

# CSE241: Digital Systems

Ben Miller

Instructed by: Jennifer Winikus



## Contents

<b>1</b>	<b>1/27/2020</b>	<b>1</b>
1.1	Lecture Notes . . . . .	1
1.2	Assigned Readings . . . . .	1
<b>2</b>	<b>1/29/2020</b>	<b>3</b>
2.1	Lecture Notes . . . . .	3
2.2	Assigned Readings . . . . .	4
<b>3</b>	<b>1/31/2020</b>	<b>5</b>
3.1	Lecture Notes . . . . .	5
3.2	Assigned Readings . . . . .	6
<b>4</b>	<b>2/3/2020</b>	<b>9</b>
4.1	Lecture Slides . . . . .	9
4.2	Assigned Readings . . . . .	12
<b>5</b>	<b>2/5/2020</b>	<b>14</b>
5.1	Lecture Slides . . . . .	14
5.2	Assigned Readings . . . . .	16
<b>6</b>	<b>2/7/2020</b>	<b>18</b>
6.1	Lecture Notes . . . . .	18
6.2	Assigned Readings . . . . .	19
<b>7</b>	<b>2/10/2020</b>	<b>23</b>
7.1	Lecture Notes . . . . .	23
<b>8</b>	<b>2/12/2020</b>	<b>25</b>
8.1	Lecture Notes . . . . .	25
8.2	Assigned Readings . . . . .	27
<b>9</b>	<b>2/14/2020</b>	<b>30</b>
9.1	Lecture Slides . . . . .	30
9.2	Assigned Readings . . . . .	32
<b>10</b>	<b>2/17/2020</b>	<b>34</b>
10.1	Lecture Slides . . . . .	34
<b>11</b>	<b>2/19/2020</b>	<b>36</b>
11.1	Lecture Slides . . . . .	36
<b>12</b>	<b>2/21/2020</b>	<b>38</b>
12.1	Lecture Slides . . . . .	38
12.2	Assigned Readings . . . . .	41
<b>13</b>	<b>2/24/2020</b>	<b>46</b>
13.1	Lecture Slides . . . . .	46
<b>14</b>	<b>2/26/2020</b>	<b>49</b>
14.1	Lecture Slides . . . . .	49

<b>15</b>	<b>2/28/2020</b>	<b>52</b>
15.1	Lecture Slides . . . . .	52
<b>16</b>	<b>3/02/2020</b>	<b>55</b>
16.1	Lecture Slides . . . . .	55
16.2	Assigned Readings . . . . .	60
<b>17</b>	<b>3/04/2020</b>	<b>64</b>
17.1	Lecture Slides . . . . .	64
17.2	Assigned Readings . . . . .	68
<b>18</b>	<b>3/06/2020</b>	<b>71</b>
18.1	Lecture Slides . . . . .	71
<b>19</b>	<b>3/22/2020</b>	<b>75</b>
19.1	Time . . . . .	75
19.1.1	Lecture Slides . . . . .	75
19.1.2	Assigned Reading . . . . .	75
19.2	Documentation . . . . .	78
19.2.1	Lecture Slides . . . . .	78
19.2.2	Assigned Reading . . . . .	80
19.3	Verilog Introduction . . . . .	83
19.3.1	Lecture Slides . . . . .	83
19.3.2	Assigned Reading . . . . .	84
19.4	Verilog Terminology . . . . .	86
19.4.1	Lecture Slides . . . . .	86
19.4.2	Assigned Reading . . . . .	88
<b>20</b>	<b>3/29/2020</b>	<b>91</b>
20.1	MSI . . . . .	91
20.2	Shifting . . . . .	91
20.3	Rotating . . . . .	91
20.4	Mux Intro . . . . .	91
<b>21</b>	<b>4/05/2020</b>	<b>93</b>
21.1	Decoders . . . . .	93
21.1.1	Lecture Slides . . . . .	93
21.1.2	Assigned Reading . . . . .	93
21.2	Encoders . . . . .	94
21.3	Continuous Assignment . . . . .	95
21.3.1	Lecture Slides . . . . .	95
21.3.2	Assigned Reading . . . . .	95
21.4	Behavioral Verilog . . . . .	97
21.4.1	Lecture Slides . . . . .	97
21.4.2	Assigned Reading . . . . .	98
21.5	Test Benches . . . . .	102
21.5.1	Lecture Slides . . . . .	102
21.5.2	Assigned Reading . . . . .	103

<b>22</b>	<b>4/12/2020</b>	<b>104</b>
22.1	What Is Sequential Logic? . . . . .	104
22.1.1	Lecture Slides . . . . .	104
22.1.2	Assigned Reading . . . . .	104
22.2	Latches . . . . .	105
22.3	Flip-Flops . . . . .	106
22.3.1	Lecture Slides . . . . .	106
22.3.2	Assigned Reading . . . . .	106
22.4	Sequential MSI Devices . . . . .	109
22.4.1	Lecture Slides . . . . .	109
22.4.2	Assigned Reading . . . . .	109
<b>23</b>	<b>4/19/2020</b>	<b>113</b>
23.1	What Is an FSM? . . . . .	113
23.2	FSM Design Algorithm . . . . .	113
<b>24</b>	<b>4/26/2020</b>	<b>115</b>
24.1	Implementation With DFF Review . . . . .	115
24.2	FSM Verilog Part 2 . . . . .	115
24.3	Implementation With TFF . . . . .	115
24.4	Memory Devices . . . . .	115

# 1 1/27/2020

## 1.1 Lecture Notes

- Don't cheat.

## 1.2 Assigned Readings

- Analog devices and systems process time-varying signals that can take on potentially any kind of valid across any measurable physical quantity.
- Digital circuits and systems act the same way, with the key difference being we pretend they don't. A digital signal is modeled as taking on only two discrete values: 0 and 1.
- There are many reasons to prefer digital circuits over analog ones, including easily reproducible results, great ease of design, expanded flexibility and functionality, and high programmability. Digital circuits are also faster, cheaper, and technologically advancing much faster than analog circuits.
- Despite the numerous benefits to digital circuits, we live in an analog world. Because most, if not all, physical quantities in real circuits are infinitely variable, we could use a physical quantity such as a signal voltage to represent a real number.
- However, stability and accuracy in physical quantities are difficult to obtain in real circuits, potentially being affected by manufacturing variations, temperature, cosmic rays, etc., causing analog values to occasionally be inaccurate. Even worse, many mathematical and logical operations can be difficult or even impossible to perform with analog quantities.
- We hide these pitfalls of our analog world using digital logic, where the infinite set of values for a physical quantity are mapped into two subsets. These two subsets correspond to only two numbers, or logic values: 0 and 1. This allows digital logic circuits to be analyzed and designed functionally.
- A logic value is often called a binary digit, or a bit. If an application would require more than these two discrete values, additional bits can be used, with a set of  $n$  bits representing  $2^n$  different values.
- With most phenomena, there is an undefined region between the 0 and 1 states. For example, picture a capacitor with a light bulb. At 0.0 V, the light is off and the capacitor is uncharged. At 1.0 V, the light is dimly lit and the capacitor is slightly charged. The undefined region exists to categorically define the 0 and 1 states, as if the boundaries are too close noise can easily corrupt results.
- The leftmost digit of a number is called the high-order or most significant digit. Conversely, the rightmost number is called the low-order or least significant digit.

- Digital circuits have signals that are normally in one of only two states, such as high or low, charged or discharged, and on or off. The signals in these circuits are interpreted to represent binary digits or bits that have one value: 0 and 1.
- The leftmost bit of a binary number is called the high-order or most significant bit. Conversely, the rightmost bit is called the low-order or least significant bit.
- The octal number system uses a base 8 counting system, while the hexadecimal or hex number system uses base 16.
- The octal system needs 8 digits, so it uses the digits 0-7 of the decimal system. The hexadecimal system needs 16 digits, so it uses the decimal digits 0-9 and the letters A-F.
- Computers primarily process information in groups of 8-bit bytes. In the hexadecimal system, two hex digits represent an 8-bit byte, and  $2n$  hex digits represent an  $n$ -byte word. In this context, a 4-bit hexadecimal digit is sometimes called a nibble.
- Converting binary numbers to decimal numbers is easy, and looks like this.
  - $1CE8_{16} = 1 \cdot 16^3 + 12 \cdot 16^2 + 14 \cdot 16^1 + 8 \cdot 16^0 = 7400_{10}$
  - $F1A3_{16} = 15 \cdot 16^3 + 1 \cdot 16^2 + 10 \cdot 16^1 + 3 \cdot 16^0 = 61859_{10}$
  - $436.5_8 = 4 \cdot 8^2 + 3 \cdot 8^1 + 6 \cdot 8^0 + 5 \cdot 8^{-1} = 286.625_{10}$
- Converting decimal numbers to binary is slightly more complicated, and looks like this.
  - $179 \div 2 = 89$  remainder 1 (LSB)
  - $89 \div 2 = 44$  remainder 1
  - $44 \div 2 = 22$  remainder 0
  - $22 \div 2 = 11$  remainder 0
  - $11 \div 2 = 5$  remainder 1
  - $5 \div 2 = 2$  remainder 1
  - $2 \div 2 = 1$  remainder 0
  - $1 \div 2 = 0$  remainder 1 (MSB)
  - Thus,  $179_{10}$  in binary is  $10110011_2$ .
- This works for other number systems too.
  - $467 \div 8 = 58$  remainder 3 (LSB)
  - $58 \div 8 = 7$  remainder 2
  - $7 \div 8 = 0$  remainder 7 (MSB)
  - Thus,  $467_{10}$  in octal is  $723_8$ .
  - $3417 \div 16 = 213$  remainder 9 (LSB)
  - $213 \div 16 = 13$  remainder 5
  - $13 \div 16 = 0$  remainder 13
  - Thus,  $3417_{10}$  in hexadecimal is  $D59_{16}$ .

## 2 1/29/2020

### 2.1 Lecture Notes

- Analog signals can take any value across a continuous range of current, voltage, etc.
- While digital circuits can be analog too, they don't, because digital works better for their purpose. Digital signals restrict themselves to two discrete values: 0 and 1.
- Systems can be represented digitally. For example, consider an image, which is just thousands of pixels represented by bits.
- Analog signals are our physical reality. Things in analog signals are continuously variable. The design of an analog signal is extremely complex, and stability and accuracy is very difficult.
- In the beginning, we used vacuum tubes to go from analog to digital. However, due to vacuum tubes being inefficient, we eventually moved to transistors.

- Problem 2.1.1: Convert 37 to binary.

37/2	18	R1
18/2	9	R0
9/2	4	R1
4/2	2	R0
2/2	1	R0
1/2	0	R1
Answer: b101001		

- Problem 2.1.2: Convert b10111 to decimal.

$$1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

$$2^4 + 2^2 + 2^1 + 2^0$$

$$16 + 4 + 2 + 1$$

Answer: 23

- In the hex number system, each group of four becomes a single hex value. For example, b10100111 is xA7.

- Problem: Convert b101110101 to hex.

- First, start from the right and collect the four rightmost bits: 0101.
- Then, grab the next four bits: 0111.
- Finally, grab the remaining bit: 1. We need to put 0's in the front to pad this out to four bits, giving us 0001.
- Convert these groups to values. 0001 is equivalent to 1, 0111 is equivalent to 7, and 0101 is equivalent to 5.

Answer: x175



- Problem 2.1.3: Convert x3FA2 to binary.

- 3 in binary is 0011.
- F in binary is 1111.
- A in binary is 1010.
- 2 in binary is 0010.

Answer: 0011111110100010

## 2.2 Assigned Readings

- Addition and subtraction using binary numbers uses the same conventions you would use to add and subtract standard numbers. Four examples for addition and four examples for subtraction are shown below.

- Addition

$\begin{array}{r} 101111000 \\ 10111110 \\ + 10001101 \\ \hline 101001011 \end{array}$	$\begin{array}{r} 001011000 \\ 10101101 \\ + 00101100 \\ \hline 11011001 \end{array}$
--	---

$\begin{array}{r} 011111110 \\ 01111111 \\ + 00111111 \\ \hline 10111110 \end{array}$	$\begin{array}{r} 000000000 \\ 10101010 \\ + 01010101 \\ \hline 11111111 \end{array}$
---	---

- Subtraction

$\begin{array}{r} 001111100 \\ 11100101 \\ - 00101110 \\ \hline 10110111 \end{array}$	$\begin{array}{r} 001011000 \\ 10101101 \\ - 00101100 \\ \hline 11011001 \end{array}$
---	---

$\begin{array}{r} 010101010 \\ 10101010 \\ - 01010101 \\ \hline 01010101 \end{array}$	$\begin{array}{r} 000000000 \\ 11011101 \\ - 01001100 \\ \hline 10010001 \end{array}$
---	---

### 3 1/31/2020

#### 3.1 Lecture Notes

- Each group of bits has a size label associated with it.
  - The bit.
  - The nibble, which is 4 bits.
  - The byte, which is 8 bits.
  - The half-word, which is 16 bits.
  - The word, which is 32 bits.
  - The double word, which is 64 bits.

- Problem 3.1.1: Add b1011 to b101.

$$\begin{array}{r} \textcolor{blue}{1111} \\ 1011 \\ + \quad 101 \\ \hline \text{b10000} \end{array}$$

Answer: b10000

- Problem 3.1.2: Add b1011 to b1001.

$$\begin{array}{r} \textcolor{blue}{1\ 11} \\ 1011 \\ + \quad 1001 \\ \hline \text{b10100} \end{array}$$

Answer: b10100

- Problem 3.1.3: Subtract b111 from b1101.

$$\begin{array}{r} 1101 \\ - \quad 111 \\ \hline \text{b0110} \end{array}$$

Answer: b0110

- Problem 3.1.4: Subtract b1111 from b10101.

$$\begin{array}{r} 10101 \\ - \quad 1111 \\ \hline 110 \end{array} \quad \text{Broke it!}$$

Answer: No answer.

- Padding is the addition of zeroes in front of a number. For example, we could pad the number 429 out to 00429 without changing the value. This works in binary as well, as the value 1111 can be padded out to 001111 without its value being changed. This is important when considering hardware.
- Overflow is when the value is incorrect because it is larger than the allowed space.
- Errors also exist when subtracting binary. We can't borrow something that doesn't exist, as seen in problem 3.1.4. The result would be negative, meaning we cannot get the correct result using this method and number system.

- Problem 3.1.5: Prove adding b1101 and b111 results in a space constrained error.

$$\begin{array}{r}
 \textcolor{blue}{1111} \\
 1101 \\
 + \quad 111 \\
 \hline
 \textcolor{red}{b}10100
 \end{array}$$

Answer: Proven. The red numbers signify overflow.

- Problem 3.1.6: Prove subtracting b111 from b101 results in a borrow error.

$$\begin{array}{r}
 101 \\
 - \quad 111 \\
 \hline
 ?10
 \end{array}
 \quad \text{Broke it!}$$

Answer: Proven. Writing either “Error” or “Broke it” is acceptable.

## 3.2 Assigned Readingss

- In the signed-magnitude system, a number consists of a magnitude and a symbol indicating whether the number is positive or negative. We assume that the sign is positive if no sign is specified. There are two representations of zero, “+0” and “-0”, both with the same value.
- The signed-magnitude system is applied to binary numbers using an extra bit position used to represent the sign, called the sign bit. The most significant bit is typically used as the sign bit, with 0 representing a positive value and 1 representing a negative value.

$$\begin{array}{l}
 - 01010101_2 = +85_{10} \\
 - 01111111_2 = +127_{10} \\
 - 00000000_2 = +0_{10} \\
 - 11010101_2 = -85_{10} \\
 - 11111111_2 = -127_{10} \\
 - 10000000_2 = -0_{10}
 \end{array}$$

- The signed-magnitude system has an equal number of positive and negative integers. An  $n$ -bit signed-magnitude integer lies within the range  $-(2^{n-1} - 1)$  through  $+(2^{n-1} - 1)$ .
- While the signed-magnitude system negates a number by changing its sign, a complement number system negates a number by taking its complement as defined by the system. Taking a complement is more difficult than simply changing the sign, however two numbers in a complement system can be added or subtracted directly without the sign and magnitude checks that have to be done in the signed-magnitude system.
- In a two’s complement system, the complement of an  $n$ -bit number  $B$  is obtained by subtracting it from  $2^n$ . If  $B$  is between 1 and  $2^n - 1$ , thus subtracting produces another number between 1 and  $2^n - 1$ . If  $B$  is 0, the result of the subtraction is  $2^n$ . Because of this, there is only one representation of zero in a two’s complement system.

- An unnecessary subtraction operation can be avoided by rewriting  $2^n$  as  $(2^n - 1) + 1$  and  $2^n - B$  as  $((2^n - 1) - B) + 1$ . For example, for  $n = 8$ ,  $100000000_2$  equals  $11111111_2 + 1$ .
- If we define the complement of a bit  $b$  to be the opposite value of the bit, then  $(2^n - 1) - B$  is obtained by simply complementing the bits of  $B$ . Therefore, the two's complement of a number  $B$  is obtained by complementing the number of individual bits in  $B$  and adding 1. Again, using  $n = 8$  as an example, the two's complement of  $01110100$  is  $10001011 + 1$ , or  $10001100$ .
- In the two's complement system, the MSB serves as the sign bit. A number is negative *if and only if* its MSB is 1.
- In a ones' complement system, the complement of an  $n$ -bit number  $B$  is obtained by subtracting it from  $2^n - 1$ . This is accomplished by complementing the individual digits of  $B$  without adding 1 like in the two's complement system. The MSB acts as the sign, with 0 being positive and 1 being negative. This gives two representations of zero, a positive zero and a negative zero.
- While positive number representations are the same for ones' and two's complement, negative numbers differ by one.
- A weight of  $(2^{n-1} - 1)$ , rather than  $-2^{n-1}$  is given to the most significant bit when computing the decimal equivalent of a ones' complement number.
- The main advantages a ones' complement system has is its symmetry and ease of complementation, given its usage in early computer. However, due to the added design of ones' complement being more complicated and there being two representations of zero, two's complement is more widely used presently.
- In an excess-B representation, an  $m$ -bit string whose unsigned integer value is  $M$  ( $0 \leq M \leq 2^m$ ) represents the signed integer  $M - B$ , where  $B$  is the bias of the number system.
- For example, an excess- $2^{m-1}$  system represents any number  $X$  in the range between  $-2^{m-1}$  through  $+2^{m-1} - 1$ . The range of this representation is exactly the same as that of  $m$ -bit two's complement numbers. In fact, the range of the two systems are identical with the sole difference being that the sign bits are always opposite. Excess representation is mostly used in floating point number systems.
- Because ordinary addition is just an extension of counting, two's complement numbers can be added using ordinary binary addition, ignoring any carries beyond the MSB. This result will always be accurate so long as the range of the number system is not exceeded.
- If an addition operation produces a result that exceeds the range of the number system, overflow is said to have occurred. Addition of two numbers with different signs will never produce overflow, but addition with like signs can, as shown below.

$$\begin{array}{r} \phantom{+} \phantom{+} \phantom{+} \phantom{+} 1101 \\ + \phantom{+} \phantom{+} \phantom{+} 1010 \\ \hline 10111 = +7 \end{array} \qquad \begin{array}{r} \phantom{+} \phantom{+} \phantom{+} \phantom{+} 0101 \\ + \phantom{+} \phantom{+} \phantom{+} 0110 \\ \hline 1011 = -5 \end{array}$$

$$\begin{array}{r} \phantom{+} \phantom{+} \phantom{+} \phantom{+} 1000 \\ + \phantom{+} \phantom{+} \phantom{+} 1000 \\ \hline 10000 = +0 \end{array} \qquad \begin{array}{r} \phantom{+} \phantom{+} \phantom{+} \phantom{+} 0111 \\ + \phantom{+} \phantom{+} \phantom{+} 0111 \\ \hline 1110 = -2 \end{array}$$

- Overflow is easy to detect in addition: An addition overflows if the addends' signs are the same but the sum's sign differs.
- Subtraction of two's complement numbers also works as if they were normal unsigned binary numbers, and appropriate rules for detecting overflow may be formulated.
- Most subtraction circuits for two's complement numbers do not perform subtraction directly. Instead, they negate the subtrahend by taking its two's complement and then add it to the minuend using the normal rules of addition. This can be easily accomplished by performing a bit-by-bit complement of the subtrahend and then adding the complemented subtrahend to the minuend with an initial carry of 1 instead of 0. Examples are given below.

$$\begin{array}{r} \phantom{+} \phantom{+} \phantom{+} \phantom{+} \phantom{+} 1 \\ \phantom{+} \phantom{+} \phantom{+} \phantom{+} 0100 \\ - \phantom{+} \phantom{+} \phantom{+} 0011 \\ \hline \phantom{+} \phantom{+} \phantom{+} \phantom{+} 10001 \end{array} \qquad \begin{array}{r} \phantom{+} \phantom{+} \phantom{+} \phantom{+} \phantom{+} 1 \\ \phantom{+} \phantom{+} \phantom{+} \phantom{+} 0100 \\ + \phantom{+} \phantom{+} \phantom{+} 1100 \\ \hline \phantom{+} \phantom{+} \phantom{+} \phantom{+} 10001 \end{array}$$

$$\begin{array}{r} \phantom{+} \phantom{+} \phantom{+} \phantom{+} \phantom{+} 1 \\ \phantom{+} \phantom{+} \phantom{+} \phantom{+} 0011 \\ - \phantom{+} \phantom{+} \phantom{+} 1100 \\ \hline \phantom{+} \phantom{+} \phantom{+} \phantom{+} 0111 \end{array} \qquad \begin{array}{r} \phantom{+} \phantom{+} \phantom{+} \phantom{+} \phantom{+} 1 \\ \phantom{+} \phantom{+} \phantom{+} \phantom{+} 0011 \\ + \phantom{+} \phantom{+} \phantom{+} 0011 \\ \hline \phantom{+} \phantom{+} \phantom{+} \phantom{+} 0111 \end{array}$$

$$\begin{array}{r} \phantom{+} \phantom{+} \phantom{+} \phantom{+} \phantom{+} 1 \\ \phantom{+} \phantom{+} \phantom{+} \phantom{+} 0011 \\ - \phantom{+} \phantom{+} \phantom{+} 0100 \\ \hline \phantom{+} \phantom{+} \phantom{+} \phantom{+} 1111 \end{array} \qquad \begin{array}{r} \phantom{+} \phantom{+} \phantom{+} \phantom{+} \phantom{+} 1 \\ \phantom{+} \phantom{+} \phantom{+} \phantom{+} 0011 \\ + \phantom{+} \phantom{+} \phantom{+} 1011 \\ \hline \phantom{+} \phantom{+} \phantom{+} \phantom{+} 1111 \end{array}$$

$$\begin{array}{r} \phantom{+} \phantom{+} \phantom{+} \phantom{+} \phantom{+} 1 \\ \phantom{+} \phantom{+} \phantom{+} \phantom{+} 1101 \\ - \phantom{+} \phantom{+} \phantom{+} 1100 \\ \hline \phantom{+} \phantom{+} \phantom{+} \phantom{+} 10001 \end{array} \qquad \begin{array}{r} \phantom{+} \phantom{+} \phantom{+} \phantom{+} \phantom{+} 1 \\ \phantom{+} \phantom{+} \phantom{+} \phantom{+} 1101 \\ + \phantom{+} \phantom{+} \phantom{+} 0011 \\ \hline \phantom{+} \phantom{+} \phantom{+} \phantom{+} 10001 \end{array}$$

Overflow in subtraction is detected by examining the signs of the minuend and the complemented subtrahend, using the same rule as addition.

