disconnected from the 2.9 volt power supply. However, the left-most N-type transistor acts like a piece of wire, connecting the output *C* to 0 volts.

Note that if both *A* and *B* are supplied with 0 volts, the two P-type transistors conduct, and the output *C* is connected to 2.9 volts. Note, further, that there is no ambiguity here, since both N-type transistors act as open circuits, and so *C* is disconnected from ground.

If either *A* or *B* is supplied with 2.9 volts, the corresponding P-type transistor results in an open circuit. That is sufficient to break the connection from *C* to the 2.9 volt source. However, 2.9 volts supplied to the gate of one of the N-type transistors is sufficient to cause that transistor to conduct, resulting in *C* being connected to ground (i.e., 0 volts).

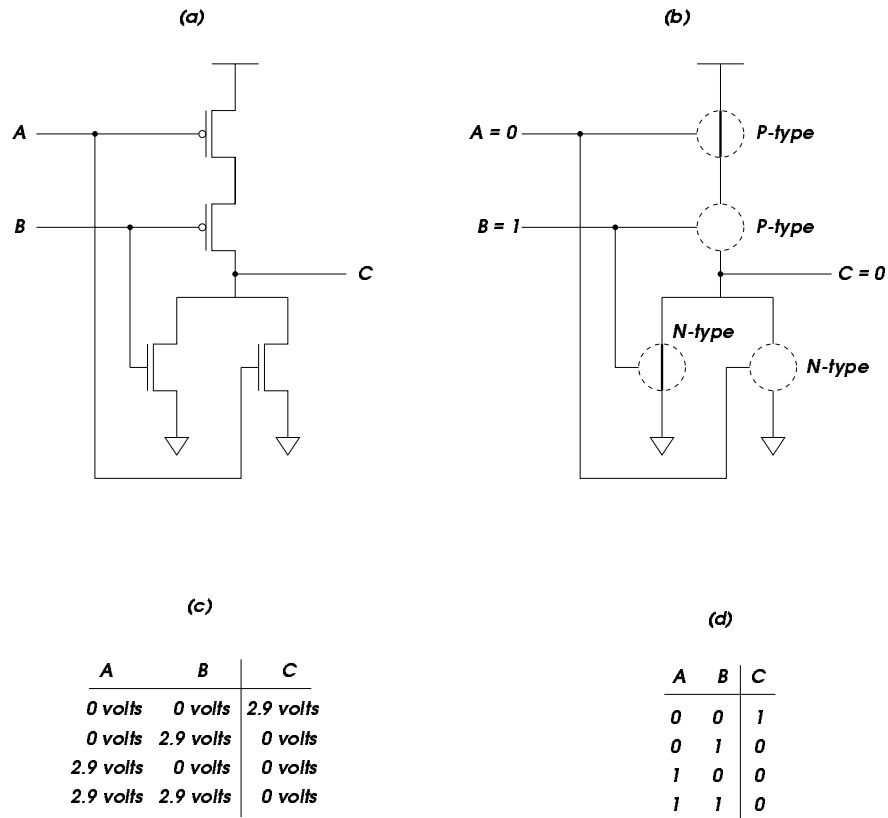


Figure 3.5: The NOR Gate.

Figure 3.5c summarizes the complete behavior of the circuit of Figure 5a. It shows the behavior of the circuit for each of the four pairs of voltages that  $A$  and  $B$  can be supplied with. That is,

$A = 0$  volts,  $B = 0$  volts  
 $A = 0$  volts,  $B = 2.9$  volts  
 $A = 2.9$  volts,  $B = 0$  volts  
 $A = 2.9$  volts,  $B = 2.9$  volts

If we replace the voltages with their logical equivalents, we have the truth table of Figure 3.5d. Note that the output  $C$  is exactly the opposite of the logical OR function discussed in Chapter 2. In fact, it is the NOT-OR function, more typically abbreviated as NOR. We refer to the circuit that implements the logical function NOR as a NOR gate.

If we augment the circuit of Figure 3.5a by adding an inverter at the output, as shown in Figure 3.6a, we have at the output  $D$  the logical function OR. Figure 3.6a is the circuit for an OR gate. Figure 3.6b describes the behavior of this circuit if the input variable  $A$  is set to 0 and the input variable  $B$  is set to 1. Figure 3.6c shows the circuit's truth table.

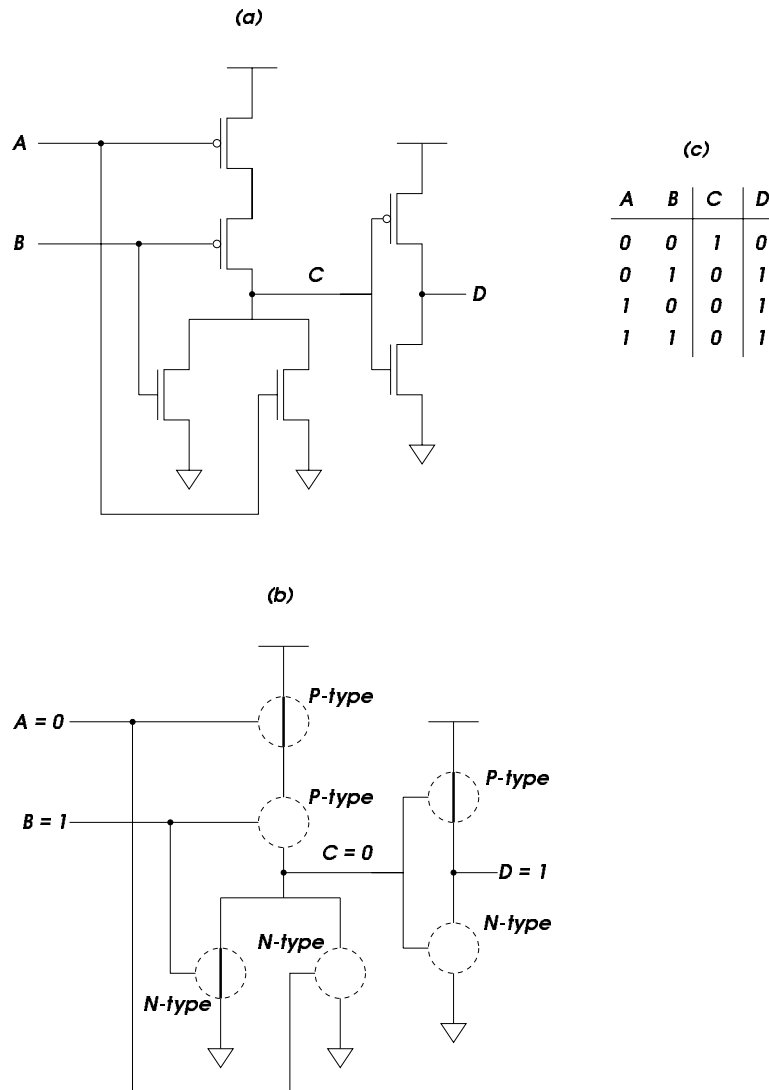


Figure 3.6: The OR Gate.