- In unsigned addition, the carry or borrow in the most significant bit position indicates an out-of-range result. In signed two's complement addition the overflow condition defined earlier indicates an out-of-range result. The carry from the most significant bit position is irrelevant in signed addition, in the sense that the overflow can occur independently whether or not carry occurs.

## 4 2/3/2020

### 4.1 Lecture Slides

- A leading bit to the left of the number is used to tell us the sign. This special bit is called the sign bit. A sign bit of 0 represents a positive number and a sign bit of 1 represents a negative number.
- In the Sign and Magnitude system, or SAM, the sign bit is just stuck to the front of a number. If padding is necessary, you pad first and then add the sign bit. This gives us a range of  $-(2^{n-1} - 1)$  to  $(2^{n-1} - 1)$ . SAM results in two zeroes, which makes math tricky.
- Problem 4.1.1: Represent -32 in 8-bit SAM.

32/2	16	R0
16/2	8	R0
8/2	4	R0
4/2	2	R0
2/2	1	R0
1/2	0	R1

In 6-bit, 32 is b100000. We pad this out to 7-bit by adding a zero, giving us b0100000. Lastly, we add the sign bit for a negative number, giving us b10100000.

Answer: b10100000

- Problem 4.1.2: Represent -13 in 8-bit SAM.

13/2	6	R1
6/2	3	R0
3/2	1	R1
1/2	0	R1

In 4-bit, -13 is b1101. We pad this out to 7-bit by adding three zeroes, giving us b0001101. Lastly, we add the sign bit for a negative number, giving us b10001101.

Answer: b10001101

- There are two complement systems: ones' complement and two's complement. The systems involve swapping  $1 \rightarrow 0$  and  $0 \rightarrow 1$ . The complement of the complement is the original number.
- Recall that the number of bits matter. If you only add a spot for the sign bit, the number may need to be extended in the event that you need more bits. If the number is positive, just pad some zeroes. If the number is negative, extend with some 1's, or alternatively pad before taking the complement.
- In ones' complement, you add a spot for the sign and then flip the number. Ones' complement has the same double zero issue as SAM.

- Problem 4.1.3: Represent 32 with 8-bit ones' complement.

– Step 1: Get the positive binary value.

32/2	16	R0
16/2	8	R0
8/2	4	R0
4/2	2	R0
2/2	1	R0
1/2	0	R1

32 = b100000.

– Step 2: Add the positive sign bit.

b0100000.

– Step 3: Sign extend.

b00100000.

Answer: b00100000

- Problem 4.1.4: Represent -32 with 8-bit ones' complement.

– Steps 1-3: Get the unsigned value (see 4.1.3 for the work).

b00100000.

– Step 4: Flip the number.

b11011111.

Answer: b11011111

- Problem 4.1.5: Represent 13 with 8-bit ones' complement.

– Step 1: Get the positive binary value.

13/2	6	R1
6/2	3	R0
3/2	1	R1
1/2	0	R1

13 = b1101.

– Step 2: Add the positive sign bit.

b01101.

– Step 3: Sign extend.

b00001101.

Answer: b00001101

- Problem 4.1.6: Represent -13 with 8 bit ones' complement.

– Steps 1-3: Get the unsigned value (see 4.1.5 for the work).

b00001101.

– Step 4: Flip the number.

b11110010.

Answer: b11110010

- Two's complement is functionally identical to ones' complement except you add one at the end, removing the second zero.

- Problem 4.1.7: Represent 32 with 8-bit two's complement.  
A positive two's complement number is the same as a positive ones' complement number. Using the work from 4.1.3, we found positive 32 to be b00100000 in ones' complement.

Answer: b00100000

- Problem 4.1.8: Represent -32 with 8-bit two's complement.
  - Steps 1-3: Get the unsigned value (see 4.1.3 for the work).  
b00100000.
  - Step 4: Flip the number.  
b11011111.
  - Step 5: Add one.

$$\begin{array}{r} \text{b11011111} \\ + \quad \quad \quad 1 \\ \hline \text{b11100000} \end{array}$$

Answer: b11100000

- Problem 4.1.9: Represent 13 with 8-bit two's complement.  
A positive two's complement number is the same as a positive ones' complement number. Using the work from 4.1.5, we found positive 13 to be b00001101 in ones' complement.

Answer: b00001101

- Problem 4.1.10: Represent -13 with 8-bit two's complement.
  - Steps 1-3: Get the unsigned value (see 4.1.5 for the work).  
b00001101.
  - Step 4: Flip the number.  
b11110010.
  - Step 5: Add one.

$$\begin{array}{r} \text{b11110010} \\ + \quad \quad \quad 1 \\ \hline \text{b11110011} \end{array}$$

Answer: b11110011

- With the Excess-B system, the main idea is to move zero to a point that isn't zero, called the bias point. This allows you to have both positive and negative values. The Excess-B system is primarily used in specific systems design and IEEE-754 representation.
- There is the potential for an incorrect answer when performing binary math, specifically when there is a carry in to the sign bit but not out (or vice versa). Unless explicitly stated, use two's complement when using signed numbers.
- Addition between two positive or two negative numbers may be incorrect answer that is out of range. If there is a carry into and out of the sign bit, the answer is correct. Otherwise, the answer is false. In the event that this occurs, you can declare the answer to have "overflowed" and leave it there or redo the math with a larger amount of bits.



- Problem 4.1.11: Add -7 to 7 in 4-bit.

- Since this is signed math, we use two's complement.
- 7 in two's complement is b0111.
- -7 in two's complement is b1001.
- Next, we need to add the numbers.

$$\begin{array}{r}
 \textcolor{blue}{1111} \\
 \text{b0111} \\
 + \text{b1001} \\
 \hline
 \textcolor{red}{1}0000
 \end{array}$$

- We throw away the excess bit, or the red “1”, leaving us with b0000.

Answer: b0000

- Problem 4.1.12: Add 7 to -8 in 4-bit.

Answer: Can't be done in 4-bit, 8=1000.

- With subtraction, we exploit the idea that  $x = a - b = a + (-b)$ .
- Problem 4.1.13: Subtract 8 from -12 in 8-bit.

- Since this is signed math, we use two's complement.
- -12 in two's complement, after being padded appropriately and flipped, is b11110100.
- -8 in two's complement, after being padded appropriately and flipped, is b11111000.
- Next, we need to subtract the numbers.

$$\begin{array}{r}
 \textcolor{blue}{1111} \\
 \text{b11111000} \\
 + \text{b11110100} \\
 \hline
 \textcolor{red}{1}11101100
 \end{array}$$

- We throw away the excess bit, or the red “1”, leaving us with b11101100.

Answer: b11101100

## 4.2 Assigned Readings

- Multiplying binary numbers functions very similarly to normal multiplication. Forming shifted multiplicands is trivial in binary multiplication since the only multiplier digits are 0 and 1. An example of this multiplication is shown below.

	1011	multiplicand
×	1101	multiplier
	1011	
	0000	
	1011	shifted multiplicands
	1011	
	10001111	product

- In a digital system, it is more convenient to add each multiplicand as it is created to a partial product. Such a method looks like this:

	1011	multiplicand
×	1101	multiplier
<hr/>		
	0000	partial product
	1011	shifted multiplicand
<hr/>		
	01011	partial product
	0000↓	shifted multiplicand
<hr/>		
	001011	partial product
	1011↓↓	shifted multiplicand
<hr/>		
	0110111	partial product
	1011↓↓↓	shifted multiplicand
<hr/>		
	10001111	product

- In general, when we multiply an  $n$ -bit number by an  $m$ -bit number, the resulting product requires at most  $n + m$  bits to express. The shift-and-add algorithm requires  $m$  partial products and additions to obtain the result.
- Multiplication of signed numbers can be accomplished using unsigned multiplication. In other words, perform an unsigned multiplication of the magnitudes and make the product positive if the operands have the same sign but negative if they have different signs.
- We perform two's complement multiplication by using a sequence of two's complement additions of shifted multiplicands. Only the last step is changed, where the shifted multiplicand corresponding to the MSB of the multiplier is negated before being added to the partial product. An example of two's complement multiplication is shown below.

	1011	multiplicand
×	1101	multiplier
<hr/>		
	00000	partial product
	11011	shifted multiplicand
<hr/>		
	111011	partial product
	00000↓	shifted multiplicand
<hr/>		
	1111011	partial product
	11011↓↓	shifted multiplicand
<hr/>		
	11100111	partial product
	00101↓↓↓	shifted and negated multiplicand
<hr/>		
	00001111	product

## 5 2/5/2020

### 5.1 Lecture Slides

- Binary multiplication functions very similarly to decimal multiplication.
- When multiplying with signed numbers, you should do all multiplication in the positive form. Remember to add the sign bit in the front. If necessary, correct the product to a negative value. After multiplying, pad to the nearest power of 2.
- Problem 5.1.1: Multiply b0101 and b0011.

$$\begin{array}{r}
 0101 \\
 \times 0011 \\
 \hline
 0101 \\
 0101\downarrow \\
 0000\downarrow\downarrow \\
 0000\downarrow\downarrow\downarrow \\
 \hline
 b00001111
 \end{array}$$

Answer: Answer: b00001111

- Problem 5.1.2: Multiply b0111 and b0011.

$$\begin{array}{r}
 0111 \\
 \times 0011 \\
 \hline
 0011 \\
 0011\downarrow \\
 0011\downarrow\downarrow \\
 0000\downarrow\downarrow\downarrow \\
 \hline
 b00010101
 \end{array}$$

Answer: b00010101

- Problem 5.1.3: Multiply b0101 and b1011.

$$\begin{array}{r}
 0100 \\
 + 1 \\
 \hline
 0101 \\
 \times 0101 \\
 \hline
 0101 \\
 0000\downarrow \\
 0101\downarrow\downarrow \\
 0000\downarrow\downarrow\downarrow \\
 \hline
 b0011001 \\
 1100110 \\
 + 1 \\
 \hline
 b11100111
 \end{array}$$

Answer: b11100111

- Problem 5.1.4: Multiply b1101 and b0011.

$$\begin{array}{r}
 0010 \\
 + \quad 1 \\
 \hline
 0011 \\
 \\
 \begin{array}{r}
 0011 \\
 \times 0011 \\
 \hline
 0011 \\
 0011\downarrow \\
 0000\downarrow\downarrow \\
 0000\downarrow\downarrow\downarrow \\
 \hline
 00001001 \\
 11110110 \\
 + \quad 1 \\
 \hline
 b11110111
 \end{array}
 \end{array}$$

Answer: b11110111

- Problem 5.1.5: Multiply b1101 and b1011.

$$\begin{array}{r}
 0010 \\
 + \quad 1 \\
 \hline
 0011 \\
 0100 \\
 + \quad 1 \\
 \hline
 0101 \\
 \\
 \begin{array}{r}
 0011 \\
 \times 0101 \\
 \hline
 0011 \\
 0000\downarrow \\
 0011\downarrow\downarrow \\
 0000\downarrow\downarrow\downarrow \\
 \hline
 b00001111
 \end{array}
 \end{array}$$

Answer: b00001111

- Problem 5.1.6: Multiply b1011 and b1011.

$$\begin{array}{r}
 0100 \\
 + \quad 1 \\
 \hline
 0101 \\
 \\
 \begin{array}{r}
 0101 \\
 \times 0101 \\
 \hline
 0101 \\
 0000\downarrow \\
 0101\downarrow\downarrow \\
 0000\downarrow\downarrow\downarrow \\
 \hline
 b00011001
 \end{array}
 \end{array}$$

Answer: b00011001

## 5.2 Assigned Readings

- A set of  $n$ -bit strings in which different bit strings represent different numbers or other things is called a code. A particular combination of  $n$  1-bit values is called a code word.
- There may not necessarily be a mathematical relationship between a particular code word and what it is supposed to represent. Furthermore, a code using  $n$ -bit strings doesn't need to contain  $2^n$  valid code words.
- Listed below are some common ways the ten decimal digits are represented.

<b>Decimal digit</b>	<b>BCD (8421)</b>	<b>2421</b>	<b>Excess-3</b>	<b>Biquinary</b>	<b>1-out-of-10</b>
0	0000	0000	0011	0100001	1000000000
1	0001	0001	0100	0100010	0100000000
2	0010	0010	0101	0100100	0010000000
3	0011	0011	0110	0101000	0001000000
4	0100	0100	0111	0110000	0000100000
5	0101	1011	1000	1000001	0000010000
6	0110	1100	1001	1000010	0000001000
7	0111	1101	1010	1000100	0000000100
8	1000	1110	1011	1001000	0000000010
9	1001	1111	1100	1010000	0000000001
Unused code words					
	1010	0101	0000	0000000	0000000000
	1011	0110	0001	0000001	0000000011
	1100	0111	0010	0000010	0000000101
	1101	1000	1101	0000011	0000000110
	1110	1001	1110	0000101	0000000111
	1111	1010	1111	...	...

- The most natural of these is the binary-coded decimal system, or BCD. This system encodes the digits 0 through 9 by their 4-bit unsigned binary representations. Because of this, conversions between BCD and decimal representations are trivial.
- Some computer programs place two BCD digits into one 8-bit byte in the packed-BCD representation. In this system, one byte may represent the values from 0 to 99 versus 0 to 255 for a normal, unsigned 8-bit binary number. BCD numbers for any desired value can be obtained by using one byte for every two digits.
- Similarly to binary numbers, there are many possible representations of negative BCD values. Signed BCD numbers have one extra digit position for the sign, and both the signed-magnitude and 10's complement representations are used in BCD arithmetic. In signed-magnitude BCD, the encoding of the sign bit string is arbitrary, while in 10's complement, 0000 indicates a positive value and 1001 indicates a negative value.
- Addition of BCD digits function similarly to adding 4-bit unsigned numbers, with the sole difference being that a correction must be made if the result exceeds 1001, in which case the result is corrected by adding six.

- Binary-coded decimal is a weighted code because each decimal digit can be obtained from its code word by assigning a fixed weight to each code-word bit. The weights for the BCD bits are 8, 4, 2, and 1, which leads to BCD sometimes being called 8421 code.
- Another set of weights leads to 2421 code, which has the advantage of being self-complementing, in that the code word for the 9s' complement of any digit can be obtained by complementing the individual bits of the digit's code word.
- Another self-complementing code is excess-3 code, which although not weighted does have a mathematical relationship with BCD code. The code word for each decimal digit in excess-3 is the corresponding BCD code word plus 0011.
- Decimal codes can have more than four digits, shown with the biquinary code system, using seven. The first two bits of the code word indicate whether the number is within the range of 0-4 and 5-9, and the remaining five indicate which of those five numbers is being represented. This system is used in an abacus.
- An advantage to using more than the minimum number of bits is an error-detecting property. In biquinary code, for example, if any single bit is changed to the opposite value the resulting code word immediately does not represent a decimal digit.
- 1-out-of-10 code uses ten bits instead of four.
- Gray code is a numbering system where only one bit changes between adjacent numbers. The code words for gray code are listed below.

Decimal Number	Binary Code	Gray Code
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

- A string of digits doesn't need to represent a number. In fact, most of the information processed by computers is nonnumeric. The most common type of nonnumeric data is text, or strings of characters from some character set. Each character is represented in the computer by a bit string according to an established convention, such as ASCII.
- ASCII represents each character with a 7-bit string, yielding a total of 128 different characters. The ASCII standard contains uppercase, lowercase, numerals, punctuation, and even various nonprinting control characters.

## 6 2/7/2020

### 6.1 Lecture Notes

- A code is a pattern that represents some idea, concept, or a piece of information. With numbers, for example, you have a code that represents a quantity.
- Gray code is an example of a code and is very useful for sensors. Gray code introduces us to the idea of a code word, which is simply a way of using 0s and 1s to represent information. BCD, ASCII, Unicode, and Gray Code are all codeword representations.
- In the Binary Coded Decimal, or BCD, representation, the numbers 0 through 9 are represented. Addition isn't too messy either, with a carry only needing to be forced when adding numbers totaling to over 9.
- Problem 6.1.1: Convert 97 to BCD.  
9 in BCD: b1001  
7 in BCD: b0111  

Answer: b10010111
- Problem 6.1.2: Convert 36 to BCD.  
3 in BCD: b0011  
6 in BCD: b0110  

Answer: b00110110
- When adding BCD, you must consider the possibility of errors resulting in invalid code. This can be fixed with a correction or a forced carry.
- Problem 6.1.3: Add 14 and 27 in BCD.  
$$\begin{array}{r} 14 = 00010100 \\ + \quad 27 = 00100011 \\ \hline 00111011 \\ + \qquad \qquad 0110 \\ \hline b01000001 \end{array}$$

Because the original number was 0011 1011, which is 3 and 11. The 11 isn't a single digit, so a carry must be forced.

Answer: b01000001
- Problem 6.1.4: Add 9 and 13 in BCD.  
$$\begin{array}{r} 14 = \quad 1001 \\ + \quad 27 = 00010011 \\ \hline 00011100 \\ + \qquad \qquad 110 \\ \hline b00100010 \end{array}$$

Answer: b00100010
- The American Standard Code for Information Exchange, or ASCII, is a 7-bit representation with 128 possible characters. Extending ASCII allows for the usage of the 8th bit, opening up 255 possible characters.

- The last important number representation is fixed point. When we deal with something that isn't a whole number (which happens very frequently), we are essentially adding negative powers to the right of the radix point.
- Fixed point representation uses the following equation, with the highlighted portion being the radix point:  $B = b_{n-1}b_{n-2}...b_1b_0.b_{-1}b_{-2}...b_{-k}$
- Listed below are various representations of 2 using fixed point.

- $2^{-1} = 0.5$
- $2^{-2} = 0.25$
- $2^{-3} = 0.125$
- $2^{-4} = 0.0625$
- $2^{-5} = 0.03125$

- Problem 6.1.5: Convert 6.3 into floating point.

6 = 0110

.3 · .2 = 0.6	0
.6 · .2 = 1.2	1
.2 · .2 = 0.4	0
.4 · .2 = 0.8	0
.8 · .2 = 1.6	1
.6 · .2 = 1.2	1
.2 · .2 = 0.4	0
.4 · .2 = 0.8	0
.8 · .2 = 1.6	1
.6 · .2 = 1.2	1

Answer: b110.0100110011

- There are two conditions that allow you to stop doing the above process. The first is if the number to the right of the radix point is 0, because 0 multiplied by anything is 0. 0.500000 is just .5, mathematically speaking. If this never happens, you go to ten places, which is what happened above.

## 6.2 Assigned Readings

- Boolean algebra is an algebraic system designed to “give expression ... to the fundamental laws of reasoning in the symbolic language of a Calculus.”
- Eventually, Boolean algebra was applied to analyze and describe the behavior of circuits built from relays. In a system called switching algebra, the condition of a relay contact, whether it be open or closed, is represented by a variable X that can have one of two possible values: 0 or 1.
- In switching algebra, we use a symbolic variable such as the aforementioned X to represent the condition of a logic signal. A logic signal can represent a variety of conditions depending on the technology. For each technology, one condition is 0 and another 1.



- Most logic circuits use the positive-logic convention, using 0 to represent a LOW voltage and 1 to represent a high voltage. The negative-logic convention is the inverse of this, but is rarely used.
- The axioms (or postulates) of a mathematical system are a minimal set of basic definitions that we assume to be true. Using axioms, all other information about a particular system can be derived.
- All axioms are stated as a pair. This is a characteristic of axioms in switching algebra called “duality.”
- An inverter is a logic circuit whose output signal level is the opposite, or complement, of its input signal level. We use a prime tick (') to denote an inverter function.
- This prime tick is an algebraic operator, and a statement such as  $X'$  is an expression.
- A 2-input AND gate is a circuit whose output is 1 if both of its inputs are 1. The function of a 2-input AND gate is sometimes called logical multiplication and is symbolized algebraically by a multiplication dot ( $\cdot$ ). Some mathematicians and logicians use the wedge ( $\wedge$ ) to denote logical multiplication.
- A 2-input OR gate is a circuit whose output is 1 if either of its inputs are 1. The function of a 2-input AND gate is sometimes called logical addition and is symbolized algebraically by a plus sign ( $+$ ). Some mathematicians and logicians use the vee ( $\vee$ ) to denote logical addition.
- By convention, logical multiplication has a higher precedence than logical addition.
- Switching algebra theorems are statements known to always be true that allow us to manipulate algebraic expressions for simpler analysis. For example, the theorem  $X + 0 = X$  allows us to substitute every occurrence of  $X + 0$  in an expression with just  $X$ . A list of theorems involving one variable are shown below.

(T1)	$X + 0 = X$	(T1D)	$X \cdot 1 = X$	(Identities)
(T2)	$X + 1 = 1$	(T2D)	$X \cdot 0 = 0$	(Null elements)
(T3)	$X + X = X$	(T3D)	$X \cdot X = X$	(Idempotency)
(T4)	$(X')' = X$			(Involution)
(T5)	$X + X' = 1$	(T5D)	$X \cdot X' = 0$	(Complements)

- Most theorems can be easily proven by using a technique called perfect induction.
- Switching algebra theorems with two or three variables are listed below.

(T6)	$X + Y = Y + X$	(T6D)	$X \cdot Y = Y \cdot X$	(Commutativity)
(T7)	$(X + Y) + Z = X + (Y + Z)$	(T7D)	$(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)$	(Associativity)
(T8)	$X \cdot Y + X \cdot Z = X \cdot (Y + Z)$	(T8D)	$(X + Y) \cdot (X + Z) = X + Y \cdot Z$	(Distributivity)
(T9)	$X + X \cdot Y = X$	(T9D)	$X \cdot (X + Y) = X$	(Covering)
(T10)	$X \cdot Y + X \cdot Y' = X$	(T10D)	$(X + Y) \cdot (X + Y') = X$	(Combining)
(T11)	$X \cdot Y + X' \cdot Z + Y \cdot Z = X \cdot Y + X' \cdot Z$			(Consensus)
(T11')	$(X + Y) \cdot (X' + Z) \cdot (Y + Z) = (X + Y) \cdot (X' + Z)$			

- Several important theorems for an arbitrary number of variables are listed below. Most of these theorems are proved using a two-step method called finite induction, where one first proves the theorem true for  $n = 2$  and then for  $n = i$ , concluding as a result it is true for  $n = i + 1$ .

(T12)	$X + X + \dots + X = X$	(Generalized idempotency)
(T12D)	$X \cdot X \cdot \dots \cdot X = X$	
(T13)	$(X_1 \cdot X_2 \cdot \dots \cdot X_n)' = X_1' + X_2' + \dots + X_n'$	(DeMorgan's theorems)
(T13D)	$(X_1 + X_2 + \dots + X_n)' = X_1' \cdot X_2' \cdot \dots \cdot X_n'$	
(T14)	$[F(X_1, X_2, \dots, X_n, \cdot)]' = F(X_1', X_2', \dots, X_n', \cdot, +)$	(Generalized DeMorgan's theorem)
(T15)	$F(X_1, X_2, \dots, X_n) = X_1 \cdot F(1, X_2, \dots, X_n) + X_1' \cdot F(0, X_2, \dots, X_n)$	(Shannon's expansion theorems)
(T15D)	$F(X_1, X_2, \dots, X_n) = [X_1 + F(0, X_2, \dots, X_n)] \cdot [X_1' + F(1, X_2, \dots, X_n)]$	

- It was previously stated that all axioms in switching algebra are in pairs. The dual of each axiom is obtained from the base axiom by simply swapping 0 and 1 and if present  $\cdot$  and  $+$ . As a result of this, we can state the following metatheorem (a metatheorem is simply a theorem about theorems).
  - Principle of Duality: Any theorem or identity in switching algebra is also true if 0 and 1 are swapped and  $\cdot$  and  $+$  are swapped throughout.
- Duality is important because it doubles the usefulness of almost everything about switching algebra and the manipulation of switching functions.
- The most basic representation of a logic function is the truth table, a brute-force representation that lists the output of the circuit with every possible input combination. Pictured below is a truth table.

Row	X	Y	Z	F
0	0	0	0	$F(0, 0, 0)$
1	0	0	1	$F(0, 0, 1)$
2	0	1	0	$F(0, 1, 0)$
3	0	1	1	$F(0, 1, 1)$
4	1	0	0	$F(1, 0, 0)$
5	1	0	1	$F(1, 0, 1)$
6	1	1	0	$F(1, 1, 0)$
7	1	1	1	$F(1, 1, 1)$

- The information obtained in a truth table can also be conveyed algebraically. To do so, a few terms must be defined.
  - A literal is a variable or the complement of a variable. Examples:  $X$ ,  $Y$ ,  $X'$ ,  $Y'$ .
  - A product term is a single literal or a logical product of two or more literals. Examples:  $Z'$ ,  $W \cdot X \cdot Y$ ,  $X \cdot Y' \cdot Z$ ,  $W' \cdot Y' \cdot Z$ .
  - A sum-of-product expression is a logical sum of product terms. Examples:  $Z' + W \cdot X \cdot Y + X \cdot Y' \cdot Z + W' \cdot Y' \cdot Z$ .
  - A sum term is a single literal or a logical sum of two or more literals. Examples:  $Z'$ ,  $W + X + Y$ ,  $X + Y' + Z$ ,  $W' + Y' + Z$ .

- A normal term is a product or sum term in which no variable appears more than once. A non-normal term can always be simplified to a constant or a normal term using one of theorems T3, T3', T5, or T5'. Examples of non-normal terms:  $W \cdot X \cdot X \cdot Y'$ ,  $W + W + X' + Y$ ,  $X \cdot X' \cdot Y$ . Examples of normal terms:  $W \cdot X \cdot Y'$ ,  $W + X' + Y$ .
- An n-variable minterm is a normal product term with n literals. There are  $2^n$  such product terms. Some examples of 4-variable minterms:  $W' \cdot X' \cdot Y' \cdot Z'$ ,  $W \cdot X \cdot Y' \cdot Z$ ,  $W' \cdot X' \cdot Y \cdot Z'$
- An n-variable maxterm is a normal sum term with n literals. There are  $2^n$  such sum terms. Examples of 4-variable maxterms:  $W' + X' + Y' + Z'$ ,  $W + X' + Y' + Z$ ,  $W' + X' + Y + Z'$
- There is a close relationship between the truth table and minterms and maxterms. A minterm defined as a product term that is 1 in exactly one row of a truth table. Similarly, a maxterm defined as a sum term that is 0 in exactly one row of a truth table.
- An n-variable minterm can be represented by an n-bit integer, the minterm number. Syntactically, the name minterm i is used to denote the minterm corresponding to row i of the truth table. In minterm i, a particular variable appears complemented if the corresponding bit in the binary representation of i is 0, otherwise it is uncomplemented. A maxterm i is the opposite, with a variable being complemented if the corresponding binary bit i is 1.
- The canonical sum of a logic function is a sum of the minterms corresponding to the truth-table rows for which the function produces a 1 output.
- The canonical product of a logic function is a product of the maxterms corresponding to input combinations for which the function produces a 0 output.

## 7 2/10/2020

### 7.1 Lecture Notes

- Boolean algebra is an approach towards closed set arithmetic, or a reasoning in a symbolic language, and was developed by George Boole in 1854. Such an approach was originally simplified to only using 0s and 1s, but it was eventually expanded upon by other mathematicians.
- Claude Shannon applied the idea of Boolean algebra, allowing it to be used from circuits, which are built from relays (alternatively known as switches).
- Axioms and postulates are the base set of mathematical ideas known to be true.
- There are two forms of logic in Boolean algebra: Positive logic (where one is true and zero is false) and negative logic (where zero is true and one is false).
- All complete equations and functions are composed of three parts: An equals sign, an expression or variable on the left side, and an expression or variable on the right side.
- An inverter is the idea of complementing or inverting a variables value in an expression. Inverting is traditionally represented by a ' '.
- Logical addition is a Boolean algebra concept and is denoted with the + sign. It is fundamentally the "OR" concept.
- Logical multiplication is another Boolean algebra concept and is denoted with the \* sign. It is fundamentally the "AND" concept.
- In Boolean algebra, operators have precedence, with some operators taking priority over others. The order is parenthesis  $\leftarrow$  inverters  $\leftarrow$  multiplication  $\leftarrow$  addition.
- Problem 7.1.1: Simplify  $Y = (1 * 1 + (0 * 1 * 1 + 0 + 1) * 0 + 1) * (0 + 1)$ . Red simplifies to 1, blue to 0, and green to 1. Thus, the we can first simplify the equation to  $Y = (1 + 0 + 1) * 1$ . Purple simplifies to 1, so we can further simplify the equation to  $Y = 1 * 1$ . This, the equation simplified is equivalent to  $Y = 1$ .  

Answer:  $Y = 1$
- Problem 7.1.2: Simplify  $T = (1 * 1 * 1 * 1 * 0 * 1 + 0 * 0) * 1 + (1 + 1 * 1 * 1)$ . Red simplifies to 0, blue to 0, and green to 1. Thus, we can first simplify the equation to  $T = (0 + 0) * 1 + (1 + 1)$ . Purple simplifies to 1, and orange simplifies to 1. Thus, we can yet further simplify the equation to  $T = 0 * 1 + 1$ . Brown simplifies to 0, finally simplifying the equation  $T = 0 + 1$ , leaving the answer as  $T = 1$ .  

Answer:  $T = 1$
- Commutativity is the idea that the order of operations do not matter.

- Associativity is the idea that the order in which segments of an equation are executed do not matter.
- Covering refers to some terms being excluded when a single term “covers” all possible cases.
- Redundancy is a simplification trick when simplifying algebraic expressions.
- General idempotency is simply the idea of applying impotence repeatedly.

$$\begin{aligned}
 - & x = x * x = x * x * x * x * x \\
 - & x = x + x = x + x + x + x + x
 \end{aligned}$$

- Problem 7.1.3: Simplify  $T = Y' * F + X * (X' + Y)$ .
  - Distributive property:  $T = Y' * F + X * X' + X * Y$
  - Identity property:  $T = Y' * F + 0 + X * Y$
  - Simplify:  $T = Y' * F + X * Y$

Answer:  $T = Y' * F + X * Y$

- Problem 7.1.4: Simplify  $Y = X * X * T + T * T + X$ .
  - Idempotency:  $Y = X * T + T + X$
  - Distributive property:  $X(T + 1) + T$
  - Simplification:  $X(1) + T$
  - Simplification:  $X + T$

Answer:  $Y = X + T$

- Augustus DeMorgan came up with a theorem that describes factoring and distributing inverted functions. His theorems are listed below.
  - $(x * y)' = x' + y'$
  - $(x + y)' = x' * y'$
  - $x + x' * y = x + y$
  - $x * (x' + y) = x * y$

- Problem 7.1.5: Simplify  $T = X * (Y + X)' + Y$ .
  - DeMorgan’s Law:  $T = X * (Y' * X) + Y$
  - Distributive Law:  $X * Y + X * X' + Y$
  - Simplification:  $T = X * Y' + Y$

Answer:  $T = Y + X$

- The main idea behind Shannon’s expansion theorem is configuring an expression to contain more variables than the hardware would normally allow.

## 8 2/12/2020

### 8.1 Lecture Notes

- Problem 8.1.1: Simplify  $Y = X'T + (X + T)'$ .

Use DeMorgan's.

$$y = \bar{x}T + [\bar{x} \cdot \bar{T}]$$

$$y = \bar{x}T + \bar{x}\bar{T}$$

$$y = \bar{x}(T + \bar{T}) \text{ (note that } T + \bar{T} = 1)$$

$$y = \bar{x} \cdot 1$$

$$y = \bar{x}$$

Answer:  $y = \bar{x}$

- Sometimes, we want to work with the complement because we can simplify things, or maybe due to power requirements. In these cases, we just NOT the entire function.

- Problem 8.1.2: Find the simplified complement of  $Y = XT + WT' + W'X$ .

$$\bar{y} = \text{complement} = [XT + WT' + W'X]'$$

$$(\bar{X} + \bar{T})(\bar{W} + \bar{T})(\bar{W} + \bar{X})$$

$$(\bar{X} + \bar{T})(\bar{W} + T)(W + \bar{X})$$

$$(\bar{X}\bar{W} + \bar{X}T + \bar{T}\bar{W} + \bar{T}T)(W + \bar{X})$$

$$\bar{W}\bar{W}\bar{X} + \bar{X}W\bar{X} + \bar{X}TW + \bar{X}T\bar{X} + \bar{T}\bar{W}W + \bar{T}W\bar{X}$$

$$\bar{X}\bar{W} + \bar{X}TW + \bar{X}T + \bar{T}W\bar{X}$$

$$\bar{X}\bar{W}(1 + T) + \bar{X}WT + \bar{X}T$$

$$\bar{X}\bar{W} + \bar{X}T(W + 1)$$

$$\bar{y} = \bar{X}\bar{W} + \bar{X}T$$

Answer:  $\bar{y} = \bar{X}\bar{W} + \bar{X}T$

- Problem 8.1.3: Find the complement of  $T = AB' + B(A' + C)$ .

$$\bar{T} = [AB' + B(A' + C)]'$$

$$(\bar{A} + \bar{B})(\bar{B} + (\bar{A} \cdot \bar{C}))$$

$$(\bar{A} + B)(\bar{B} + (A\bar{C}))$$

$$\bar{A}\bar{B} + A(\bar{B}C) + B\bar{B} + BAC$$

$$\bar{T} = \bar{A}\bar{B} + BAC$$

Answer:  $\bar{T} = \bar{A}\bar{B} + BAC$

- Below is a list of various representations of logic functions.
  - A literal is a variable or complement in the function.
  - A product term is a single literal that is the product of two or more single literals.
  - A sum term is a single literal composed of single literals.
  - A normal term is a logic function in which no variable appears more than once.
- A truth table is a visual tabular representation of a logical function. It lists all of the inputs and outputs for a function and the order of the bits is sorted in binary counting order. If there are  $n$  inputs, you need  $2^n$  lines in your truth table.

- Problem 8.1.4: Create the truth table for  $Y = X + W'T + WT'$ .

$T$	$W$	$X$	$Y$	$WT$	$WT'$
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	1
0	1	1	1	0	1
1	0	0	1	1	0
1	0	1	1	1	0
1	1	0	0	0	0
1	1	1	1	0	0

- Problem 8.1.5: Create the truth table for  $F = A'B + ABC'$ .

$A$	$B$	$C$	$\overline{A}B$	$AB\overline{C}$	$F$
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	1	0	1
0	1	1	1	0	1
1	0	0	0	0	0
1	0	1	0	0	0
1	1	0	0	1	1
1	1	1	0	0	0

- Duality is how we describe the idea that each of the axioms and theorems have two parts. To find the dual of the function, change...

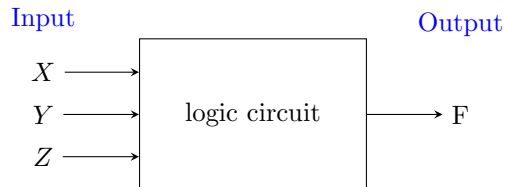
- $\cdot \rightarrow +$
- $+$   $\rightarrow \cdot$
- $0 \rightarrow 1$
- $1 \rightarrow 0$

Swapping 0 and 1 is NOT the same as complementing.

- A self dual is when you can take a dual and the resulting function generates the same outcomes as the original. Not all functions that have a dual can produce a self dual (most can't, in fact).
- A minterm is the function output for an input combination of 1, and is also a normal product term.
- A maxterm is the function output for an input combination of 0, and is also a normal sum term.

## 8.2 Assigned Readings


- A logic circuit can be represented with a minimum amount of detail by representing it as a “black box” with a certain number of inputs and outputs. Below is an example of a logic circuit.



- A logic circuit whose outputs depend only on its current inputs is called a combinational circuit. The operation of such a circuit is fully described by a truth table that lists all combinations of input values and the corresponding output values. Below is an example of a truth table.

$X$	$Y$	$Z$	$F$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1


- A circuit with memory whose outputs depend on the current input in addition to the sequence of past inputs are called sequential circuits. Its behavior may be described by a state table, which specifies its output and next state as functions of its current state and input.
- The most basic digital devices are called gates. Generally speaking, a gate has one or more inputs and produces an output that is a function of the current input values.
- Just three basic logic functions (AND, OR, and NOT) can be used to build any combinational logic circuit. The graphical symbols, along with their corresponding truth tables, are shown below.



X AND Y

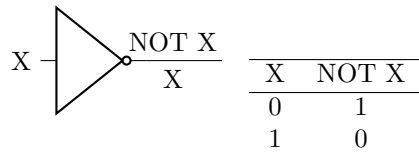
$X \cdot Y$

X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1



X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1





The gates' functions are easily described using words.

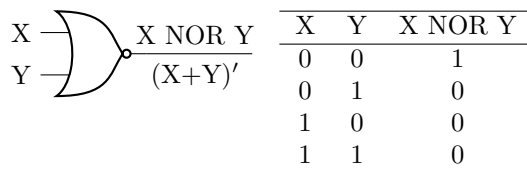
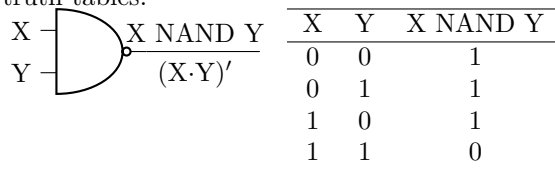
- An AND gate produces a 1 output if and only if all of its inputs are 1.
- An OR gate produces a 1 output if and only if one or more of its inputs is 1.
- A NOT gate is usually called an inverter and produces an output value the opposite of its input value.

- Take notice of the below circle from the inverter symbol's output.

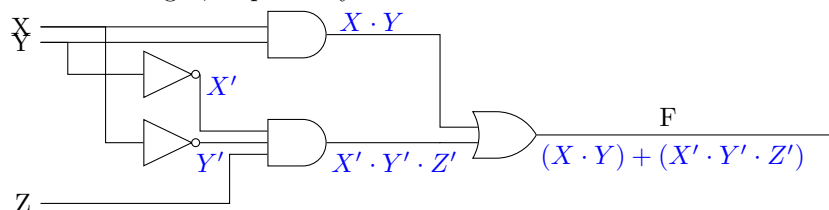


This is called an inversion bubble and is used in this and other gate symbols to denote “inverting” behavior.

- Two additional logic functions are obtained by inverting the outputs of AND and OR gates. Shown below are their graphical symbols and their truth tables.



- A logic diagram shows the graphical symbols for multiple logic gates and other elements in a logic circuit, in addition to their interconnections (called wires). The output of each element may connect to inputs of one or more other elements. Signals in a logic diagram traditionally flow left to right, and inputs and outputs of the overall circuit are drawn on the left and right, respectively.



- Besides voltage and current, logic circuits are also useful in representing time. A timing function graphically shows how a circuit may respond to a time-varying pattern of various input signals. Time is graphed horizontally and logic values are graphed vertically.
- By obtaining a formal definition of a combinational circuit's logic function, we can analyze it. This description allows us to perform a variety of operations, such as determining the logic circuit's behavior, manipulating an algebraic or equivalent graphical description to suggest different circuit elements for the logic function, and more.
- Given a logic diagram for a combinational circuit, there are several ways to obtain a formal description of the circuit's function, the most basic of which is the truth table.
- Using only the basic axioms of switching algebra, we can easily obtain the truth table of an  $n$ -input circuit by working our way through all  $2^n$  input combinations. For each input combination, we determine each of the gate outputs produced by the input and propagate information from the circuit inputs to outputs.
- The number of input combinations for a logic circuit grows exponentially in relation to the number of inputs, so an exhaustive approach such as the one described above can become tiring. Because of this, for many analysis problems it is better to use an algebraic approach whose complexity is more linearly proportional to the size of the circuit.
- This new method is simple: we build up a parenthesized logic expression corresponding to the logic operators and the structure of the circuit. We start at the inputs and propagate expressions as we move toward the output. You can either simplify these expressions using the axioms of switching algebra as you go or all at once at the end.