### 3.2.3 AND and NAND gates.

Next, we will examine Figure 3.7. Note that if either  $A$  or  $B$  is supplied with 0 volts, there is a direct connection from  $C$  to the 2.9 volt power supply. The fact that  $C$  is at 2.9 volts means the N-type transistor whose gate is connected to  $C$  provides a path from  $D$  to ground. Therefore, if either  $A$  or  $B$  is supplied with 0 volts, the output  $D$  of the circuit of Figure 3.7 is 0 volts.

Again, we note that there is no ambiguity. The fact that at least one of the two inputs  $A$  or  $B$  is supplied with 0 volts means that at least one of the two N-type transistors whose gates are connected to  $A$  or  $B$  is an open, and that consequently,  $C$  is disconnected from

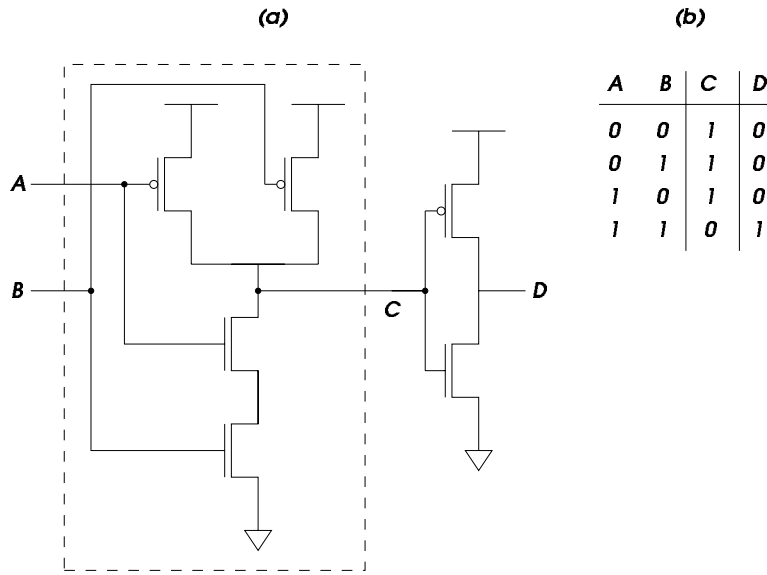


Figure 3.7: The AND Gate.

ground. Furthermore, the fact that  $C$  is at 2.9 volts means the P-type transistor whose gate is connected to  $C$  is open-circuited. Therefore,  $D$  is not connected to 2.9 volts.

On the other hand, if both  $A$  and  $B$  are supplied with 2.9 volts, then both of their corresponding P-type transistors are open. However, their corresponding N-type transistors act like pieces of wire, providing a direct connection from  $C$  to ground. Because  $C$  is at ground, the right-most P-type transistor acts like a closed circuit, forcing  $D$  to 2.9 volts.

Figure 3.7b summarizes in truth table form the behavior of the circuit of Figure 3.7a. Note that the circuit is an AND gate. The circuit shown within the dashed lines (i.e., having output  $C$ ) is a NOT-AND gate, which we generally abbreviate as NAND.

The gates discussed above are very common in digital logic circuits and in digital computers. There are hundreds of thousands of inverters (NOT-gates) in the Pentium II microprocessor. As a convenience, we can represent each of the above gates by standard symbols, as shown in Figure 3.8. The bubble shown in the inverter, NAND and NOR gates signifies the complement (i.e., NOT) function.

From now on, we will not draw circuits showing the individual transistors. Instead, we will use the symbols of Figure 3.8.

### 3.2.4 DeMorgan's Law.

Note (see Figure 3.9a) that one can complement an input before applying it to a gate. Consider the effect on the two input AND gate if we apply the complements of  $A$  and  $B$  as inputs to the gate, and also complement the output of the AND gate. The bubbles at the inputs to the AND gate designate that the inputs  $A$  and  $B$  are complemented before they are used as inputs to the AND gate.

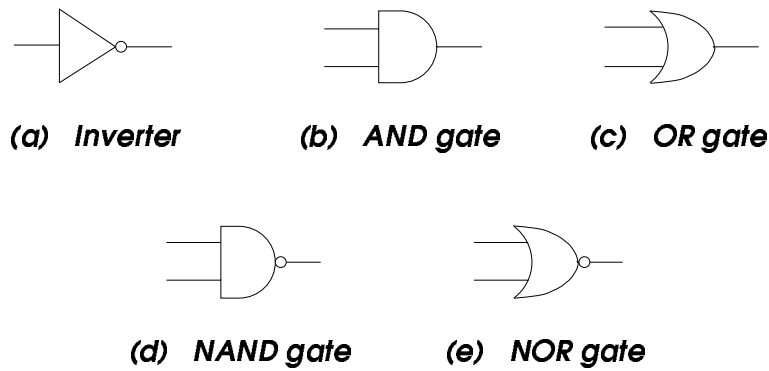


Figure 3.8: Basic Logic Gates.

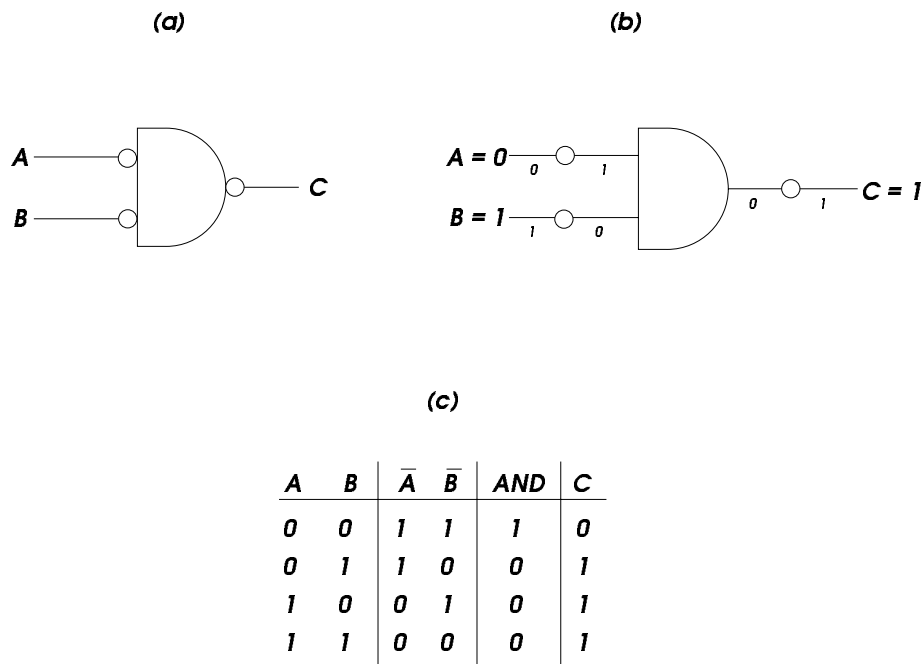


Figure 3.9: DeMorgan's Law.

Figure 3.9b shows the behavior of this structure for the input combination  $A = 0$ ,  $B = 1$ . For ease of representation, we have moved the "bubbles" away from the inputs and the output of the AND gate. That way, we can more easily see what happens to each value as it passes through a bubble.

Figure 3.9c summarizes by means of a truth table the behavior of the logic circuit of Figure 3.9a for all four combinations of input values. Note that the NOT of  $A$  is represented as  $\bar{A}$ .

We can describe the behavior of this circuit algebraically:

$$\overline{\bar{A} \text{ AND } \bar{B}} = A \text{ OR } B$$

This equivalence is known as DeMorgan's Law. Is there a similar result if one inverts both inputs to an OR gate, and then inverts the output?

### 3.2.5 Larger Gates.

Before we leave the topic of logic gates, we should note that the notion of AND, OR, NAND, and NOR gates extends to larger numbers of inputs. One could build a three-input AND gate or a four-input OR gate, for example. An  $n$ -input AND gate has an output value of 1 only if ALL the input variables have values of 1. If any of the  $n$  inputs has a value of 0, the output of the  $n$ -input AND gate is 0. An  $n$ -input OR gate has an output value of 1 if ANY of the input variables has a value of 1. That is, an  $n$ -input OR gate has an output value of 0 only if ALL  $n$  input variables have values of 0.

Figure 3.10 illustrates a three-input AND gate. Figure 3.10a shows its truth table. Figure 3.10b shows the symbol for a three-input AND gate.

(a)				(b)	
<i>A</i>	<i>B</i>	<i>C</i>	<i>OUT</i>	<i>A</i>	—
0	0	0	0	<i>B</i>	—
0	0	1	0	<i>C</i>	—
0	1	0	0		—
0	1	1	0		—
1	0	0	0		—
1	0	1	0		—
1	1	0	0		—
1	1	1	1		—

Figure 3.10: A Three-input AND Gate.

Can you draw a transistor-level circuit for a three-input AND gate? How about a four-input OR gate?

## 3.3 Combinational logic structures.

Now that we understand the workings of the basic logic gates, the next step is to build some of the logic structures that are important components of the microarchitecture of a computer.

There are fundamentally two kinds of logic structures, those that include the storage of information and those that don't. In sections 3.4 and 3.5, we will deal with structures that store information. In this section, we will deal with those that don't. These structures are

sometimes referred to as decision elements. Usually, they are referred to as combinational logic structures, because their outputs are strictly dependent on the combination of input values that are being applied to the structure *right now*. Their outputs are not at all dependent on any past history of information that is stored internally, since no information can be stored internally in a combinational logic circuit.

We will examine a Decoder, a Mux, and a Full Adder.

### 3.3.1 DECODER

Figure 3.11 shows a logic gate description of a two-input decoder. A decoder has the property that it provides at its output exactly one 1 and all the rest 0s. The one output that is logically 1 is the output corresponding to the input pattern that it is expected to detect. In general, decoders have  $n$  inputs and  $2^n$  outputs. We say the output line that detects the input pattern is *asserted*. That is, that output line has the value 1, rather than 0 as is the case for all the other output lines. In Figure 3.11, note that for each of the four possible combinations of inputs  $A$  and  $B$ , exactly one output has the value 1 at any one time. In Figure 3.11b, the input to the Decoder is 10, resulting in the third output line being asserted.

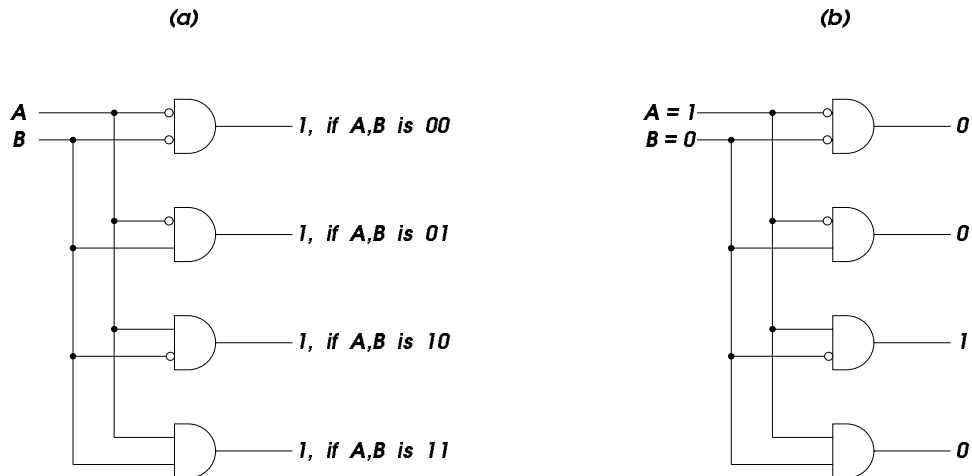


Figure 3.11: A Two-input Decoder.

The decoder is useful in determining how to interpret a bit pattern. We will see in Chapter 5 that the work to be carried out by each instruction in the LC-2 is determined by a four bit pattern, called an opcode, that is part of the instruction. A four-to-16 decoder is a simple combinational logic structure for identifying what work is to be performed by each instruction.

## 3.3.2 MUX

Figure 3.12a shows a gate level description of a two-input multiplexer, more commonly referred to as a MUX. The function of a MUX is to select one of the inputs and connect it to the output. The select signal ( $S$  in Figure 3.12) determines which input is connected to the output.

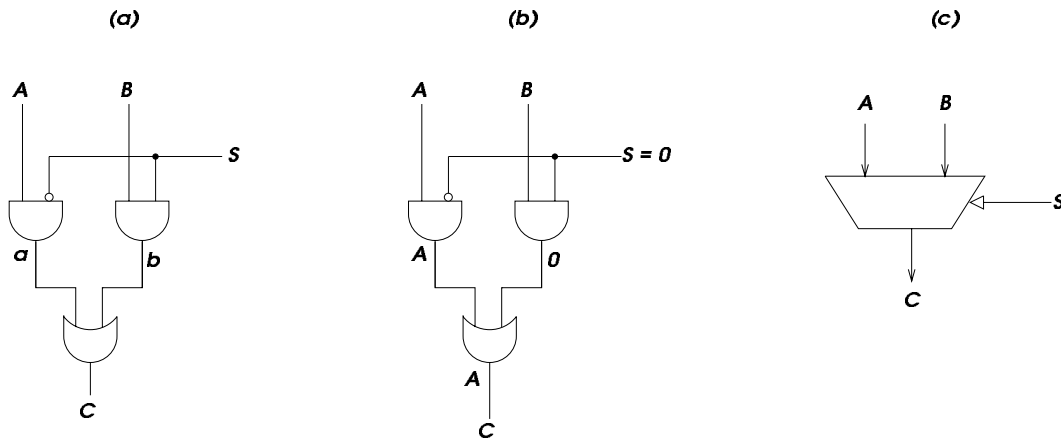


Figure 3.12: A 2-to-1 MUX.