## 9 2/14/2020

### 9.1 Lecture Slides

- Problem 9.1.1: Take the dual of  $Y = X + W'T + WT'$ .  
We must swap  $\cdot \rightarrow +$  and  $+ \rightarrow \cdot$ . There are no zeros or ones so we don't need to worry about swapping those.  

Answer: Dual of  $Y = X \cdot (W' + T) \cdot (W + T')$
- Problem 9.1.2: Take the dual of  $F = A'B + ABC'$ .  
We must swap  $\cdot \rightarrow +$  and  $+ \rightarrow \cdot$ . There are no zeros or ones so we don't need to worry about swapping those.  

Answer: Dual of  $F = (A' + B) \cdot (A + B + C')$
- With canonical representation, every term in our equation contains every input.
- In SOP, or Sum of Products, each term is the product of the literals and they are all summed together. In a truth table, if the input for a literal for that minterm is a 1, then the literal is itself. Conversely, if the input for a literal for that minterm is a 0, then the literal is a complement of itself. Since SOP is a sum, we represent it with  $\sum_{\text{variables in the function}}$ .
- POS, or Product of Sums, is when each term is summed together and then is taken as the product. In a truth table, if the input for a literal for that maxterm is a 1, then the literal is a complement of itself. Conversely, if the input for a literal for that maxterm is a 0, then the literal is the complement of itself. We use the capital greek letter pi, or  $\prod$ , to represent POS.
- Because functions can get long, we use shorthand to make writing them more manageable. When writing a function, just write which minterm or maxterm numbers to include. Furthermore, use  $\sum$  for SOP (also known as sigma notation) and  $\prod$  for POS (also known as pi notation).
- Problem 9.1.3: Create the truth table for  $F = \sum(2, 4, 6, 7)$ .

$x$	$y$	$z$	$F$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

★ Truth tables are only complete when every row has an output.

- Problem 9.1.4: Create the truth table for  $F = \sum(0, 5, 6)$ .

$x$	$y$	$z$	$F$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

- Problem 9.1.5: Create the canonical function for  $\sum(2, 4, 6, 7)$ .  
Using a truth table, find the values of 2, 4, 6, and 7. 2 is 010, 4 is 100, 6 is 110, and 7 is 111. Next, convert these into the canonical function using  $x$ ,  $y$ , and  $z$ .

Answer:  $F = \overline{x}y\overline{z} + x\overline{y}z + xy\overline{z} + xyz$

- Problem 9.1.6: Create the canonical function for  $\sum(0, 5, 6)$ .  
Using a truth table, find the values of 0, 5, and 6. 0 is 000, 5 is 101, and 6 is 110. Next, convert these into the canonical function using  $x$ ,  $y$ , and  $z$ .

Answer:  $F = \overline{x}y\overline{z} + x\overline{y} + xy\overline{z}$

- Problem 9.1.7: Create the truth table for  $F = \prod(2, 4, 6, 7)$ .

$x$	$y$	$z$	$F$
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

- Problem 9.1.8: Create the truth table for  $F = \prod(0, 5, 6)$ .

$x$	$y$	$z$	$F$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

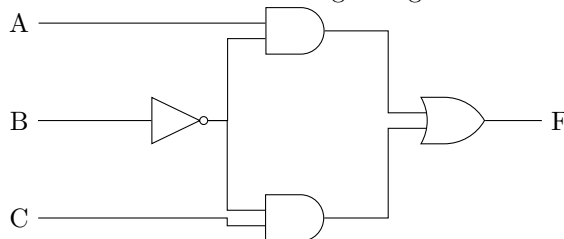
- Problem 9.1.9: Create the canonical function for  $F = \prod(2, 4, 6, 7)$ .  
Using a truth table, find the values of 2, 4, 6, and 7. 2 is 010, 4 is 100, 6 is 110, and 7 is 111. Next, convert these into the canonical function using  $x$ ,  $y$ , and  $z$ .

Answer:  $F = (x + y + \overline{z})(\overline{x} + y + z)(\overline{x} + \overline{y} + \overline{z})(\overline{x} + \overline{y} + z)$

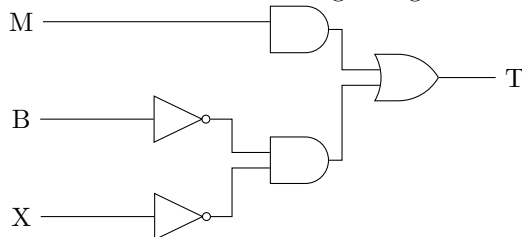
- Problem 9.1.10: Create the canonical function for  $F = \prod(0, 5, 6)$ . Using a truth table, find the values of 0, 5, and 6. 0 is 000, 5 is 101, and 6 is 110. Next, convert these into the canonical function using  $x$ ,  $y$ , and  $z$ .

Answer:  $F = (x + y + z)(\bar{x} + y + z)(\bar{x} + \bar{y} + z)$

- On-set and off-set is a system used to describe the shorthand list of items. Minterms in shorthand have a list of the rows with 1 as the output, making it on-set. The complementary idea also holds true, with off-set being for maxterms.
- Logic gates are used to physically represent functions and can be drawn. AND, OR, and NOT are said to create a complete logic set.
- Logic diagrams are drawings that help with design and preparation for implementation. For now, we will only use computational logic.
- Multi level and two level are forms of canonical representation systems. Two level is SOP or POS form. NOTs don't count. Unless explicitly told otherwise, always simplify and create a SOP style.
- Problem 9.1.11: Draw the logic diagram for  $F = AB' + B'C$ .



- Problem 9.1.12: Draw the logic diagram for  $T = MX + B'X'$ .



## 9.2 Assigned Readings

- A logic circuit description is occasionally just a list of input combinations for when a signal would be on or off. For example, the description of a 4-bit prime-number detector might be “Given a 4-bit input combination  $N = N_3N_2N_1N_0$ , produce a 1 output for  $N = 1, 2, 3, 5, 7, 11, 13$ .”

- A logic function described in this way can be designed directly from a given canonical sum or produce expression. For the above prime-number detector, we would have...

$$\begin{aligned}
 F &= \sum_{N_3, N_2, N_1, N_0} (1, 2, 3, 5, 7, 11, 13) \\
 &= N'_3 \cdot N'_2 \cdot N'_1 \cdot N_0 + N'_3 \cdot N'_2 \cdot N_1 \cdot N'_0 + N'_3 \cdot N'_2 \cdot N_1 \cdot N_0 + N'_3 \cdot N_2 \cdot N'_1 \cdot N_0 \\
 &\quad + N'_3 \cdot N_2 \cdot N_1 \cdot N'_0 + N'_3 \cdot N_2 \cdot N_1 \cdot N_0 + N_3 \cdot N'_2 \cdot N'_1 \cdot N_0
 \end{aligned}$$

- More often than this, however, we describe a logic function using the natural-language connections “and”, “or”, and “not.” For example, we might describe an alarm circuit by saying

“The ALARM output is 1 if the PANIC input is 1, or if the ENABLE input is 1, the EXITING input is 0, and the house is not secure; the house is secure if the WINDOW, DOOR, and GARAGE inputs are all 1.”

Such a description can be directly translated into algebraic expressions.

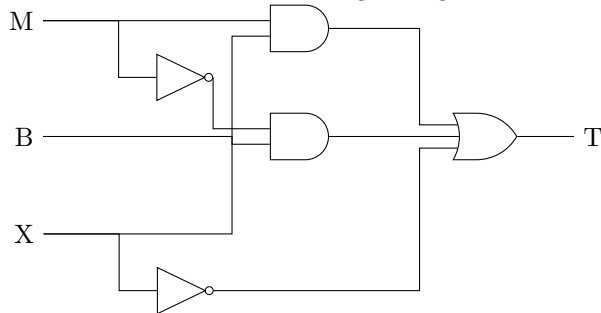
$$\begin{aligned}
 \text{ALARM} &= \text{PANIC} + \text{ENABLE} \cdot \text{EXITING}' \cdot \text{SECURE}' \\
 \text{SECURE} &= \text{WINDOW} \cdot \text{DOOR} \cdot \text{GARAGE} \\
 \text{ALARM} &= \text{PANIC} + \text{ENABLE} \cdot \text{EXITING}' \cdot (\text{WINDOW} \cdot \text{DOOR} \cdot \text{GARAGE})
 \end{aligned}$$

- A circuit realizes an expression if its output function equals that expression. If a circuit realizes an expression, it is called a realization of the function. In place of realization, the term “implementation” is sometimes used. Recognize both - they are both used in practice.
- Once we have any expression, we can do more than just build a circuit from the expression. We can, for example, manipulate the expression to get different circuits. The aforementioned ALARM expression can be multiplied out to get a sum-of-products circuit, as an example. Alternatively, if the number of variables isn't very large, we can create a truth table for the expression.
- In general, when we are designing a logic function for an application, it is easier to describe the logic function in words using logical connectives than it is to write a complete truth table (especially if the number of variables is large).
- Sometimes however we start with imprecise word descriptions of logic functions, such as the sentence “The ERROR output should be 1 if the GEARUP, GEARDOWN, and GEARCHECK inputs are inconsistent.” In these cases, a truth-table approach is more optimal because it allows us to determine the output required for every input. Using a logic expression in this case might make it difficult to notice “corner cases” and handle them appropriately.

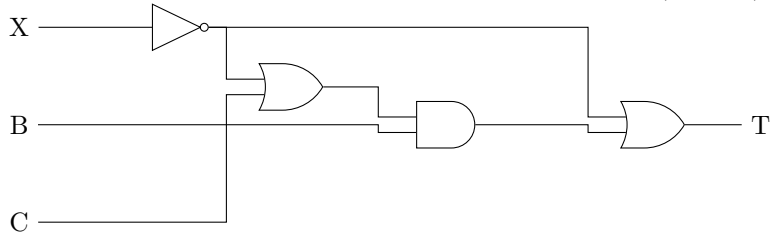
10 2/17/2020

## 10.1 Lecture Slides

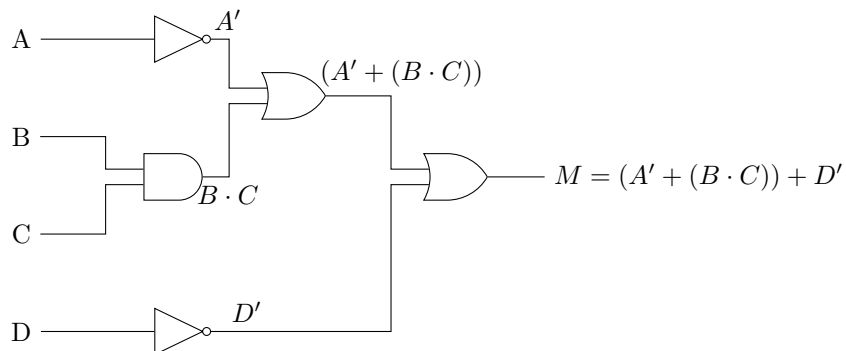
- Problem 10.1.1: Draw the logic diagram for  $T = MX + BM' + X'$ .



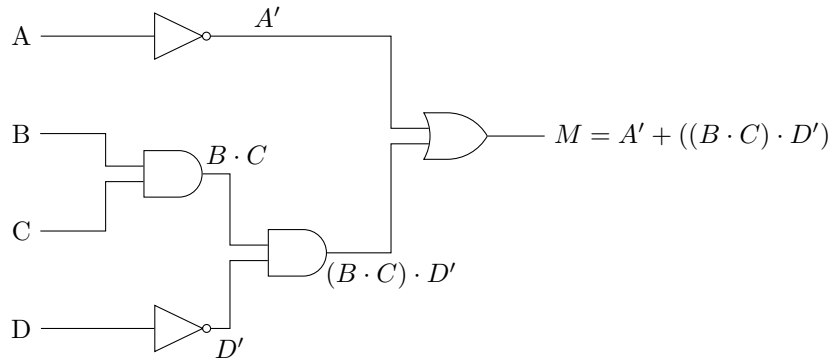
- Problem 10.1.2: Draw the logic diagram for  $T = X' + B(X' + C)$ .



- You can always follow the path of connections in a diagram to see the behavior that it is implementing. This is done by labeling the function after each gate. See the below problems for examples.
- Problem 10.1.3: Develop the equation for the logic diagram. Do not simplify the solution.



- Problem 10.1.4: Develop the equation for the logic diagram. Do not simplify the solution.



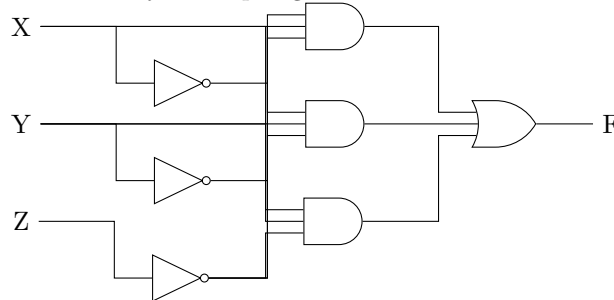
- Combinational logic synthesis is the process that specifies the required function and creates the details for implementation. Combinational logic design is the broader overview of the entire process, which includes logic synthesis.



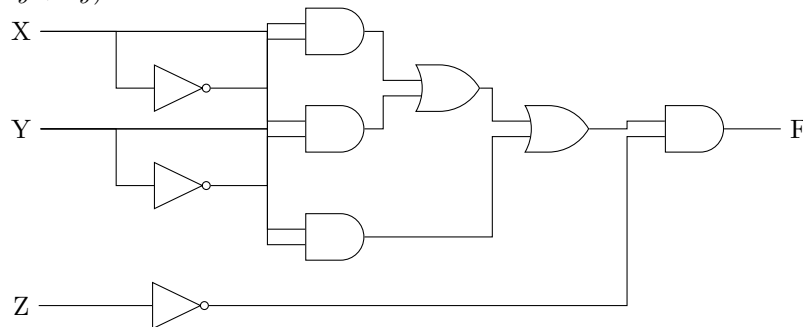
## 11 2/19/2020

### 11.1 Lecture Slides

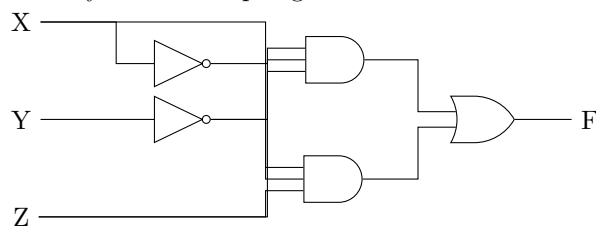
- A manipulator is a fancy way of saying that we can make an equivalent function with different arrangements.
- Problem 11.1.1: Draw the canonical logic diagram for  $F = \sum_{x,y,z}(2,4,6)$  but with only two input gates and NOTs.



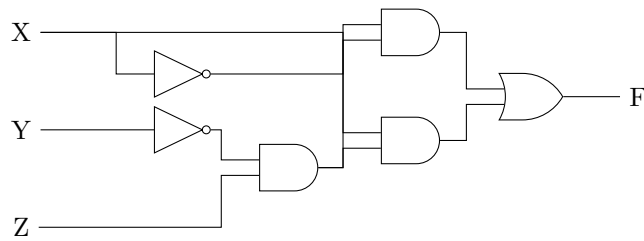
This is a two level, three input gate  $F = \bar{x}y\bar{z} + x\bar{y}\bar{z} + xy\bar{z}$ . Using the associative property, we can create a multi-level two input gate  $F = \bar{z}(\bar{x}y + x\bar{y} + xy)$ .



- Problem 11.1.2: Draw the canonical logic diagram for  $F = \sum_{x,y,z}(1,5)$  but only with two input gates and NOTs.



This above logic diagram is for  $F = \bar{x}y\bar{z} + x\bar{y}\bar{z}$  and is incorrect. The below correct logic diagram takes advantage of the distributive property.  $F = \bar{x}(\bar{y}z) + x(\bar{y}z)$ .

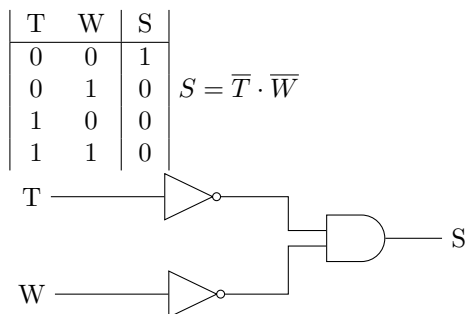


- When designing a system, there are a few important questions to ask.
  - What are the inputs?
  - What are the outputs?
  - Are there any constraints or relationships between the inputs and outputs?

After asking these questions, you can build the design.

- Problem 11.1.3: Design a system to turn on and off a sprinkler in the yard.  
 Inputs: Time(day<sup>1</sup> or night<sup>0</sup>) → T  
 Weather (rain<sup>1</sup> or no rain<sup>0</sup>) → W

Sprinkler: On<sup>1</sup> or off<sup>0</sup> = output → S



- Problem 11.1.4: Design a system to determine if you are allowed to watch Netflix.  
 Inputs: Homework (have<sup>1</sup> or have not<sup>0</sup>) → H  
 Class (in class<sup>1</sup> or not in class<sup>0</sup>) → C  
 Output: Watch Netflix (yes<sup>1</sup> or no<sup>0</sup>) → N

H	C	N
0	0	1
0	1	0
1	0	0
1	1	0

$$N = \bar{H} \cdot \bar{C}$$

$$\bar{H} \cdot \bar{C} = N$$

$$N = \bar{H} \cdot \bar{C} + H\bar{C} = \bar{C}$$

## 12 2/21/2020

### 12.1 Lecture Slides

- When solving logic systems, we usually want to take the lazier approach, so we look for methods that are simpler with less spots for error and a lower price to implement.
- Minimization aims to reduce the “cost”, which is done by reducing the number and size of gates.
- There are three key ways to reduce cost.
  1. Minimize the number of first-level gates.
  2. Minimize the number of inputs on each first level gate.
  3. Minimize the number of inputs on each second level gate.
- How are you handle how many inputs to use for a gate? Do as you’re told, do as you’re equipped to handle, and do as you want (in that order).
- Problem 12.1.1: Find the simplified POS  $F = \prod_{x,y,z}(1, 2, 5, 6)$ .  
 $F = (x + y + \bar{z})(x + \bar{y} + z)(\bar{x} + y + \bar{z})(\bar{x} + \bar{y} + z)$   
 $F = (\cancel{x}x^x + x\bar{y} + xz + yx + \cancel{y}\bar{y}^0 + yz + \bar{z}x + \bar{z}x + \cancel{\bar{z}}\bar{z}^0)^0$   
 $(\cancel{x}x^x + x\bar{y} + y\bar{x} + \cancel{y}\bar{y}^0 + yz + \bar{z}x + \bar{z}y + \cancel{\bar{z}}\bar{z}^0)$   
 $F = x\bar{x} + x\bar{y} + x\bar{x}z + xy\bar{z} + xyz + x\bar{z}x + x\bar{z}y$   
 This takes too long! For POS functions, simplifying them algebraically is far too complicated. We need an alternative.
- Instead of solving algebraically, we use Karnaugh maps. Some Karnaugh maps are shown below.

		$X$	
		0	1
$Y$	0	0	2
	1	1	3

		$X$			
		00	01	11	10
$Z$	0	0	2	6	4
	1	1	3	7	5

$Y$

		WX		W	
		00	01	11	10
YZ	00	0	4	12	8
	01	1	5	13	9
	11	3	7	15	11
	10	2	6	14	10

X

Z

- There are a few things to note about K-Maps. First, working with more than four variables generally isn't worth it, at which computer aided design begins to be used. Second, the order of the numbers is in gray code.
- Why gray code? Take the two terms  $A'BC + ABC$ . If we apply the distributive property we get  $BC(A' + A)$ . Then, we can apply the complement theorem to get  $(A' + A) = 1$ , which reduces to just  $BC$ . This can be interpreted from the K-Map through gray code, forcing terms that can cancel to be adjacent.
- Problem 12.1.2: How does the below truth table map to a K-Map?

x	y	F
0	0	A
0	1	B
1	0	C
1	1	D

F:

		x	
		0	1
y	0	0	2
	1	1	3

- Problem 12.1.3: How does the below truth table map to a K-Map?

x	y	z	F
0	0	0	A
0	0	1	B
0	1	0	C
0	1	1	D
1	0	0	E
1	0	1	F
1	1	0	G
1	1	1	H

$F:$

$xy$		$x$			
		00	01	11	10
$z$	0	000	010	110	100
	1	001	011	111	101

$y$

- To use the K-Map for reduction, some vocabulary terms need to get defined.
  - Implicants (or a covers) are power of 2 circles/rectangles that go around neighboring 1's in SOP and 0's in POS. The prime implicant, or PI, is the largest cover possible.
  - Distinguished 1's are an easy way of checking if there are PIs. A distinguished 1 is a 1 on the map covered by only one implicant. If that implicant isn't used, the function produced is not the intended output.
  - The essential prime implicant, or EPI, is an implicant that must be used in order to obtain the correct output of a function.
- For now, we will focus on SOP. This is a "recipe" for making an SOP K-map.
  - Place all of the minterms on the K-Map.
  - Draw all of the PIs, starting with the largest size possible and then working your way down. All PIs must be powers of 2.
  - Determine the distinguished 1's. Using these distinguished 1's, find the essential prime implicants.
  - Begin creating the function using these essential prime implicants.
  - Look at the map and check if there are any 1's not covered by prime implicants.
- The rules for "NOTing" are identical to those of a truth table. When dealing with product terms, NOT only if the input for a variable is 0. When dealing with sum terms, NOT only if the input for a variable is 1.
- Two rules must be followed when reducing/combining terms. First, the terms must be edge adjacent. Second, you must group by powers of 2, starting with the largest powers first.