The MUX of Figure 3.12 works as follows: Suppose  $S=0$ , as shown in Figure 3.12b. Since the output of an AND gate is 0 unless all inputs are 1, the output of the right-most AND gate is 0. Also, the output of the left-most AND gate is whatever the input  $A$  is. That is, if  $A=0$ , then the output of the left-most AND gate is 0, and if  $A=1$ , then the output is 1. Since the output of the right-most AND gate is 0, it has no effect on the OR gate. Consequently, the output at  $C$  is exactly the same as the output of the left-most AND gate. The net result of all this is that if  $S=0$ , the output  $C$  is identical to the input  $A$ .

On the other hand, if  $S=1$ , it is  $B$  that is ANDed with 1, resulting in the output of the OR gate having the value of  $B$ .

In summary, the output  $C$  is always connected to either the input  $A$  or the input  $B$  – which one depends on the value of the select line  $S$ . We say  $S$  selects the source of the MUX (either  $A$  or  $B$ ) to be routed through to the output  $C$ .

Figure 3.12c shows the standard representation for a MUX.

In general a MUX consists of  $2^n$  inputs and  $n$  select lines. Figure 3.13a shows a gate-level description of a four-input MUX. It requires two select lines. Figure 3.13b shows the standard representation for a four-input MUX.

Can you construct the gate-level representation for an eight-input MUX? How many select lines must you have?

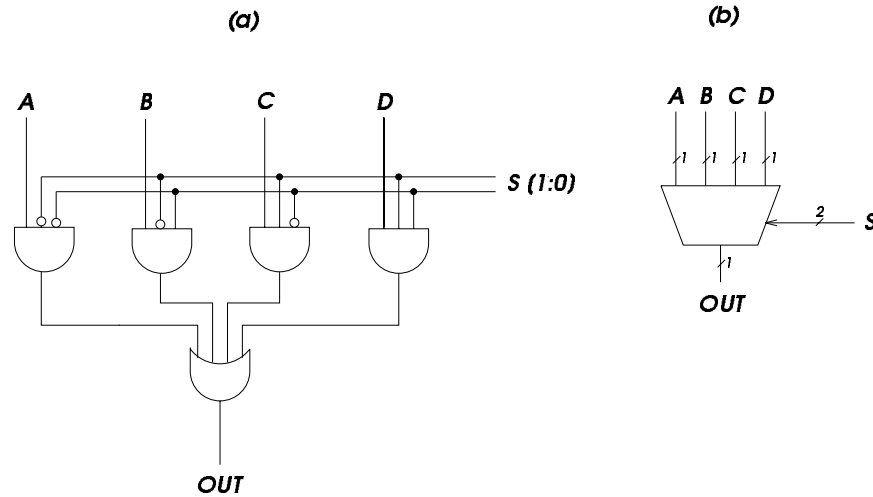


Figure 3.13: A Four-input MUX.

### 3.3.3 FULL ADDER circuit.

In Chapter 2, we discussed binary addition. Recall that a simple algorithm for binary addition is to simply proceed as you have always done in the case of decimal addition, except one gets a carry after 1, rather than after 9.

Figure 3.14 is a truth table that describes the result of binary addition on *one column* of bits within two  $n$ -bit operands. At each column, there are three values that must be added: one bit from each of the two operands and the carry from the previous column. We designate three bits as  $a_i$ ,  $b_i$ , and  $carry_i$ . There are two results, the sum bit ( $s_i$ ) and the carry over to the next column,  $carry_{i+1}$ . Note that if only one of the three bits equals 1, we get a sum of 1, and no carry (i.e.,  $carry_{i+1} = 0$ ). If two of the three bits equal 1, we get a sum of 0, and a carry of 1. If all three bits equal 1, the sum is 3, which in binary addition corresponds to a sum of 1 and a carry of 1.

$a_i$	$b_i$	$CARRY_i$	$CARRY_{i+1}$	$S_i$
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

Figure 3.14: A Truth Table for a Binary Adder.

Figure 3.15 is the gate level description of the truth table of Figure 3.14. We call a circuit that provides three inputs ( $a_i$ ,  $b_i$ , and  $carry_i$ ) and two outputs (the sum bit ( $s_i$ ) and the carry over to the next column  $carry_{i+1}$ ) a *Full Adder*.

Note that each input combination (a specific value associated with each of the three input variables) corresponds to a three-input AND gate with the inputs complemented if the corresponding variable is 0. The output of an AND gate is asserted (equals 1) if the corresponding input combination is present. Thus, to implement the logic circuit to perform the carry function, it is only necessary (as shown in Figure 3.15) (1) to construct AND gates that are asserted for each input combination that produces a carry out, and (2) to connect those AND gates as inputs to a single OR gate. The carry out is 1 if at least two of the three bits being added are 1's. That is, the carry out is 1 if the input combination is 011, 101, 110, or 111. Thus, we first construct AND gates that will produce an output 1 for each of these four input combinations. Since the output of an OR gate is 1 if any of its input combinations is 1, we connect these four AND gates as inputs to a single OR gate. The result is that if the input combination produces a carry out, the output of the OR gate will be 1. The logic circuit to perform the sum function is built in a similar way.

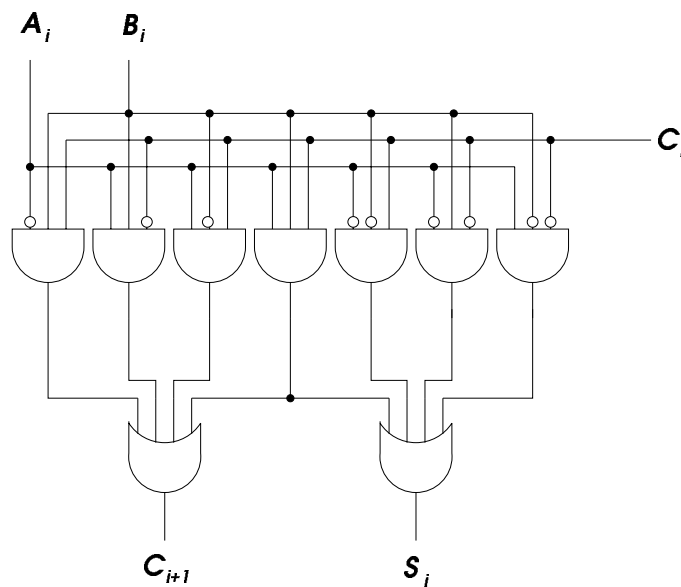


Figure 3.15: Gate Level Description of a Full Adder.

Figure 3.16 illustrates a circuit for adding two 4-bit binary numbers, using four of the full adder circuits of Figure 3.15. Note that the carry out of column  $i$  is an input to the addition performed in column  $i+1$ .

### 3.3.4 Logical Completeness.

Before we leave this section, it is worth noting that any arbitrary truth table can be implemented by a logic circuit, provided sufficiently many AND, OR, and NOT gates are

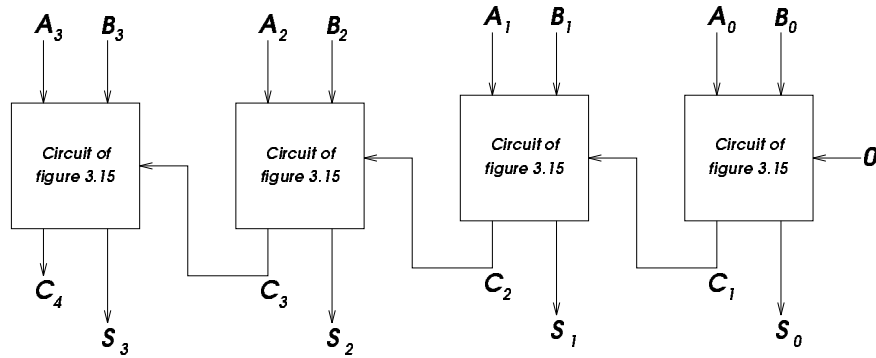


Figure 3.16: A Circuit for Adding Two Four-bit Binary Numbers.

available. We say that the set of gates {AND, OR, NOT} are *logically complete* because we can build a circuit to carry out the specification of any arbitrary truth table without using any other kind of gate. That is, the set of gates {AND, OR, and NOT} is logically complete because a barrel of AND gates, a barrel of OR gates and a barrel of NOT gates are sufficient to build a logic circuit that carries out the specification of any desired truth table. The barrels may have to be big ones, but the point is, we don't need any other kind of gate to do the job.

We can prove this in the following way.

1. Choose any arbitrary specification of a truth table. There will be  $n$  input variables,  $2^n$  input combinations, and  $2^n$  entries in the output column, one for each input combination.
2. For each input combination that results in an output 1, construct an AND gate having  $n$  inputs. For each input that is a 1 in the input combination, apply the corresponding input directly to the input of the AND gate. For each input that is a 0 in the input combination, invert the input before applying it to the corresponding input of the AND gate. The output of this AND gate equals 1 exactly when the inputs to the circuit are the corresponding input combination. The number of AND gates constructed in this step is equal to the number of 1s in the output column of the truth table. Note that this is exactly how we constructed the full adder circuit in the previous section.
3. Apply the outputs of each of the AND gates to an input of one (perhaps large) OR gate. The output of that OR gate will exactly carry out the truth table of step 1, as follows:
  - a. Any input combination of the truth table that results in an output 1 corresponds (by construction in step 2) to an AND gate. The output of that AND gate equals 1. Since the output of that AND gate is an input to the OR gate, the output of the OR gate is likewise 1.
  - b. Any input combination of the truth table that does not result in an output 1 does not correspond to any AND gate. Since the output of each AND is equal to 1 only if the input combination is the one specified for that AND gate, the outputs of all AND gates equal 0. Since all inputs to the OR gate are 0, the output of the OR gate is 0. Done!

### 3.4 Basic Storage Elements.

Recall our statement at the beginning of Section 3.3 that there are two kinds of logic structures, those that involve the storage of information and those that don't. We have discussed three examples of those that don't: the Decoder, the MUX and the Adder. Now we are ready to discuss logic structures that do include the storage of information.

#### 3.4.1 The R-S Latch.

A simple example of a storage element is the R-S latch. It can store one bit of information. The R-S latch can be implemented in many ways, the simplest being the one shown in Figure 3.17. Two two-input NAND gates are connected such that the output of each is connected to one of the inputs of the other. The remaining input S and R are normally held at a logic level 1.

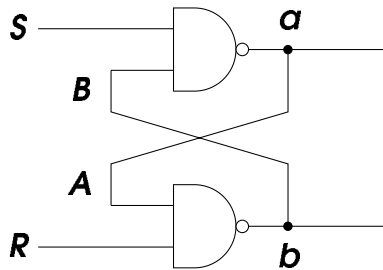


Figure 3.17: An R-S Latch.

The R-S latch works as follows: We start with what we call the quiescent (or quiet) state, where inputs S and R both have logic value 1. We consider first the case where the output a is 1. Since that means the input A equals 1, (and we know the input R equals 1 since we are in the quiescent state), the output b must be 0. That, in turn, means the input B must be 0, which results in the output a equal to 1. As long as the inputs S and R remain 1, the state of the circuit will not change. We say the R-S latch stores the value 1 (the value of the output a).

If, on the other hand, we assume the output a is 0, then the input A must be 0, and the output b must be 1. This in turn results in the input B equal to 1, and combined with the input S equal to 1 (again due to quiescence) results in the output a equal to 0. Again, as long as the inputs S and R remain 1, the state of the circuit will not change. In this case, we say the R-S latch stores the value 0.

The latch can be set to 1 by momentarily setting S to 0, provided we keep the value of R at 1. Similarly, the latch can be set to 0 by momentarily setting R to 0, provided we keep the value of S at 1. We use the term *set* to denote setting a variable to 0 or 1, as in *set to 0* or *set to 1*. In addition, we often use the term *clear* to denote the act of setting a variable to 0.

If we clear  $S$ , then  $a$  equals 1, which in turn causes  $A$  to equal 1. Since  $R$  is also 1, the output at  $b$  must be 0. This causes  $B$  to be 0, which in turn makes  $a$  equal 1. If we now return  $S$  to 1, it does not affect  $a$ , since  $B$  is also 0, and only one input to a NAND gate must be 0 in order to guarantee that the output of the NAND gate is 1. Thus the latch continues to store a 1 long after  $S$  returns to 1.

In the same way, we can clear the latch (set the latch to 0) by momentarily setting  $R$  to 0.

We should also note that in order for the R-S latch to work properly, one must take care that it is never the case that both  $S$  and  $R$  are allowed to be set to 0 at the same time. If that does happen, the outputs  $a$  and  $b$  are both 1, and the final state of the latch depends on the electrical properties of the transistors making up the gates and not on the logic being performed. How the electrical properties of the transistors will determine the final state in this case is a subject we will have to leave for a later semester.

### 3.4.2 The gated D latch.

To be useful, it is necessary to control when a latch is set and when it is cleared. A simple way to accomplish this is with the gated latch.

Figure 3.18 shows a logic circuit that implements a gated D latch. It consists of the R-S latch of Figure 3.17, plus two additional gates that allow the latch to be set to the value of  $D$ , but **only** when  $WE$  is asserted.  $WE$  stands for Write Enable. When  $WE$  is not asserted (i.e., when  $WE$  equals 0), the outputs  $S$  and  $R$  are both equal to 1. Since  $S$  and  $R$  are also inputs to the R-S latch, if they are kept at 1, the value stored in the latch remains unchanged, as we explained above,

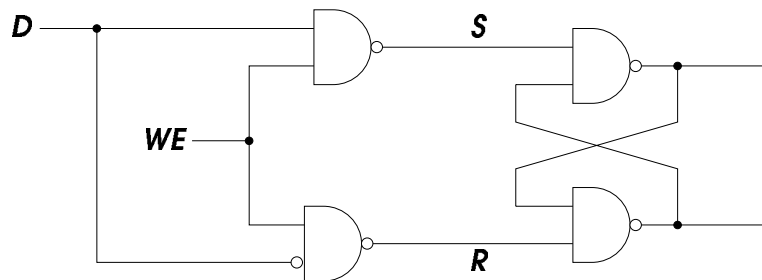


Figure 3.18: A Gated D Latch.