- Problem 12.1.4: Find the simplified SOP for  $F = \sum_{x,y,z}(0, 4, 5, 6, 7)$  algebraically.

$$F = \overline{x}\overline{y}\overline{z} + x\overline{y}\overline{z} + x\overline{y}z + xy\overline{z} + xyz$$

$$F = \overline{y}\overline{z}(\overline{x} + x^1) + x\overline{y}z + xy(\overline{z} + z^1)$$

$$F = \overline{y}\overline{z} + x\overline{y}z + xy$$

$$\boxed{\text{Answer: } F = \overline{y}\overline{z} + x\overline{y}z + xy}$$

- Problem 12.1.5: Find the simplified SOP for  $F = \sum_{x,y,z}(0, 4, 5, 6, 7)$  using a K-Map.

$F:$

		$x$			
		00	01	11	10
$z$	0	0 1	2	6 1	4 1
	1	1	3	7 1	5 1
		$y$			

$$\overline{xy}z + x\overline{y}z$$

$$\overline{yz}(\overline{x} + x^1)$$

$$\overline{yz}$$

110  $xy\overline{z}$   
 100  $\overline{x}\overline{y}z$   
 111  $xyz$   
 101  $\overline{x}\overline{y}z$

If it's the same, it stays, but if it changes it goes.

Distinguished 1's: 0, 6, 7, 5

Essential prime implicant(s):  $\overline{yz}, x$

Answer:  $F = \overline{yz} + x$

- Problem 12.1.6: Find the simplified SOP for  $F = \sum_{x,y,z}(0, 1, 4, 5, 7)$  using a K-Map.

$F:$

		$x$			
		00	01	11	10
$z$	0	0 1	2	6	4 1
	1	1 1	3	7 1	5 1
		$y$			

Distinguished 1's: 0, 1, 4, 7

Essential prime implicant(s):  $\overline{y}, xz$

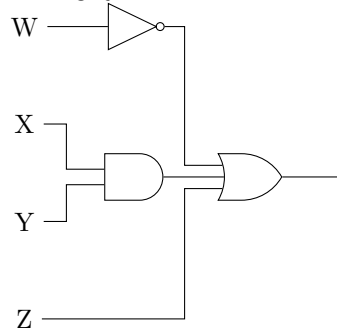
Answer:  $F = \overline{y} + xz$

## 12.2 Assigned Readings

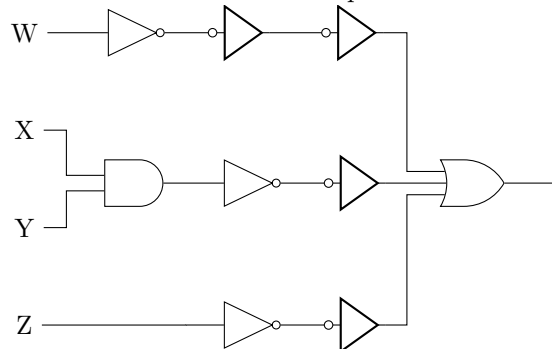
- We have previously described AND, OR, and NOT gates. We also want to use NAND and NOR gates, as these are faster than AND and OR gates in most technologies.

- NAND and NOR connectives aren't very logically intuitive, however. For example, you wouldn't say "I won't date you if you're not clean or not wealthy and also you're not smart or not friendly." You would instead say "I'll date you if you're clean and wealth or if you're smart and friendly." To produce a "natural" logic expression we need ways to translate this into other forms for a more efficient implementation.
- We can translate any logic expression into an equivalent sum-of-product expression simply by multiplying it out.
- We can insert a pair of inverters between each AND-gate output and the corresponding OR-gate input in a two-level AND-OR circuit. These inverters, per T4 (see page 21) have no effect on the output function of a circuit.
- However, if these inverters are absorbed into the AND and OR gates, we wind up with an AND-NOT gate on the first level and a NOT-OR gate on the second. These are two symbols for the same type of gate: The NAND gate. A two level AND-OR gate can be converted to a two level NAND-NAND gate simply by substituting gates. Below is a realization of a sum-of-products circuit.

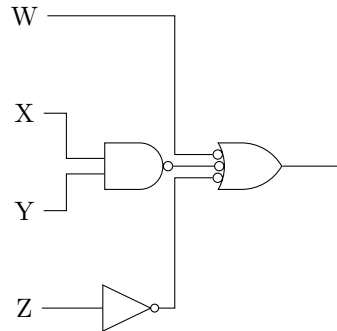
a. AND-OR



b. AND-OR with extra inverter pairs

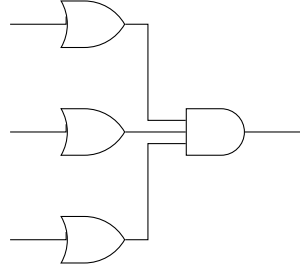


c. NAND-NAND

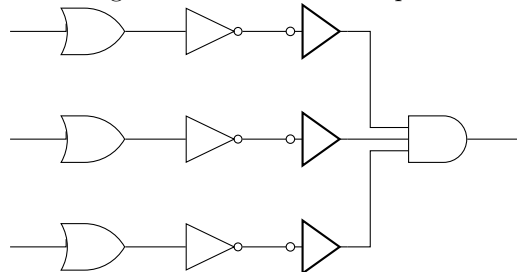


- The dual of this idea also holds true, in that any product-of-sums expression can be realized as an OR-AND circuit or as a NOR-NOR circuit. Below is a realization of a product-of-sums circuit.

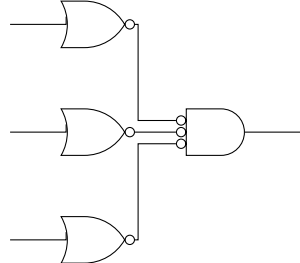
a. OR-AND



b. OR-AND gate with extra inverter pairs



c. NOR-NOR





- Often, it is uneconomical or inefficient to realize a logic circuit directly from the first logic expression you think of. Canonical sum and product expressions are particularly expensive because the number of possible minterms and maxterms grows exponentially with the number of variables. We need to minimize a combinational circuit by reducing the number and size of gates that are needed to build it.
- There are three minimization methods used to reduce the cost of a two-level AND-OR, OR-OR, NAND-NAND, or NOR-NOR circuit.
  1. Minimize the number of first-level gates.
  2. Minimize the number of inputs on each first level gate.
  3. Minimize the number of inputs on the second level gate, which is actually just a side effect of the first reduction.
- A two-gate realization that has the minimum possible number of first level gates and gate inputs is called a minimal sum or minimal product. Some functions have multiple minimal sums or products.
- Most minimization methods are generalizations of T10 and T10D (see page 22). That is, if two product or sum terms differ only in the complementing or not of one variable, we can combine them into a single term with one less variable.
- Karnaugh maps were originally used to create graphical representations of logic functions, allowing minimization opportunities to be identified by a simple recognizable visual pattern. The key feature of a Karnaugh map (hereafter referred to as a K-Map) is its cell layout, in which “adjacent pairs of cells corresponding to a pair of minterms that differ in only one variable which is uncomplemented in one cell and complemented in another”. Below are of various K-Maps.

(a) 2-variable

		$X$	
		0	1
$Y$	0	0	2
	1	1	3

(b) 3-variable

		$X$			
		00	01	11	10
$Z$	0	0	2	6	4
	1	1	3	7	5

$Y$

(c) 4-variable

		WX		W	
		00	01	11	10
Y	Z				
	00	0	4	12	8
	01	1	5	13	9
	11	3	7	15	11
	10	2	6	14	10

$X$

$Z$

- Observe the below K-Map.

$F$ :

		AB		A	
		00	01	11	10
C	D				
	00	0	4	12	8
	01	1 1	5 1	13 1	9
	11	3 1	7 1	15	11 1
	10	2 1	6	14	10

$B$

$D$

Take note of how adjacent 1 cells were grouped to correspond to their prime implicant, or “product terms that cover only input combinations for which the function has a 1 output, and that would cover at least one input combination with a 0 output if any variable were removed.”

## 13 2/24/2020

### 13.1 Lecture Slides

- Problem 13.1.1: Simplify  $F = \sum_{A,B,C,D}(0, 6, 9, 11, 14)$ .

$F$ :

		$AB$		$A$	
		00	01	11	10
$CD$	00	0 1	4	12	8
	01	1	5	13	9 1
	11	3	7	15	11 1
	10	2	6 1	14 1	10
		$B$		$D$	

Distinguished 1's: 0,6,9,11,14

Essential prime implicant(s):  $\overline{A}BC\overline{D}$ ,  $A\overline{B}D$ ,  $BC\overline{D}$

Answer:  $F = \overline{A}BC\overline{D} + A\overline{B}D + BC\overline{D}$

- Problem 13.1.2: Simplify  $F = \sum_{A,B,C,D}(0, 2, 8, 10, 15)$ .

$F$ :

		$AB$		$A$	
		00	01	11	10
$CD$	00	0 1	4	12	8 1
	01	1	5	13	9
	11	3	7	15 1	11
	10	2 1	6	14	10 1
		$B$		$D$	

Distinguished 1's: 0,2,8,10,15

Essential prime implicant(s):  $ABCD$ ,  $\overline{B}\overline{D}$

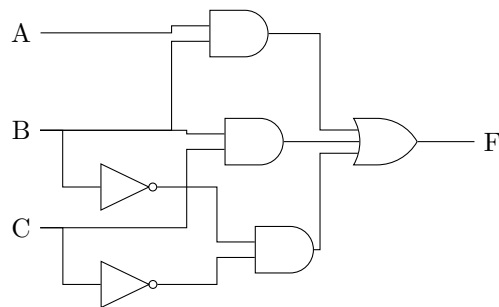
Answer:  $F = ABCD + \overline{B}\overline{D}$

- Problem 13.1.3: Find the shorthand POS and simplified SOP and draw the two level logic diagram for  $F = \prod_{A,B,C}(1,2,5)$ .

A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

$F$ :

		$A$			
		00	01	11	10
$C$	0	0 1	2	6 1	4 1
	1	1	3 1	7 1	5
		$B$			



Distinguished 1's: 0, 3

Essential prime implicant(s):  $\overline{BC}, BC$

Answer:  $F = \overline{BC} + BC + AC$

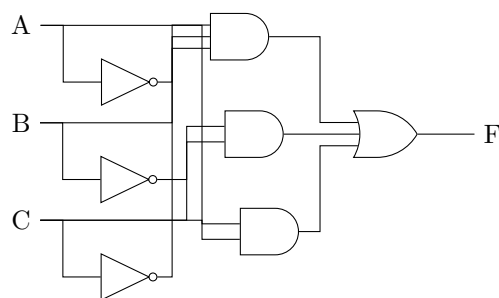
- Problem 13.1.4: Find the shorthand POS and simplified SOP and draw the two level logic diagram for  $F = \prod_{A,B,C}(0,3,4,6)$ .

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

$F$ :

$AB$		$A$			
		00	01	11	10
$C$	0	0	2 1	6	4
	1	1 1	3	7 1	5 1

$B$



Distinguished 1's: 1, 2, 7

Essential prime implicant(s):  $\overline{B}C, \overline{A}B\overline{C}, AC$

Answer:  $F = \overline{B}C + \overline{A}B\overline{C} + AC$

- Problem 13.1.5: Find the simplified SOP from  $F = \prod_{A,B,C,D}(0, 1, 2, 3, 4, 6, 8, 10, 11, 14)$  and draw the 2 level logic diagram.

$F$ :

$AB$		$A$			
		00	01	11	10
$CD$	00	0	4	12 1	8
	01	1	5 1	13 1	9 1
$C$	11	3	7 1	15 1	11
	10	2	6	14	10

$B$

$D$

Distinguished 1's: 5, 7, 12, 15, 9

Essential prime implicant(s):  $AB\overline{C}, A\overline{C}D, BD$

## 14 2/26/2020

### 14.1 Lecture Slides

- Equipment costs money, so we need to have a method to describe the cost of a function. This is done by counting the number of gates. Due to standard practice, initial NOT gates are not counted.
- Problem 14.1.1: Draw the K-Map for  $F = \sum_{A,B,C,D}(0, 1, 3, 4, 7, 13, 15)$  and state the cost.

$F$ :

		$AB$		$A$	
		00	01	11	10
$CD$	00	0 1	4 1	12	8
	01	1 1	5	13 1	9
	11	3 1	7 1	15 1	11
	10	2	6	14	10
		$B$		$D$	

Distinguished 1's: 4, 13

Essential prime implicant(s):  $\overline{ACD}, ABD$

Answer:  $F = \overline{ACD} + ABD + \overline{ABD} + BCD \therefore$  Cost is 5

- Problem 14.1.2: Draw the K-Map for  $F = \sum_{A,B,C,D}(1, 2, 7, 9, 10, 11, 15)$  and state the cost.

$F$ :

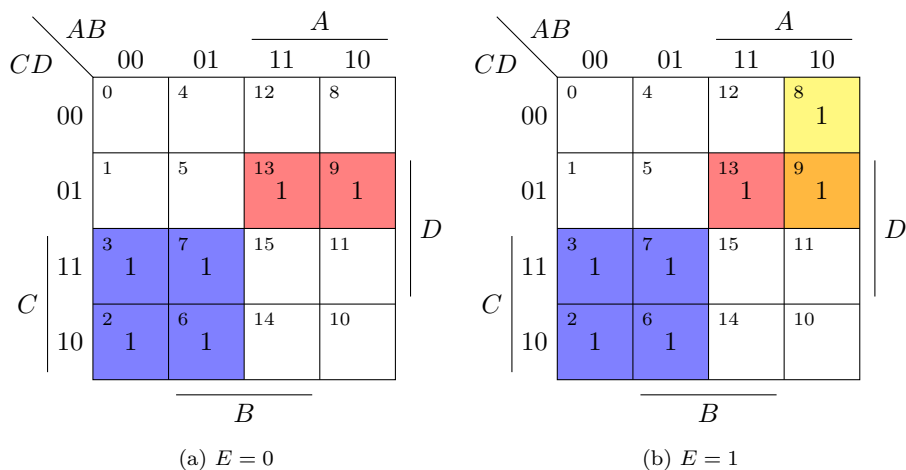
		$AB$		$A$	
		00	01	11	10
$CD$	00	0	4	12	8
	01	1 1	5	13	9 1
	11	3	7 1	15 1	11 1
	10	2 1	6	14	10 1
		$B$		$D$	

Distinguished 1's: 1, 2, 7

Essential prime implicant(s):  $\overline{BCD}, \overline{BCD}, BCD$

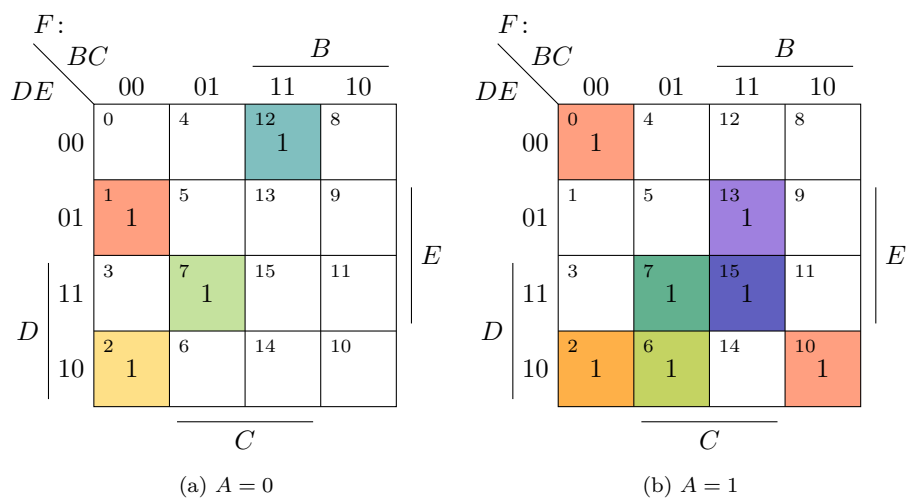
Answer:  $F = \overline{BCD} + \overline{BCD} + BCD + \overline{ABD} + \overline{ABC} \therefore$  Cost is 6

- 5-variable K-Maps take the form of  $f(A, B, C, D, E)$ . To solve these, we use two 4-variable K-Maps, one with  $A = 0$  (for 0-15) and one with  $A = 1$  (for 16-31). We combine adjacent 1's in three directions: vertically, horizontally, and out of the page. Below is an example.



Answer:  $F = AC + \overline{A}\overline{C}D + \overline{A}\overline{B}\overline{C}E$

- Problem 14.1.3: Solve the 5-variable K-Map  $F(A, B, C, D, E) = \sum(1, 2, 7, 12, 16, 18, 22, 23, 26, 29, 31)$ .



Distinguished 1's: 1, 12, 2, 7, 16, 26, 29

EPIs:  $\overline{A}BC\overline{D}\overline{E}, \overline{A}BCD\overline{E}, \overline{B}C\overline{D}\overline{E}, \overline{B}CDE, \overline{A}\overline{B}\overline{C}\overline{E}, \overline{A}\overline{B}CE, \overline{A}\overline{C}\overline{D}\overline{E}, \overline{A}\overline{B}CD$

$F = \overline{A}BC\overline{D}\overline{E} + \overline{A}BCD\overline{E} + \overline{B}C\overline{D}\overline{E} + \overline{B}CDE + \overline{A}\overline{B}\overline{C}\overline{E} + \overline{A}\overline{B}CE + \overline{A}\overline{C}\overline{D}\overline{E} + \overline{A}\overline{B}CD$

- Incompletely specified functions contain a don't care condition which is denoted by an 'X'. A don't care is an input condition that either can't

occur or even if it occurs it doesn't matter. A circuit with one or more don't cares is called an incompletely specified circuit.

- When forming prime implicant's, treat don't cares as a 1, but ignore them when actually selecting prime implicants. Never select a prime implicant only to cover a don't care.
- Problem 14.1.4: Draw the 5-variable K-Map  $F(A, B, C, D, E) = \sum(1, 9, 10, 17, 21, 27) + d(3, 5, 11, 25, 26)$ .

$F:$

$BC$		$B$			
		00	01	11	10
$DE$	00	0	4	12	8
	01	1	5 X	13	9 1
	11	3 X	7	15	11 X
	10	2	6	14	10 1
		$C$			

(a)  $A = 0$

$F:$

$BC$		$B$			
		00	01	11	10
$DE$	00	0	4	12	8
	01	1	5 1	13	9 X
	11	3	7	15	11 1
	10	2	6	14	10 X
		$C$			

(b)  $A = 1$

Distinguished 1's: 1, 9, 10, 17, 21  
 Essential prime implicants:  $\overline{BDE}, \overline{BCE}, \overline{BCD}$   
 Answer:  $F = \overline{BDE} + \overline{BCE} + \overline{BCD}$

- Problem 14.1.5: Draw the K-Map for  $F(A, B, C, D) = \sum(1, 6, 7) + d(3, 5, 13, 15)$ .

$F:$

$AB$		$A$			
		00	01	11	10
$CD$	00	0	4	12	8
	01	1	5 X	13 X	9
	11	3 X	7 1	15 X	11
	10	2	6 1	14	10
		$B$			

Distinguished 1's: 1, 6  
 Essential prime implicant(s):  $\overline{AD}, \overline{ABC}$   
 Answer:  $F = \overline{AD} + \overline{ABC}$



## 15 2/28/2020

### 15.1 Lecture Slides

- Thus far, we have been focusing on minimization with SOP, but what about POS? POS minimization is important to understand for considering other designs in the future. It has similar rules to SOP minimization, but flipped.
- Recall the “recipe” for SOP minimization from page 42. We will now construct a “recipe” for constructing a POS K-Map.
  1. Place all of the maxterms on the K-Map.
  2. Draw all of the PIs, starting with the largest size possible and then working your way down. All PIs must be powers of 2. This time, we cover 0's.
  3. Determine the distinguished 0's. Using these distinguished 0's, find the essential prime implicants.
  4. Begin creating the function using these essential prime implicants.
  5. Look at the map and check if there are any 0's not covered by prime implicants.
- Problem 15.1.1: Find the simplified POS for  $F = \prod_{w,x,y,z}(1, 2, 5, 6, 9, 10, 13, 15)$ .

$F$ :

		WX		W	
		00	01	11	10
YZ	00	0	4	12	8
	01	1 0	5 0	13 0	9 0
	11	3	7	15 0	11
	10	2 0	6 0	14	10 0
		X			

$Z$

Distinguished 0's: 1, 5, 6, 9, 10, 15

Essential prime implicant(s):  $(W + \bar{Y} + Z), (X + \bar{Y} + Z), (Y + \bar{Z}), (\bar{W} + \bar{X} + \bar{Z})$

Answer:  $F = (W + \bar{Y} + Z)(X + \bar{Y} + Z)(Y + \bar{Z})(\bar{W} + \bar{X} + \bar{Z})$

- Problem 15.1.2: Find the simplified POS for  $F = \prod_{w,x,y,z}(1, 2, 5, 6, 9, 10, 13, 15) + d(0, 3, 14)$ .