When  $WE$  is momentarily asserted (i.e., set to 1), exactly one of the outputs  $S$  or  $R$  is set to 0, depending on the value of  $D$ . If  $D$  equals 1, then  $S$  is set to 0. If  $D$  equals 0, then both inputs to the lower NAND gate are 1, resulting in  $R$  being set to 0. As we saw above, if  $S$  is set to 0, the R-S latch is set to 1. If  $R$  is set to 0, the R-S latch is set to 0. Thus, the R-S latch is set to 1 or 0 according to whether  $D$  is 1 or 0. When  $WE$  returns to 0,  $S$  and  $R$  return to 1, and the value stored in the R-S latch persists.

### 3.4.3 A Register.

We have already seen in Chapter 2 that it is useful to deal with values consisting of more than 1 bit. In chapter 5, we will introduce the LC-2 computer, where most values are represented by 16 bits. It is useful to be able to store these larger numbers of bits as a self-contained unit. The *register* is a structure that stores a number of bits, taken together as a unit. That number can be as large as is useful or as small as 1. In the LC-2, we will need many 16-bit registers, and also a few one-bit registers. We will see in Figure 3.22, which describes the internal structure of the LC-2, that PC, IR, and MAR are all 16-bit registers, and that N, Z, and P are all one-bit registers.

Figure 3.19 shows a 4-bit register made up of 4 gated D latches. The four-bit value stored in the register is  $Q_3, Q_2, Q_1, Q_0$ . The value  $D_3, D_2, D_1, D_0$  can be written into the register when WE is asserted.

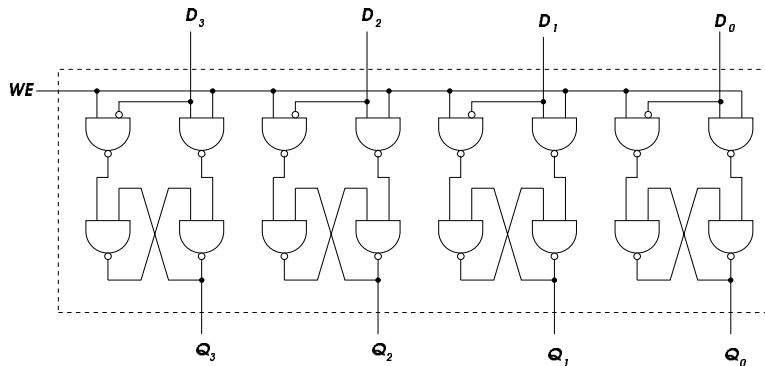


Figure 3.19: A Four-bit Register.

Note: a common shorthand notation to describe a sequence of bits that are numbered as above is  $Q[3:0]$ . That is, each bit is assigned its own bit number. The right-most bit is bit[0], and the numbering continues from right to left. If there are  $n$  bits, the left-most bit is bit[ $n-1$ ]. For example, in the following 16-bit pattern,

0011101100011110

bit[15] is 0, bit[14] is 0, bit[13] is 1, bit[12] is 1, etc.

We can designate a subunit of this pattern with the notation  $Q[l:r]$ , where  $l$  is the left-most bit in the sub-unit and  $r$  is the right-most bit in the sub-unit. We call such a sub-unit a *field*.

In the above 16-bit pattern, if  $A[15:0]$  is the entire 16-bit pattern, then, for example:

$A[15:12]$  is 0011  
 $A[13:7]$  is 1110110  
 $A[2:0]$  is 110  
 $A[1:1]$  is 1

We should also point out that the numbering scheme from right to left is purely arbitrarily. We could have just as easily designated the left-most bit as bit[0] and numbered them from left to right. Indeed, many people do. So, it is not important whether the numbering scheme is left to right or right to left. But it is important that the bit numbering be consistent in a given setting, that is, that it always be done the same way. In our work, we will always number bits from right to left.

## 3.5 The Concept of Memory

We now have all the tools we need to describe what is perhaps the most important structure in the electronic digital computer, its *memory*. We will see in Chapter 4 how memory fits into the basic scheme of computer processing, and you will see throughout the rest of the book and indeed the rest of your work with computers how important the concept of memory is to computing.

Memory is made up of a (usually large) number of locations, each uniquely identifiable and each having the capability to store a value. We refer to the unique identifier associated with each memory location as its *address*. We refer to the number of bits of information stored in each location as its *addressibility*.

For example, an advertisement for a personal computer might say, “This computer comes with 16 megabytes of memory.” Actually, most ads generally use the abbreviation 16 MB. This statement means, as we will explain momentarily, that the computer system includes 16 million memory locations, each containing one byte of information.

### 3.5.1 Address Space.

We refer to the total number of uniquely identifiable locations as the memory’s *address space*. A 16MB memory, for example, refers to a memory that consists of 16 million uniquely identifiable memory locations.

Actually, the number 16 million is only an approximation, due to the way we identify memory locations. Since everything else in the computer is represented by sequences of 0s and 1s, it should not be surprising that memory locations are identified by binary addresses, as well. With  $n$  bits of address, we can uniquely identify  $2^n$  locations. Ten bits provide 1024 locations, which is approximately one thousand. If we have 20 bits to represent each address, we have  $2^{20}$  uniquely identifiable locations, which is approximately one million. Thus “16 mega” really corresponds to the number of uniquely identifiable locations that can be specified with 24 address bits. We say the address space is  $2^{24}$ , which is *exactly* 16,777,216 locations, rather than 16,000,000 that we colloquially refer to it as.

### 3.5.2 Addressability.

The number of bits stored in each memory location is the memory's addressability. A 16 megabyte memory is a memory consisting of 16,777,216 memory locations, each containing one byte (i.e., 8 bits) of storage. Most memories are byte-addressable. The reason is historical; most computers got their start processing data, and one character stroke on the keyboard corresponds to one 8-bit ASCII character, as we learned in Chapter 2. If the memory is byte-addressable, then each ASCII code occupies one location in memory. Uniquely identifying each byte of memory allowed individual bytes of stored information to be changed easily.

Many computers that have been designed specifically to perform large scientific calculations are 64-bit addressable. This is due to the fact that numbers used in scientific calculations are frequently represented as 64 bit floating point quantities. Recall we discussed the floating point data type in Chapter 2. Since scientific calculations are likely to use numbers that require 64 bits to represent them, it is reasonable to design a memory for such a computer that stores one such number in each uniquely identifiable memory location.

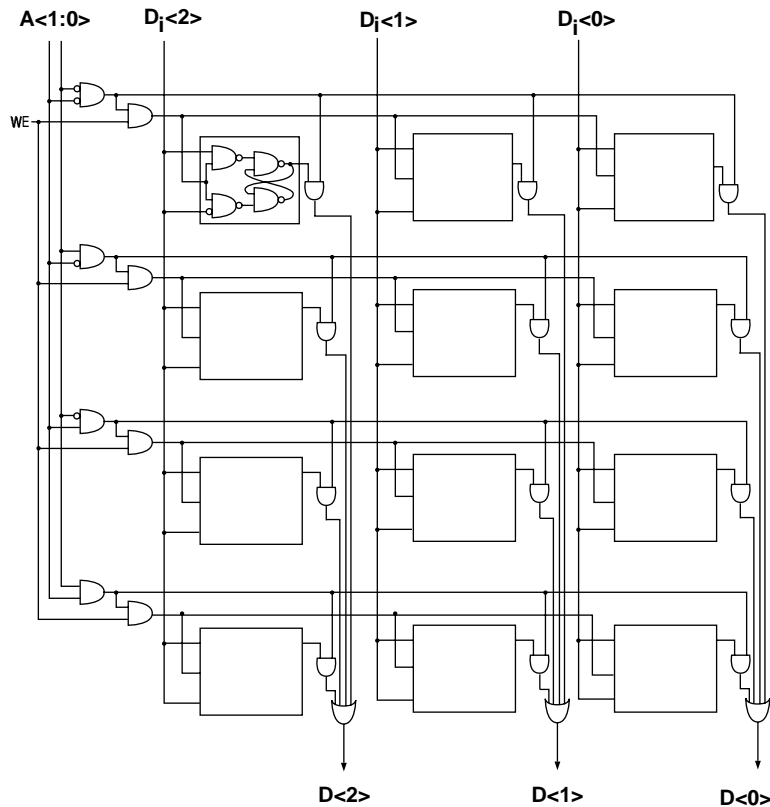
### 3.5.3 A $2^2$ by 3 bit Memory.

Figure 3.20 illustrates a memory of size  $2^2$  by 3 bits. That is, the memory has an address space of four locations, and an addressability of 3 bits. A memory of size  $2^2$  requires 2 bits to specify the address. A memory of addressability 3 stores 3 bits of information in each memory location.

Accesses of memory require decoding the address bits. Note that the Address Decoder takes as input  $A[1:0]$  and asserts exactly one of its four outputs, corresponding to the *word line* being addressed. In Figure 3.20, each row of the memory corresponds to a unique 3-bit word; therefore, the term *word line*. Memory can be read by applying the address  $A[1:0]$ , which asserts the word line to be read. Note that each of the 12 bits in this memory is ANDed with its word line and then ORed with the corresponding bits of the other words. Since only one word line can be asserted at a time, the output of each *bit line* is the value stored in the corresponding bit of the word line that is asserted.

Figure 3.21 shows the process of reading location 3. The code for 3 is 11. The address  $A[1:0] = 11$  is decoded, and the bottom word line is asserted. Note that the three other decoder outputs are not asserted. That is, they have the value 0. The value stored in location 3 is 101. These three bits are each ANDed with their word line producing the bits 101 which are supplied to the three output OR gates. Note that all other inputs to the OR gates are 0, since they have been produced by ANDing with unasserted word lines. The result is that  $D[2:0] = 101$ . That is, the value stored in location 3, is output by the OR gates.

Memory can be written in a similar fashion. The address specified by  $A[1:0]$  is presented to the Address Decoder, resulting in the correct word line being asserted. With WE asserted as well, the three bits  $D_i[2:0]$  can be written into the three gated latches corresponding to that word line.

Figure 3.20: A  $2^2$  by 3 bit Memory.

### 3.6 The Data Path of the LC-2.

(Preview of Coming Attractions)

In chapter 5, we will specify a computer, which we call the LC-2, and you will have the opportunity to write computer programs to execute on the LC-2. Figure 3.22 is a block diagram of what we call the *Data Path* of the LC-2. The Data Path consists of all the logic structures that combine to process information in the core of the computer. Right now, the interconnection of all the units in Figure 3.22 is still a mystery. Note however, that you do already know how many of the elements in the data path work, and furthermore, you know how those elements are constructed from gates. For example, PC, IR, MAR, and MDR are registers and store 16 bits of information, each. Each wire that is labeled with a cross-hatch 16 represents 16 wires, each carrying one bit of information. N, Z, and P are one-bit registers. There are three MUXes, one supplying a 16-bit value to the PC register. In Chapter 5 we will see why these elements must be connected as shown in order to execute the programs written for the LC-2 computer.

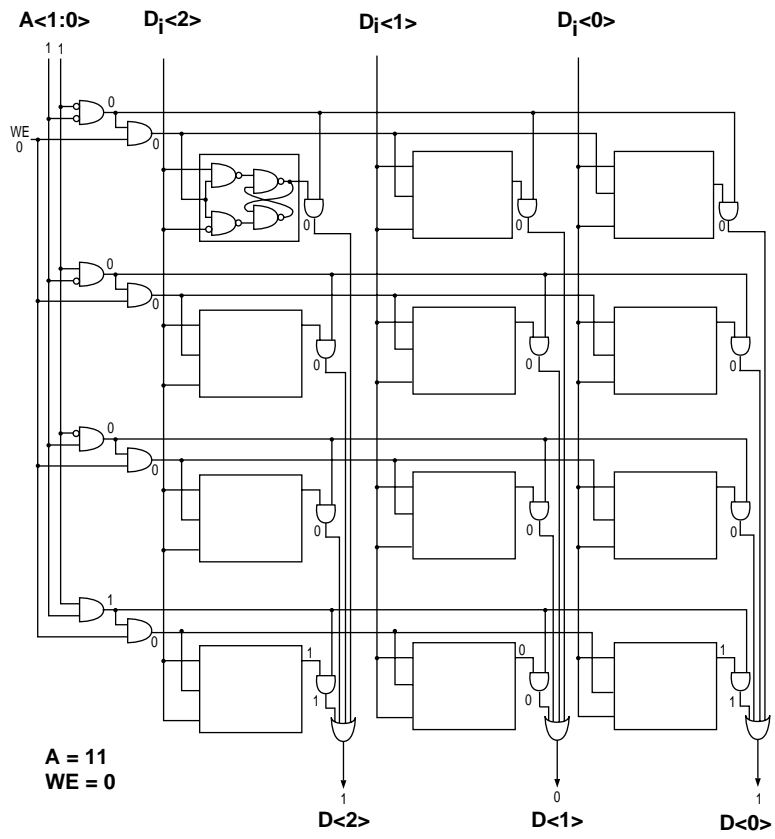


Figure 3.21: Reading location 3 in our  $2^2$  by 3 bit Memory.