$F$ :

		$WX$		$W$	
		00	01	11	10
$YZ$	00	0 X	4	12	8
	01	1 0	5 0	13 0	9 0
	11	3 X	7	15 0	11
	10	2 0	6 0	14 X	10 0

$X$

$Z$

Distinguished 0's: 5, 6, 9, 10

Essential prime implicant(s):  $(Y + \bar{Z}), (\bar{Y} + Z)$

Answer:  $F = (Y + \bar{Z})(\bar{Y} + Z)(\bar{W} + \bar{X} + \bar{Z})$

- Problem 15.1.3: Find the simplified POS for  $F = \prod_{x,y,z}(1, 5, 6) + d(2, 7)$ .

$F$ :

		$XY$		$X$	
		00	01	11	10
$Z$	0	0	2 X	6 0	4
	1	1 0	3	7 X	5 0

$Y$

Distinguished 0's: 1

Essential prime implicant(s):  $(Y + \bar{Z})$

Answer:  $F = (Y + \bar{Z})(\bar{X} + \bar{Y})$

- You may also encounter a situation where you need to convert a function to a K-Map to ultimately end up with a simplified form.

- Problem 15.1.4: Convert the following function into a map and find the simplified SOP.  $F = \overline{A}BC\overline{D} + \overline{A}BCD + \overline{A}B\overline{D} + \overline{A}BC + \overline{A}BCD + \overline{A}BCD + \overline{A}BC + \overline{A}B\overline{D}$ .

$F$ :

		$AB$		$A$	
		00	01	11	10
$CD$	00	0 1	4	12	8 1
	01	1	5	13	9 1
$C$	11	3	7 1	15	11
	10	2 1	6 1	14	10 1
		$B$			

$D$

Distinguished 1's: 0, 7, 10, 10

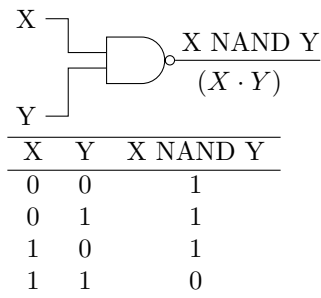
Essential prime implicant(s):  $\overline{A}BC$ ,  $\overline{B}D$ ,  $\overline{A}BC$

Answer:  $F = \overline{A}BC + \overline{B}D + \overline{A}BC$

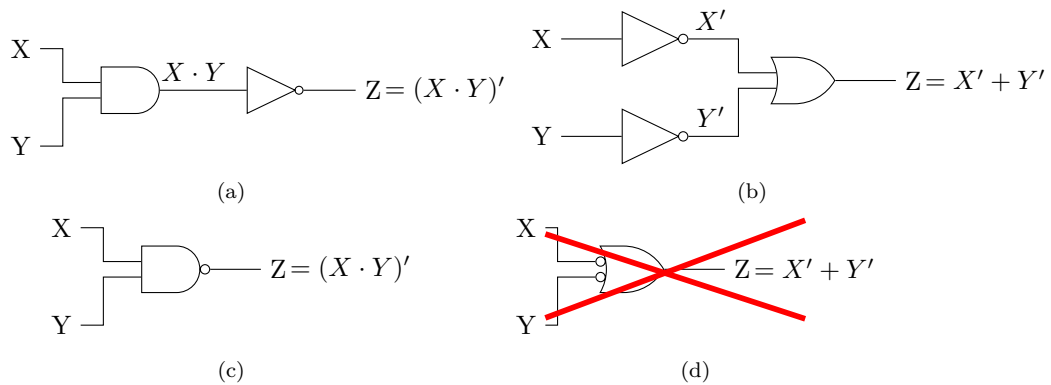
## 16 3/02/2020

### 16.1 Lecture Slides

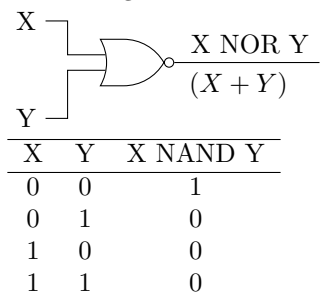
- We sometimes use other gates because they can be more efficient or convenient.
- The first gate we will go over is the NAND gate, a Not AND gate.



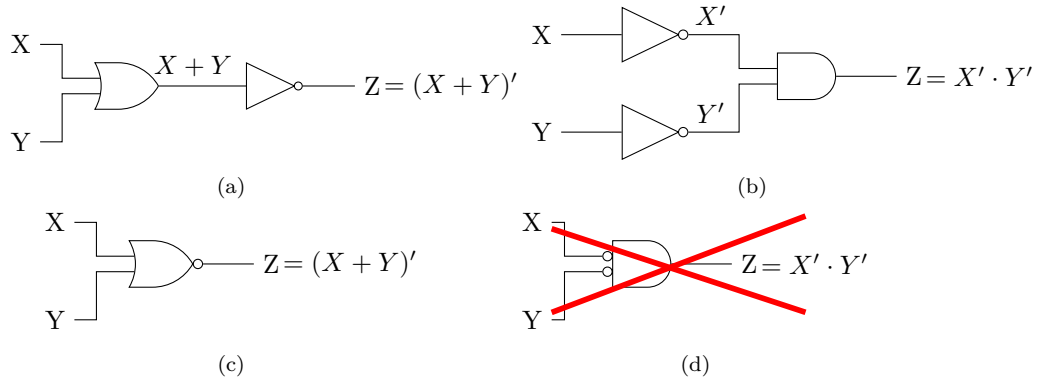
- Below are some other perspectives of a NAND gate.



- The next gate is the NOR gate, or a Not OR gate.



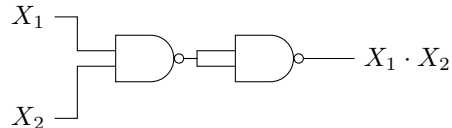
- Below are some other perspectives of a NOR gate.



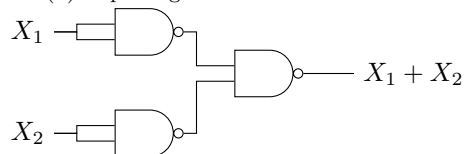
- Why do we use these new gates? The answer is simple: These are complete sets. One gate can be used to create a complete logic gate set (AND, OR, and NOT). Not having to mess around with other types of configurations makes setup much easier.
- In the end, it's all about mapping.



(a) Replacing a NOT with a NAND

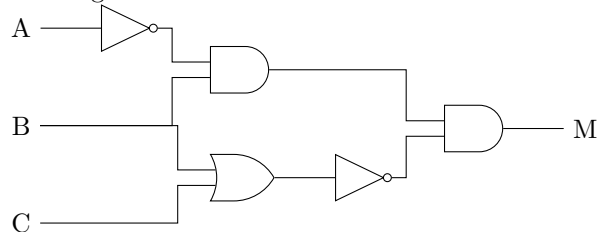


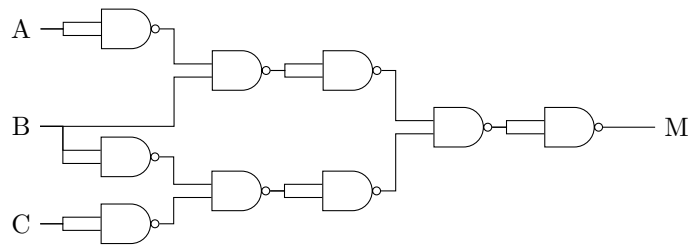
(b) Replacing an AND with a NAND



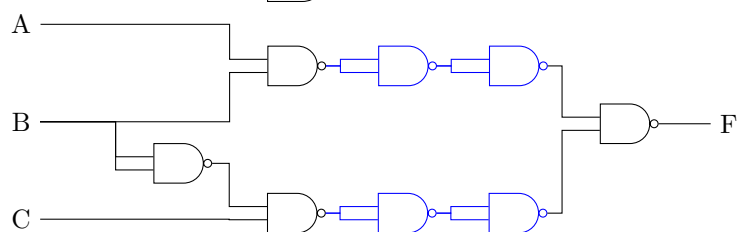
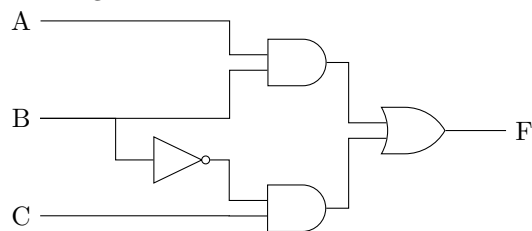
(c) Replacing an OR with a NAND

- So why does this work? In short, we are adapting DeMorgan's Law and the Involution Theorems.
- Problem 16.1.1: Convert the given logic diagram into one only using NAND gates.

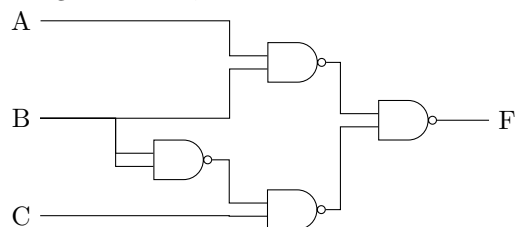




- Problem 16.1.2: Convert the given logic diagram into one only using NAND gates.



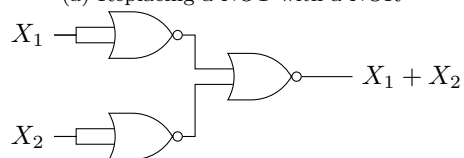
Using involution, we can cross out the blue elements, giving us...



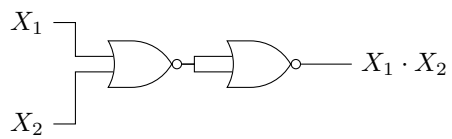
- We also have mappings for NOR gate conversions.



(a) Replacing a NOT with a NOR

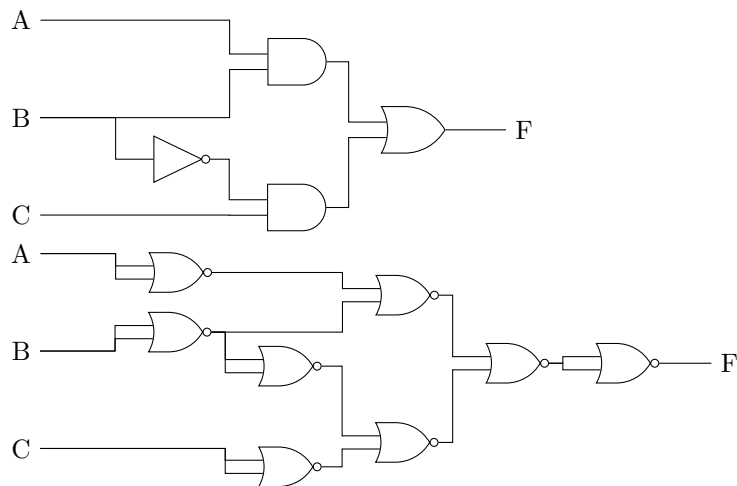


(b) Replacing an AND with a NOR

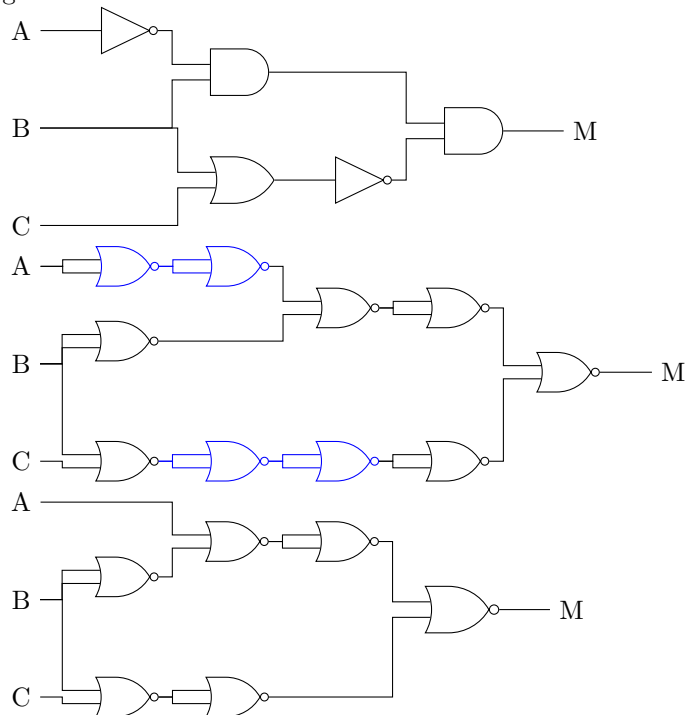


(a) Replacing an OR with a NAND

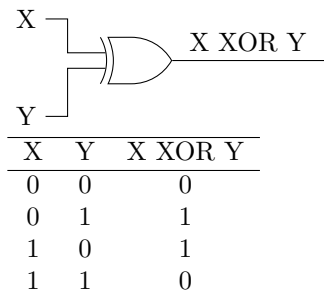
- Problem 16.1.3: Convert the given logic diagram into one only using NOR gates.



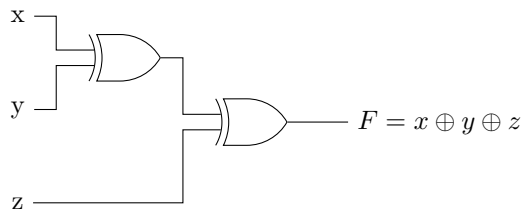
- Problem 16.1.4: Convert the given logic diagram into one only using NOR gates.



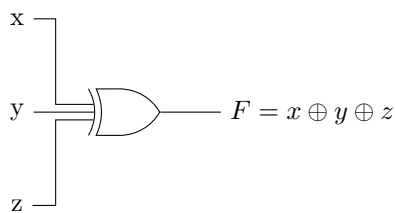
- Another new gate is the XOR gate.



- It can be tricky to recognize when XOR can be used. A general rule of thumb is not to use this type of gate when told to only use specific gates. You can use truth tables, arithmetic, k-maps, and inspection to otherwise find when to use the XOR gate.



(a) Using 2-input gates



(b) Using 3-input gate

$F:$

		$BC$		$B$	
		00	01	11	10
$A$	0	0	2	6	4
	1	1	3	7	5

$C$

(a) Odd function  $F = A \oplus B \oplus C$

$F:$

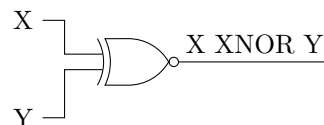
		$BC$		$B$	
		00	01	11	10
$A$	0	0	2	6	4
	1	1	3	7	5

$C$

(b) Even function  $F = (A \oplus B \oplus C)'$

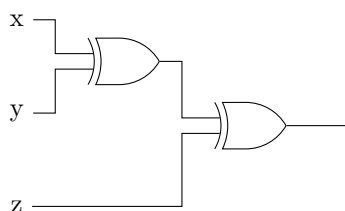


- The last new logic gate is the XNOR gate. XNOR is the even function and the complement of XOR. It is also  $(x \oplus y)' = xy + x'y'$ .

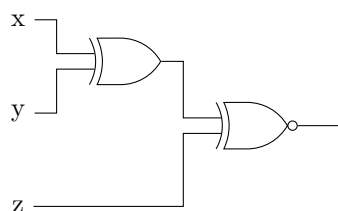


X	Y	X XNOR Y
0	0	1
0	1	0
1	0	0
1	1	1

- How do you break down a logic diagram when you have more than 2 inputs but simultaneously have a limited amount of gate inputs?



(a) 3-input odd function



(b) 3-input even function

## 16.2 Assigned Readings

- We previously discussed analysis methods to analyze the behavior of a circuit. However, these are flawed in that they only predict the steady-state behavior of a circuit. That is, they predict a circuit's output as a function of its inputs under the assumption that the inputs have been stable for a long time. However, the actual delay from an input change to the corresponding output change in a real logic circuit is nonzero and can depend on many factors.
- Because of circuit delays, the transient behavior of a combinational logic circuit may differ from what is predicted by steady-state analysis. In particular, a circuit's output may produce a short pulse, often called a glitch, at a time when steady-state analysis would suggest such an output wouldn't occur. A hazard is said to exist when a circuit has the possibility of producing such a glitch.
- Depending on how a circuit's output is produced, a system's operation might not even be adversely impacted by a glitch. When a glitch is harmful, however, it is up to the logic designer to be prepared to eliminate the hazards, or the possibilities of glitches occurring.

- A static-1 hazard is the possibility of a circuit's output producing a 0 glitch when we would expect the output to remain at a nice steady 1 based on a static analysis of the function. It has the following formal definition:

“A static-1 hazard is a pair of input combinations that: (a) differ in only one input variable and (b) both give a 1 output; such that it is possible for a momentary 0 output to occur during a transition in the differing input variable.”

- A static-0 hazard is the possibility of a 1 glitch when we expect the circuit to have a steady 0 output. It has the following formal definition:

“A static-0 hazard is a pair of input combinations that: (a) differ in only one input variable and (b) both give a 0 output; such that it is possible for a momentary 1 output to occur during a transition in the differing input variable.”

- Karnaugh maps can be used to detect static hazards in a two-level SOP or POS circuit. The existence or nonexistence of static hazards depends on the circuit design for a logic function.
- A properly designed two-level SOP (AND-OR) circuit has no static-0 hazards. A static-0 hazard would only exist in the circuit if both a variable and its complement were connected to the same AND gate. However, these can have static-1 hazards.
- For static-1 hazard analysis, we circle the product terms corresponding to the AND gates in the circuit and we search for adjacent 1 cells that are not covered by a single product term.
- Below is a Karnaugh map.

$F:$

		$XY$			
		00	01	11	10
$Z$	0	0	2	6 1	4 1
	1	1	3 1	7 1	5
		$Y$			

It should be immediately obvious that there is no single product term that can cover both combinations 111 and 110. Thus, it is theoretically possible for the output to momentarily “glitch” to 0 if the AND gate output that covers one of the combinations goes to 0 before the other can go to 1. This is solved by simply adding an extra product term (another AND gate) to over the hazardous pair, shown below.

$F:$

$XY$		$X$			
		00	01	11	10
$Z$	0	0	2	6 1	4 1
	1	1	3 1	7 1	5

$Y$

The extra product term is called the consensus, which is what we add to eliminate hazards.

- A properly designed two-level POS (OR-AND) circuit has no static-1 hazards. It can however have static-0 hazards, which are eliminated in a manner dual to the foregoing.

$F:$

$WX$		$W$			
		00	01	11	10
$YZ$	00	0	4 1	12 1	8
	01	1 1	5 1	13	9
$Y$	11	3 1	7 1	15 1	11 1
	10	2	6	14 1	10 1

$X$

(a) As originally designed

$F:$

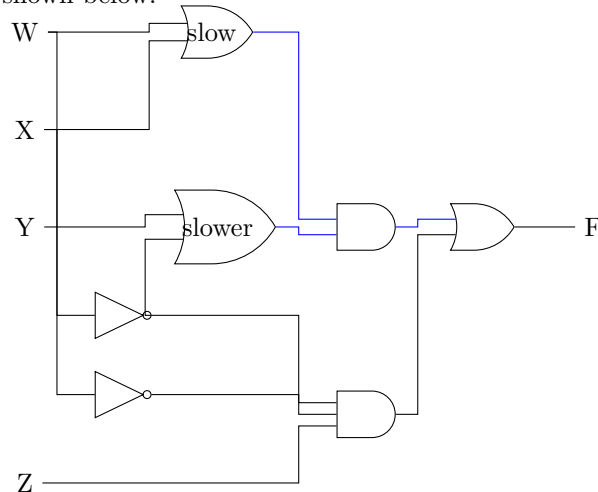
$WX$		$W$			
		00	01	11	10
$YZ$	00	0	4 1	12 1	8
	01	1 1	5 1	13	9
$Y$	11	3 1	7 1	15 1	11 1
	10	2	6	14 1	10 1

$X$

(b) With extra product terms to cover static-1 hazards

- A dynamic hazard is the possibility of an output changing more than once as the result of a single input transition. Multiple output transitions can occur if there are multiple paths with different delays from the input to the output. Dynamic hazards do not appear in properly designed two-level AND-OR or OR-AND circuits.

- Let's go over an example of a dynamic hazard. Consider the below circuit. It has three paths from input X to output F. One of the paths goes through a slower OR gate and another goes through an even slower OR gate. If the circuit's input is  $W, X, Y, Z = 0, 0, 0, 1$ , then the output will be 1, as shown below.



Now, let's suppose we change the X input to a 1. Assuming that all of the gates except the two "slow" gates are very fast, the transitions not marked blue will occur next, and the output goes to zero. Eventually the output of the "slow" gate changes and the output becomes a 1. Finally, the output of the "slower" gate changes and the output is finally at 0.