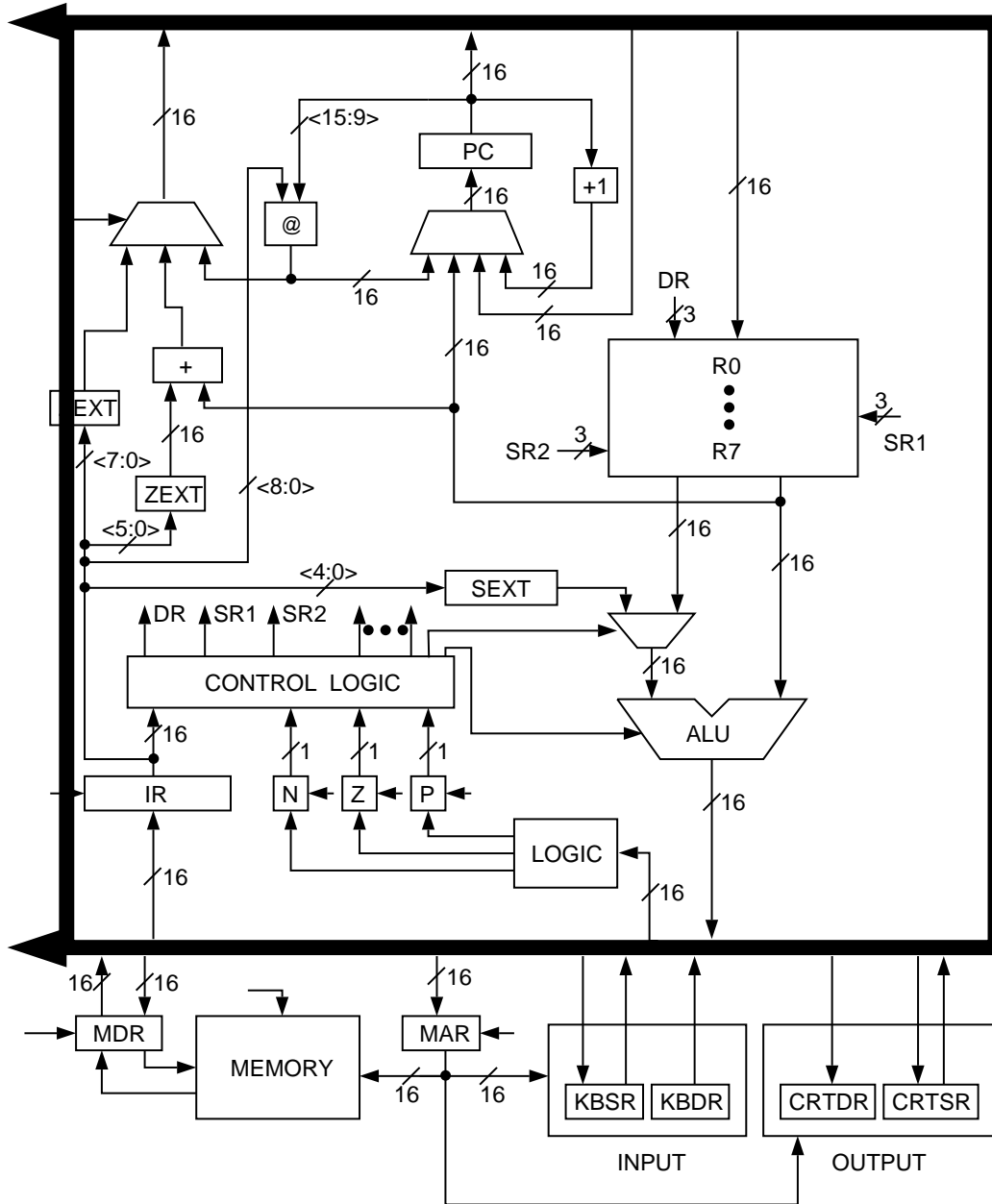


Figure 3.22: The Data Path of the LC-2 Computer.

### 3.7 Exercises.

3.1 Give the truth table for the circuit below. The first line is given as an example:

$$Q1 = \sim(\sim X \mid (X \& Y \& Z))$$

$$Q2 = \sim((Y \mid Z) \& (X \& Y \& Z))$$

X	Y	Z	Q1	Q2
0	0	0	0	1

3.2 1. If A and B are 4-bit unsigned binary numbers, 0111 and 1011, respectively, complete the table obtained when using the 2-bit full adder in the coursepack to calculate the sum of A and B.

C(in)	----	----	----	----
A	0	1	1	1
B	1	0	1	1
-----				
S	----	----	----	----
C(out)	----	----	----	----

2. Check you answer by adding together the decimal values of A and B. Are the answers the same?

3. Why or why not?
- 3.3 If a computer has 8-byte addressability and needs 3 bits to access a location in memory, what is the total size of memory? (Answer in bytes)
- 3.4 Using the diagram of the 4-entry, 3-bit memory from the coursepack:
1. If I wish to read from memory location 4, what must the values of  $A_i1_i$ ,  $A_i0_i$ , and WE be?
  2. If I wanted to change the number of entries in the memory from 4 to 60, how many address lines would I need in total? Would the addressability of the memory change?
  3. Suppose the Program Counter's size was the minimum number of bits needed to address all 60 memory locations in part b. How many additional memory locations could be added without having to alter the number of bits in the PC?
- 3.5 (f95.e1.1c) A memory is addressed by 22 bits. The memory is 3 bit addressible. How many bits of storage does the memory contain?
- 3.6 Implement the following functions using logic gate symbols (AND, OR, NOT). You do not need to draw the transistor-level diagram. Inputs : A, B. Output : F
1. F has the value one only if A has the value zero and B has the value one.
  2. F has the value one only if A has the value one and B has the value zero.
  3. Use your answers to (a) and (b) to implement a one-bit adder, which has the following truth table:
- | A | B | SUM |
|---|---|-----|
| 0 | 0 | 0   |
| 0 | 1 | 1   |
| 1 | 0 | 1   |
| 1 | 1 | 0   |
4. Is it possible to create a four-bit adder (a circuit that will correctly add two four-bit quantities) using only four copies of the logic diagram from part (c)? If not, what information is missing? Hint: When  $A = 1$  and  $B = 1$ , a SUM of 0 is produced. What information is thrown away?
- 3.7 (f95.e3.1e) Logic circuit 1 has inputs A,B, and C. Logic circuit 2 has inputs A and B. Both logic circuits have an output D. There is a fundamental difference between the behavior characteristics of these two circuits. What is it? [Hint: What happens when the voltage at input A goes from 0 to 1 in both circuits?]

circuit 1 is a MUX             $D = A*C + B*\sim C$ ;  
 circuit 2 is a RS-latch    D output, A,B inputs

- 3.8 (f96.e3.1c) You know a byte is 8 bits. We call a four bit unit of storage a nibble. If a byte addressible memory has a 14 bit address, how many nibbles of storage are in this memory?
- 3.9 (w98.e1.3) Below is a logic circuit that appears in many of today's processors. Each of the boxes is a full adder circuit. What does the value on the wire X do? That is, what is the difference in the output of this circuit if  $X=0$  versus  $X=1$ .

<<  $X = 0$  then output is  $A + B$ ,  $X = 1$ ,  $A - B$  >>