- Only a few situations (such as the design of feedback sequential circuits) require hazard-free combinational circuits. Methods for finding hazards in arbitrary circuits can be difficult, so if you absolutely need a hazard-free design, it is best to utilize a circuit structure that is simple to analyze.
- If cost isn't a problem, a brute force method for designing a hazard-free circuit is just to use the complete sum of the logic function.
- Everything that has been said about AND-OR circuits applies to NAND-NAND circuits, and everything said about OR-AND circuits applies to NOR-NOR circuits.
- It is important to note that most hazards aren't actually that dangerous. Any combinational circuit can be analyzed for the presence of a hazard, however any well designed synchronous digital system is specifically structured to prevent the occurrence of hazards. Hazard analysis is typically only necessary in asynchronous sequential circuits.

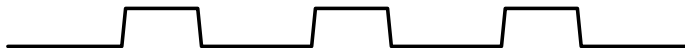
## 17 3/04/2020

### 17.1 Lecture Slides

- Time is officially a component of circuit designs, now. Why? “Time is real,” and getting used to it now will make future designs much more practical.
- We look at time using a timing diagram, which is just a twist on the normal truth table. 0 is represented as a low wave value and 1 is represented as a high wave value.



- While we do operate in 0 and 1, the values always exist and are continuous. Timing diagrams are what we actually work with and show behaviors such as verifying functionality and time impacts.



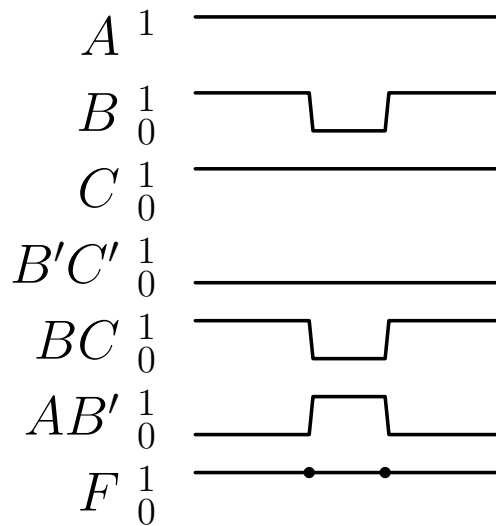
- Three variables aren't enough to draw a timing diagram for a function such as  $F = B'C' + BC + AB'$ . Those three variables only have eight possible combinations, which isn't enough. To draw the timing diagram, we need to look at all of the possible combinations.
- Determining all of the potential transitions can be annoying. We need to consider design behaviors.
- Recall that Gray Code is used with sensors because there is less change between values leaving less room for error. We can use a similar approach here to identify places where errors can occur. To find our “edge cases,” we just need to look at our K-Map. First we are going to identify where these edge cases can occur, and then we will understand what is going on at these edge cases.
- An edge case, when working in SOP, is a transition between cells containing minterms. Specifically, these occur with adjacent, but not overlapping, prime implicants. A transition inside of a prime implicant doesn't matter as its stable.

- Problem 17.1.1: Draw the timing diagram for  $F = B'C' + BC + AB'$ .

$F$ :

		$AB$		$A$	
		00	01	11	10
$C$	0	0 1	2	6	4 1
	1	1	3 1	7 1	5 1
		$B$			

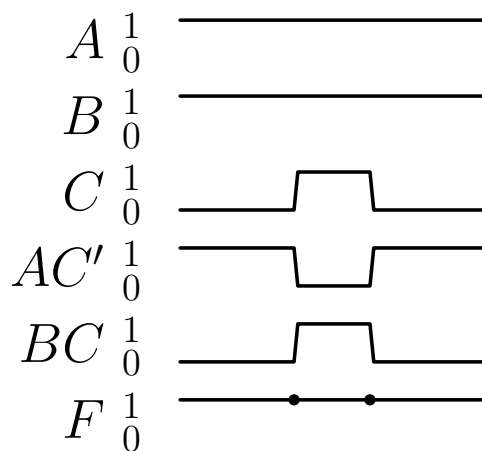
The transition between  $111 \leftrightarrow 101$  is left uncovered (Note: Yes, you can draw implicants connecting these two but here she only drew the original function)!



- Problem 17.1.2: Draw the timing diagram for  $F = AC' + BC$ .

$F$ :

		$AB$		$A$	
		00	01	11	10
$C$	0	0	2	6 1	4 1
	1	1	3 1	7 1	5
		$B$			



- Transition times in gates and other electronical technologies result in hazards. Let's first go over static hazards, which exist when two adjacent one's in a K-Map are not covered by a prime implicant in the resulting minimal function. To fix it, include additional prime implicants whenever there are two adjacent one's. While the resulting function will no longer be minimal, there will be no undesired behavior.
- A hazard can also be described as the brief period of time where the value of an output changes even though the input's change should have left the output unaffected. A static 1 hazard is occurs when it should stay 1, and a static 0 hazard occurs when it should stay 0.

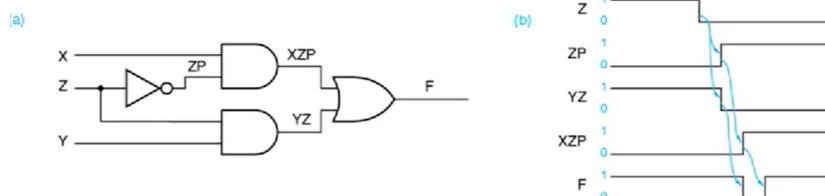


(a) Static hazard



(b) Dynamic hazard

- Static-1 Hazards look like this between a circuit diagram and the timing diagram.

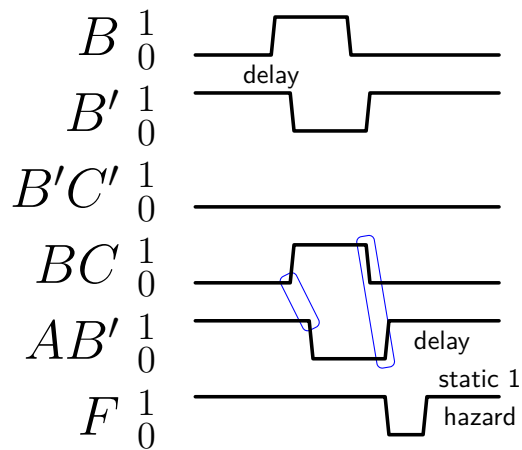


- Problem 17.1.3: Identify the hazard for  $F = B'C' + BC + AB'$ .

$F$ :

		$AB$		$A$	
		00	01	11	10
$C$	0	0 1	2	6	4 1
	1	1	3 1	7 1	5 1
		$B$			

If you specify that B is the only factor (it is the only thing that changes in  $111 \leftrightarrow 101$ ), you don't have to draw A and B in the timing diagram.

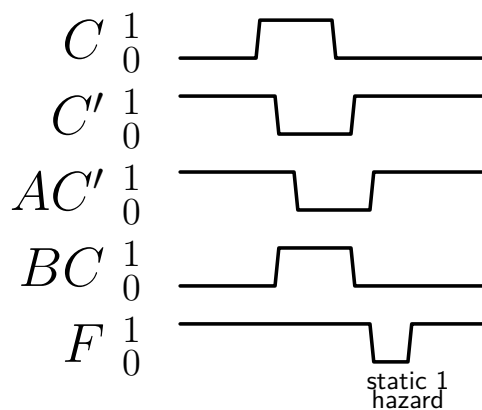


- Problem 17.1.4: Identify the hazard for  $F = AC' + BC$ .

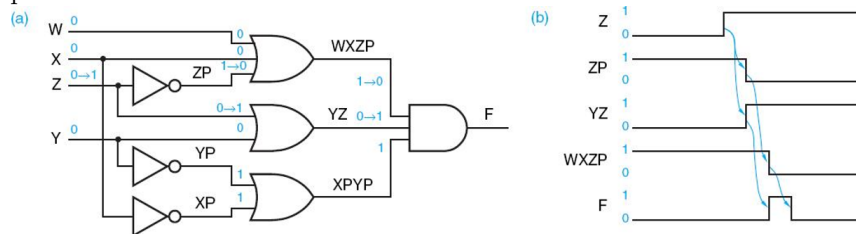
$F$ :

		$AB$		$A$	
		00	01	11	10
$C$	0	0	2	6 1	4 1
	1	1	3 1	7 1	5
		$B$			





- You don't need to know much about static-0 hazards besides "they happen in POS."



- So how do we fix these? We need to review a previously discussed subject: Function equivalence. This is when two functions accomplish the same thing but with different terms. This is crucial in making things smaller and faster, but can come at the price of getting hazards.

## 17.2 Assigned Readings

- An adder combines two arithmetic operands using the addition rules previously discussed.
- An adder can perform subtraction as the addition of the minuend and the complemented negated subtrahend, but you can also build subtractors that perform subtraction directly.
- The simplest adder is called a half adder, and it adds two 1-bit operands  $A$  and  $B$  producing a 2-bit sum. The sum can range from 0 to 2 in base 10 and requires two bits to express. The low-order bit of the sum can be named the HS (or half sum) and the high-order bit can be named CO (or carry-out). We can write the following equations for HS and CO.

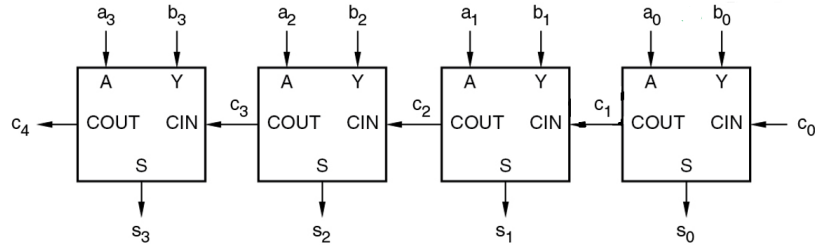
$$\begin{aligned}
 HS &= A \oplus B \\
 &= A \cdot B' + A' \cdot B \\
 CO &= A \cdot B
 \end{aligned}$$

- To add operands with more than one bit, we must also provide for the carries between bit positions. The building block for this sort of operation is called a full adder. Besides the addend-inputs  $A$  and  $B$ , a full adder also has a carry-bit input ( $C_{in}$ ). The sum of the three inputs can range from 0 to 3, which can still be expressed with just two output bits,  $S$  and  $C_{OUT}$ .

$$\begin{aligned}
 S &= A \oplus B \oplus C_{in} \\
 &= A \cdot B' \cdot C'_{in} + A' \cdot B \cdot C'_{in} + A' \cdot B' \cdot C_{in} + A \cdot B \cdot C_{in} \\
 C_{OUT} &= A \cdot B + A \cdot C_{in} + B \cdot C_{in}
 \end{aligned}$$

Here,  $S$  is 1 if an odd number of the inputs are 1 and  $C_{out}$  is 1 if two or more of the inputs are 1.

- Two binary words, each with  $n$  bits, can be added using a ripple adder, or a cascade of  $n$  full-adder stages each handling a single bit. The circuit for a 4-bit ripple adder looks like this.



The carry input to the least significant bit (in this case  $c_0$ ) is normally set to 0 and the carry output of each full adder is connected to the carry input of the next most significant full adder.

- A ripple adder is slow since in the worst case a carry must propagate from the least significant full adder to the most significant one.
- A faster adder must be made. This can be done by obtaining each sum output  $s_i$  with just two levels of logic, accomplished by writing an equation for  $s_i$  in terms of  $x_0-x_i$ ,  $y_0-y_i$ , and  $c_0$ , a total of  $2i + 3$  inputs. Then, you “multiply/add out” to obtain an SOP or POS expression and build the corresponding circuit. Unfortunately, beyond  $s_2$  the resulting expressions have too many terms, limiting the usage of this method.
- A full subtractor handles one bit of the binary subtraction algorithm, having inputs  $A$  (the minuend),  $B$  (the subtrahend), and  $B_{in}$  (the borrow in). It also has the outputs  $D$  (difference) and  $B_{out}$  (borrow out). We can write logic expressions corresponding to binary subtraction as follows:

$$\begin{aligned}
 D &= A \oplus B \oplus B_{in} \\
 B_{out} &= A' \cdot B + A' \cdot B_{in} + B \cdot B_{in}
 \end{aligned}$$

- Any  $n$ -bit adder circuit can function as a subtractor by complementing the subtrahend and treating the carry-in and carry-out signals as borrows with the opposite active level.

- The most well known method to speed up adders are called carry lookaheads. The logic equation for sum bit  $i$  of a binary adder can actually be written simply as  $s_i = a_i \oplus b_i \oplus c_i$ .
- While all of the addend bits are normally presented to an adder's inputs and are valid almost simultaneously, the output of this above equation is invalid until the carry input is valid. This can be a problem in ripple-adder designs where it takes a long time for the most significant carry input bit to be valid.
- A carry-lookahead adder uses three-level equations in each adder stage. Each stage's sum output is produced by combining its carry bit above with two addend bits.
- In any given technology, the carry equations beyond a certain bit position cannot be implemented effectively in just three levels of logic, for they would require gates with too many inputs. Wider AND and OR functions can be build with two or more levels of logic, but a more economical approach is to use carry lookahead only for a small group where the equations can be implemented in three levels and then use ripple carry between groups.
- A 74x283 is an MSI 4-bit binary adder that forms its sum and carry outputs with just a few levels of logic using the carry-lookahead technique.
- Fast group-ripple adders with more than four inputs can be made by cascading the carry outputs and inputs of 283's.
- We can take carry lookaheads even further by creating group-carry-lookahead outputs for each  $n$ -bit group and combining these into two levels of logic to provide the carry inputs for all of the groups without rippling carries in between them.

## 18 3/06/2020

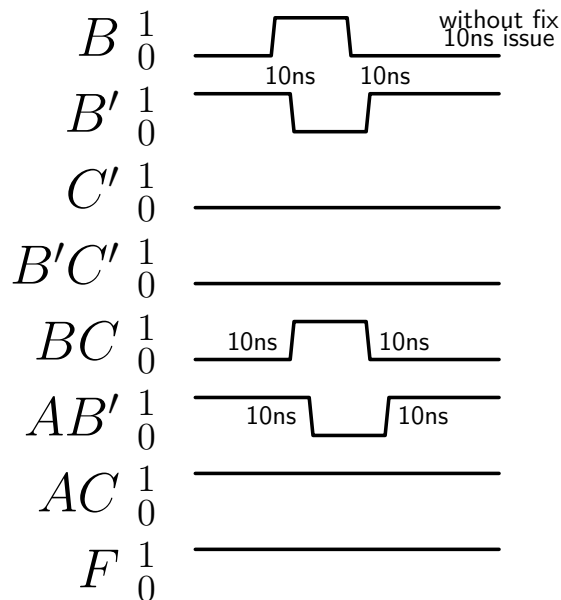
### 18.1 Lecture Slides

- Problem 18.1.1: Fix the hazard for  $F = B'C' + BC + AB'$ .

$F$ :

		$AB$				$A$			
		00	01	11	10				
$C$	0	0 1	2	6	4 1				
	1	1	3 1	7 1	5 1				
		$B$							

The hazard occurs at  $111 \leftrightarrow 101$ , so we can add  $AC$  to fix the hazard.

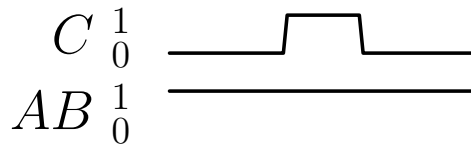


- Problem 18.1.2: Fix the hazard for  $F = AC' + BC$ .

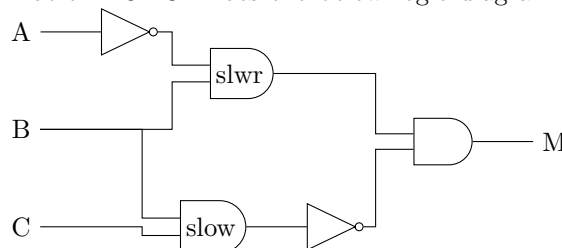
$F$ :

		$AB$				$A$			
		00	01	11	10				
$C$	0	0	2	6 1	4 1				
	1	1	3 1	7 1	5				
		$B$							

To fix the hazard, we need to fix  $AB$ , which is where the transition  $110 \leftrightarrow 111$  occurs. Since  $C$  is the only value changing, we can exclude  $A$  and  $B$  from the timing diagram.

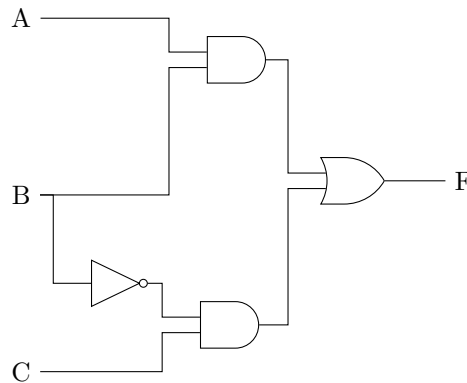


- Dynamic hazards occur when the output is supposed to change but oscillates for a brief period of time before settling down. These occur because of race conditions. We identify these through timing diagrams, logic gate inspections, and analysis.
- Problem 18.1.3: Does the below logic diagram have a dynamic hazard?



This logic diagram doesn't have any dynamic hazards, as they all cancel out. The delay the first not gate introduces to the slwr (stand-in for slower) gate is equivalent to the delay introduced to the second not gate by the slow gate.

- Problem 18.1.4: Does the below logic diagram have a dynamic hazard?



This logic diagram does indeed have a dynamic hazard. The first path ( $A \rightarrow \text{and} \rightarrow \text{or} \rightarrow F$ ) goes through only one logic gate, the and gate. The second path ( $B \rightarrow \text{not} \rightarrow \text{and} \rightarrow \text{or} \rightarrow F$ ), however, goes through two, which will introduce greater delay and thus brings a race condition.

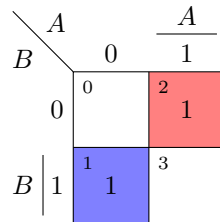
- So how do we solve these dynamic hazards? It's complicated. You should first try a two level design, described below. If that doesn't work, you can also try clocked synchronization.
  - If SOP: Check your not gates  $\rightarrow$  and gates  $\rightarrow$  or gates.
  - If POS: Check your not gates  $\rightarrow$  or gates  $\rightarrow$  and gates.

There should be a uniform number of gates in all paths.

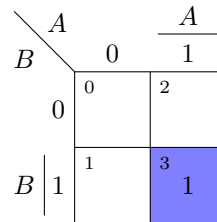
- MSI stands for medium scale integration and involves from 20 to about 200 logic gates. Our AND gate IC has four gates, making it SSI. MSI is the direction of complexity we will soon be approaching.
- The MSI combinational logic devices we will be looking at are adders, subtractors, multiplexors, decoders, encoders, and shifters.
- For this next section, you must recall binary addition. If you need a refresher, go read pages 6-8.
- The half adder is, bluntly, the addition that happens on the right most bit. There is no carry in, which is a key component of half adding. Furthermore, a sum bit and a carry out bit are produced.
- Below is an example of an implemented half adder.

Arithmetic: $A + B$		Carry bit	Sum bit
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

This half adder table essentially creates two K-Maps, both shown below.

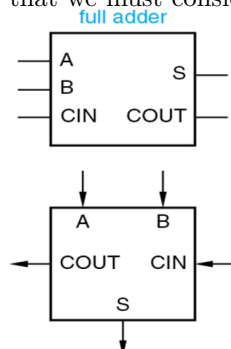


(a)  $S = A'B + AB' \rightarrow S = A \oplus B$



(b)  $C = AB$

- A full adder is what we use to consider the remainder of the bits when working with addition between two numbers. There is now also a carry in that we must consider.



- So what does a full adder look like when implemented?

$A + B + C_{in}$			$C_{out}$	$S$
$A$	$B$	$C_{in}$		
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

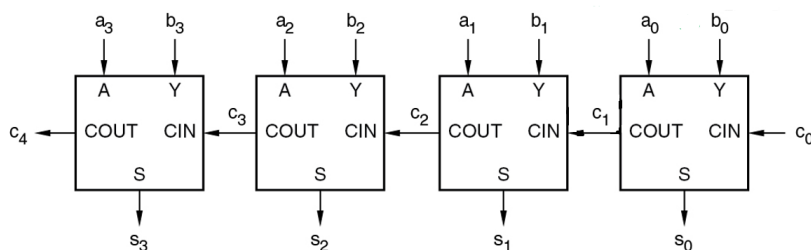
$AB$		$A$			
$C$	$B$	00	01	11	10
		0	2	6	4
0			1		1
$C$	$B$	1	3	7	5
		1		1	

(a)  $S$

$AB$		$A$			
$C$	$B$	00	01	11	10
		0	2	6	4
0				1	
$C$	$B$	1	3	7	5
			1	1	1

(b)  $C$

- If we want to calculate multiple complex numbers, we need to chain together multiple FAs, because at the end of the day one FA is still a single bit.



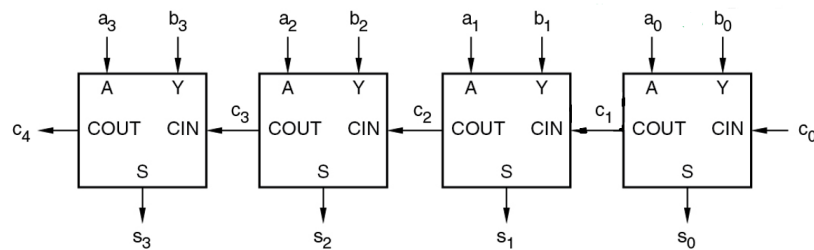
- The biggest issue with this approach comes from propagation delays. The FA's need a long time until they get the correct value, which can significantly lower overall efficiency.
- The solution to this comes with fast adders, which are adders specifically designed to address propagation delay. We are only briefly looking at these to understand their purpose and key ideas.
- Propagation delays are a significant problem with ripple delays, so the solution is just to carry out all of the delays at once. The only problem is that this can create *extremely* complicated.
- To avoid "gate explosion," we divide and conquer. This is known as HCLA or the Hierarchical Carry Look Ahead Adder (also known as the Grouped Carry Lookahead Adder).

## 19 3/22/2020

### 19.1 Time

#### 19.1.1 Lecture Slides

- Let's begin with some review. Recall what signals are from 17.1. We operate in the continuous and ever-present values 0 and 1. These values are used to actually show the behaviors of functions, including verifying functionality and time impacts.
- Around this point of class, we also began discussing why time was suddenly important in understanding circuits. Why? Because time *is* real, and getting used to it now will assist with making future designs more practical.
- Hardware specifications allow us to actually find where time-related issues occur, because nothing is ideal. Hardware specifications can generally be found in locations such as datasheets.
- One example of a time concern can be found in ripple carries. Observe the below picture.



The first adder must be completed in order for the correct  $c_1$  value to be obtained. This propagation delay stockpiles over time leading to a substantial time concern.

- Transitions between low and high take time, and more specifically they take energy. A transition in a gate from low to high will cost more than a transition from high to low. In other words, it takes more time to build up charge than it does to release it.
- To control the time between transitions, we need to use a physical device that stabilizes the difference between transitions. This physical device is called a capacitor. Capacitors are our go to device to control the rate of change when moving from high to low.

#### 19.1.2 Assigned Reading

- The outputs of real circuits take time to react to their inputs. In fact, many of today's circuits and systems are so fast that even the speed-of-light delay in "propagating an output signal to an input on the other side of a board or chip" can be significant.



- Most digital systems are sequential circuits that operate step-by-step under the control of a periodic clock signal. The speed of said clock is usually limited by the worst-case time it would take for the operations in one step to complete. It is appropriately paramount that a digital designer always be keenly aware of timing behaviors in order to be able to build a circuit that always operates correctly regardless of the conditions.
- A timing diagram is used to illustrate the logical behavior of signals in a digital circuit as a function of time. An important part of the documentation for any digital system, these diagrams are also used to explain the relationship between signals within a system as well as to define the timing of external signals that are applied to and produced by a module. These characteristics are called timing specifications.
- The time it takes for a signal to change from one state to the other is called the transition time. More specifically, the time to go from LOW to HIGH is called the rise time and the time to go from HIGH to LOW is called the fall time.
- Arrows are occasionally drawn (particularly in complex timing diagrams) to show causality, or which particular input transitions cause specific output transitions.
- The most important information provided by a timing diagram is the specification of a delay between transitions. Different paths through a circuit may have different delays. Furthermore, the delay through any given path may vary depending on whether the output is changing from LOW to HIGH or from HIGH to LOW.
- Most timing diagrams are accompanied by a timing table that specifies each delays amount and the specific conditions under which it applies. Since the delays of real digital environments can be affected by a vast variety of factors, the delay is rarely specified as a single number and is more commonly given as a range of values.
- The propagation delay of a signal path is the time it takes for a change at the input of the path to produce a change at the output of the path. Propagation delays depend on the internal analog design of the circuit as well as a variety of operational characteristics, including but not limited to...
  - Power-supply voltage
  - Temperature
  - Output loading
  - Input rise and fall times
  - Transition direction
  - Speed-of-light delays
  - Noise and crosstalk
  - Manufacturing tolerances

- With all of these difference sources of timing variation, it simply isn't practical to determine the exact timing of a particular application and environment. Fortunately, we don't have to. The industry standard is making good engineering estimates based on the "maximum," "minimum," and "typical" propagation delays specified by IC manufacturers.
- A logic designer who combines ICs in a larger circuit can use individual device specifications to analyze the overall circuit timing. For simpler designs this can be done by hand, but for more complex designs a timing analysis program can be used. In both cases the delay of a path through the overall circuit is computed as the sum of the delays through the individual devices on the path plus speed-of-light delays, assuming they are not negligible.
- While delays can be highly dependent on the signal routing of a chip, the final routing is unknown until the circuit's design has been completed. Thus, timing analysis is typically analyzed at two different stages in the design. First, the timing can be estimated once the logic design is complete using the known timing for the individual logic elements and an estimate of routing delays based on factors the designer is already aware of at that stage. Later, once the logic elements have been physically placed on the chip, it is possible to calculate the expected delay more precisely using the details of placed elements, wire lengths, and other factors.
- A manufacturer's timing specifications for a device may give a minimum, typical, and maximum value for each propagation delay path and the transition direction. Here is what each of those mean.
  - Maximum: The most commonly used specification by experienced designers since a path "never" has a propagation delay longer than the maximum. Amusingly, the definition of "never" can vary between logic families and manufacturers.
  - Typical: The specification most often used by designers who don't expect to be around when "their product leaves the friendly environment of the engineering lab and is shipped to customers." The typical delay is what you would see for a device manufactured on a good day operating under near-ideal conditions. Because of the danger on relying on such a idealistic number, some manufacturers have stopped using them for many newer CMOS logic families.
  - Minimum: The smallest propagation delay that a path will ever exhibit. Most well-designed circuits don't depend on this number, as the circuit should already work properly if the delay is zero. However, in some very high-speed logic families, a nonzero minimum delay is specified to allow the designer to ensure that hold-time requirements of latches and flip-flops are met.
- To permit for a simplified "worst-case" analysis, many board-level designers will use a single worst-case delay specification that provides the worst-case conditions of voltage and temperature. While this is a pessimistic view of the overall circuit delay, it always works and saves analysis time.

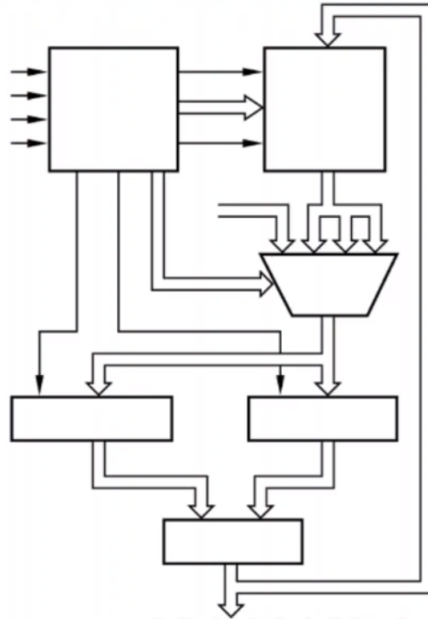
- To accurately analyze the timing of a circuit containing more than a few gates, a designer may have to study its logical behavior with more scrutiny and in excruciating detail. A moderate-sized circuit can have many different paths from a set of input signals to a set of output signals. Thus, to determine the minimum and maximum delays of a circuit, you must look at every possible path.
- Of course, since a combinational circuit can potentially have a path between every input and every output, and in some paths a single may even fan out to multiple internal paths, analyzing the different delay paths in a large circuit is usually only practical with the assistance of automated tools.
- Even through the assistance of a simulator, it is still up to the designer to supply the input sequences for which the simulator should produce outputs for. Thus, a designer would still need to have a good feel of what to look for and how to stimulate the circuit in order to produce and observe worst-case delays.
- Instead of using a simulator and supplying your own input sequences, you can use a timing analysis program (or a timing analyzer) instead. Such a program can automatically find all possible delay paths and print out a sorted list of them starting with the slowest. However, these results can be overly pessimistic, as some paths may not actually be used in the normal operation of a circuit, thus a designer must still “use some intelligence and experience” in order to interpret the results properly.
- On the first attempt, the results for a “final” realized circuits may not meet the requirements of the design, whether it be because the circuit is too slow, or parts are too fast, etc. When this happens, the designer must change parts of the circuit as necessary and then re-synthesize the circuit. Afterwards, the timing results must be checked again and this process must be repeated until the performance goals are met. This is called timing closure and with larger projects can take multiple months.

## 19.2 Documentation

### 19.2.1 Lecture Slides

- Generally speaking, documentation is critical in allowing both proper design and efficient maintenance. Documentation is a crucial all-encompassing idea, taking many forms. Some cases necessitate all types of documentation while others do not.
- As we get into more and more complicated designs and ideas, knowing how to understand documentation will prove to be very helpful.
- Documentation is a key component of communication in both the industry and in research. For computer scientists, this can be important because of the language aspect of the craft. For engineers, documentation is very important with the collaborative and evolutionary structure of the field.

- Specifications, often referred to as specs, describe what a system should do. Specs can include inputs, outputs, and functions. Specs are the what, not the how.
- A block diagram is a high level graphical overview used in documentation. It shows function modules and basic interconnections.



- A logic-device description is used in custom functions. These are commonly seen with ASIC, FPGA, PLD, and CPLD type devices. These are mostly in English, but internal specifications generally need another approach to describe this type of behavior.
- A BOM, or Bill of Materials, is the list of components that will be needed for implementation.
- Problem 19.2.1: Create a BOM for the function  $F = AB + CD$ .
  - 2 AND gates
  - 1 OR gate
  - 5 resistors
  - 5 LEDs
- Test plans are methods and resources used to verify and test the proper operation of a design. These are used both before and after construction. A truth table, for example, tests LED behavior. A timing diagram is another example of a test plan, used for finding hazards.
- Active levels are a way of designating what voltage goes with what discrete value. Up until this point, we have considered 1 to be true; this line of thinking is called Active High. The opposite, where 0 is considered true, is called Active Low. Active Low is used for improved compatibility with other devices, better electrical performance, and reduced noise issues.

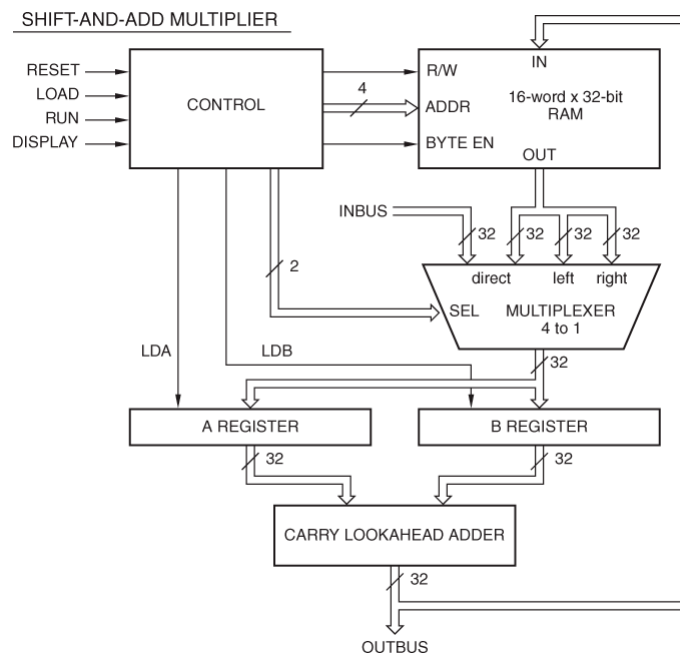
- Names are arbitrary yet an important skill to get good at. A good name tells you what the purpose of a signal, input, or function is. Names can vary between active high and active low, as shown in the above table. Active high is the default logic system, so many active high names don't come with additional symbols added to signify it as active high.
- Busses are a communication channel used to transmit data and power. Each wire in a bus has a specific purpose. While colors don't matter, they do help with construction. The connections, however, do matter.

### 19.2.2 Assigned Reading

- Good documentation is essential for the correct design and efficient maintenance of digital systems. In addition to being accurate and complete, documentation must also be somewhat instructive so that almost anybody can figure out how a system works just by reading its documentation.
- Although the type of documentation depends on a system's complexity and its environment, all documentation packages should (generally) contain at least the following items.
  1. A specification (sometimes shorthand as spec) describes exactly what the circuit or system is supposed to do, including a description of all inputs and outputs, as well as a list of functions that are to be performed. The spec doesn't have to specify how the system achieves its results, just what those results are supposed to be.
  2. A block diagram is an informal pictorial description of the system's major functional modules and their basic interconnections.
  3. A logic-device description describes the functions of each "custom" logic device used in the system ("standard" devices are described by data sheets or manufacturer-issued user manuals). Custom devices include application-specific integrated circuits (ASICs), field-programmable gate arrays (FPGAs), and programmable logic devices (PLDs and CPLDs).  
At a high level, these device descriptions are written in English, but the internals are usually described in an HDL like Verilog. Some internals can also be specified using logic diagrams, equations, state tables, or state diagrams. Sometimes, a programming language like C is even used to model the operation of a circuit.
  4. A schematic diagram is a formal specification of the electrical components of the system, their interconnections, and related details needed to build the system. Previously, we have drawn logic diagrams, which are less formal drawings that don't have this level of detail.
  5. A timing diagram shows the values of various logic signals as a function of time, including the cause-and-effect delays between critical signals.
  6. A circuit description is a narrative text document that in conjunction with other documentation explains how the function works internally. This circuit description should list any assumptions and potential

pitfalls in the design of the system, while also bringing attention to any non-obvious design “tricks.”

7. Finally, a test plan describes the methods and resources that will be necessary to test the system for proper operation, both before and after it is constructed.
- A block diagram shows the inputs, outputs, functional modules, internal data paths, and important control signals of a given system. Larger systems are designed and described hierarchically. At the top level, and in the corresponding block diagram, the system is partitioned into a small number of independent subsystems or blocks that interact with each other in well defined ways. Each of these subsystems or blocks is further partitioned until an appropriate level of detail has been reached. Below is a sample block diagram. Each block is labeled with the function of the block, not the individual components that comprise it.



- A bus is a collection of two or more related signal lines. In a block diagram, buses may be drawn with a double or heavy line, as shown above. A slash and a number can be used to indicate how many individual signal lines are contained in a bus. Size can also be denoted by the bus' name. Active levels and inversion bubbles may or may not appear in block diagrams, but even if they do they are usually unimportant at this level of detail.
- A buffer, shown below, is a circuit that converts an electrically weak logic signal one into a stronger one with the same the same logic value.



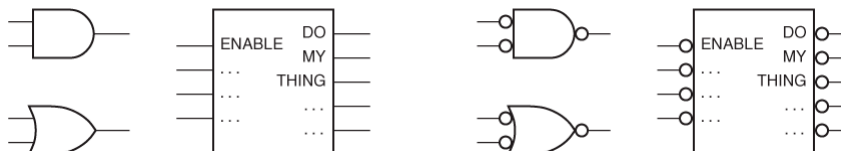
- A small circle, called an inversion bubble, denotes logical inversion or complementing.



- Each input and output signal in a logic circuit should have a descriptive alphanumeric label representing the signal's name. HDL and most EDA programs for drawing logic circuits usually allow the use of certain special characters, such as \* and \$, to be used in signal names. In a real system, well-chosen signal names convey information the same way that variable names in a software program do.
- Each signal name should have an active level associated with it. A signal is active high if it performs the named action or denotes the named condition when it is HIGH or 1. A signal is active low if it does the inverse and operates at LOW or 0. A signal is said to be asserted when it is at its active level and a signal is said to be negated when not.
- The active level of each signal in a circuit is normally specified as part of its name, based off of some convention. Examples of several different active-level naming conventions are shown below.

Active Low	Active High
READY-	READY+
ERROR.L	ERROR.H
ADDR15(L)	ADDR15(H)
RESET*	RESET
ENABLE~	ENABLE
GO	GO
/RECEIVE	RECEIVE
TRANSMIT_L	TRANSMIT

- It is important to be able to differentiate between signal names, expressions, and equations. A signal name is just a name, in other words just an alphanumeric label. A logic expression combines signal names using switching algebra operators. A logic equation is an assignment of a logic expression to a signal name.
- To represent a circuit between active levels, inversion bubbles are used. In the below circuit, the leftmost elements are active high and the rightmost active low.



## 19.3 Verilog Introduction

### 19.3.1 Lecture Slides

- Software is a big part of digital design and exploits electronic design automation, or EDA. Automation in this sense is focused on improving the efficiency of the productivity, verification, and quality.
- Software allows for multiple design approaches and considerations in testing. Computer aided engineering, or CAE, is all about the front end of the system and encompasses all of the design and testing tools. Computer aided design, or CAD, is the back end of the system where things like wire routing happen. CAD is typically a tool for non-electric things since we want the backend to do the work for us.
- Key design is all about the design and problem solving of a system. This means that answers are useless.
- We can tackle key design using a programming language called Verilog, which is an HDL, or a hardware description language. When we get to more complicated concepts, creating diagrams and subsequently testing logic can get tedious.
- Verilog is only one of the HDLs and was developed in 1984 with C-like syntax. In the 1990's, HDLs allowed for the acceleration of technology, resulting in things like PLDs, CPLDs, and FPGAs becoming cheaper and much more common. However, ASIC also became more difficult to design as a direct result of this as well.
- Presently, we use the IEEE 1364 standard for Verilog: Verilog95 and Verilog01. Another alternative is the Department of Defense approved VHDL. Recently, two more variants have surfaced in System C and SystemVerilog.
- The key concept behind Verilog is that execution is not strictly linear.
- So what is Verilog? Verilog, as previously mentioned, is an HDL, or a hardware description language. It is a modeling language, not just computational, and looks like C. It creates hardware structures, not GUIs, for performing complex calculations.
- Verilog is a great prototype and development tool. Field Programmable Gate Arrays, or FPGAs, are able to be configured using the design specified for testing before even manufacturing the design.
- Verilog is worth using for a variety of reasons, including...
  - Design Entry
  - Logic Simulation
  - Logic Synthesis
  - Timing Verification
  - Fault Verification



### 19.3.2 Assigned Reading

- In the 1990s, HDL usage by digital system designers began to accelerate as PLDs, CPLDs, and FPGAs became inexpensive and commonplace. Simultaneously, as ASIC densities continued to increase, it became more difficult to describe large circuits using schematics alone, causing many ASIC designers to turn towards HDLs as a means of designing individual modules within a system-on-a-chip. Today HDLs are by far the most common way to describe both the top-level and detailed module-level design of an ASIC, FPGA, or CPLD.
- Digital design activity is ever moving to higher levels of abstraction. This mode has been both enabled and necessitated by decreasing cost per function and the ever higher level of functionality and integration that can be achieved on a single chip.
- On traditional software design, high-level programming languages like C, C++, and Java raise the level of abstraction to a point that designers can create larger, more complex systems, albeit with a sacrifice in performance. However, we would have *no* performance in today's complex software systems if they had to be written in assembly language as they never would have been finished.
- For hardware systems, the situation is similar. Verilog and VHDL allow designers to describe hardware at a high level and then interconnect multiple modules in a hierarchy to perform an even higher-level function.
- Typically, a single integrated tool suite can handle several different aspects of an HDL's use. This is informally called the "HDL compiler," but an EDA tool suite for design with HDLs really includes many different tools with their own names and purposes.
  - The *text editor* allows you to write, edit, and save an HDL source file. Since the editor is often coupled to the rest of the HDL development system, it often contains HDL-specific features such as recognizing specific filename extensions commonly associated with HDL and recognizing HDL reserved words.
  - The *compiler* is responsible for parsing the HDL source file, finding syntax errors, and figuring out what the model is actually "saying." A typical HDL compiler creates a file in an intermediate, technology-neutral digital-design description language typically called an RTL, or register-transfer language.
  - The *synthesizer*, or *synthesis tool*, targets the RTL design to a specific hardware technology such as the aforementioned ASIC, FPGA, or CPLD. In doing so, it refers to one or more *libraries* containing details of the targeted technology. Synthesis typically has multiple phases, described below.
    - \* The first phase is *mapping* the RTL design into a set of hardware elements that are available in the target technology.
    - \* The second phase is *placement* of the needed elements onto a physical substrate, which is usually a chip layout.

- \* In FPGA and ASIC based designs, the third phase is *routing*, or finding and creating paths between the inputs and outputs of placed elements. In CPLD design, the interconnect is usually fixed and resources were already selected previously based on available connections.
  - The inputs to a *simulator* are the HDL model and a timed sequence of inputs for the hardware that it describes. This input sequence can be contained in or generated algorithmically by another program called a *test bench* usually written in the same HDL. It can also be described graphically using another tool called a *waveform editor*.
- Several other useful programs and utilities may be found in a typical EDA root suite for an HDL, including...
  - A *template generator* creates a text file with the outline of a commonly used HDL structure so that the designer can fill in the blanks to create source code for a very specific purpose.
  - A *schematic viewer* may create a schematic diagram corresponding to an HDL model based on the RTL output of the compiler. Such a schematic is an accurate representation of the function performed by the final synthesized circuit, with a few exceptions, such as if the compiler output hasn't been targeted and optimized.
  - A *chip viewer* lets the designer see how the synthesis tool has physically placed and routed a design on the chip.
  - A *constraints editor* lets the user define instructions and preferences to be used by the synthesizer and other tools as they do their jobs.
  - A *timing analyzer* calculates the delays through some or all of the signal paths in the final chip and produces a report showing the worst-case paths and their respective delays.
  - Finally, a *back annotator* inserts delay clauses or statements into the original HDL source code corresponding to the delays calculated by the timing analyzer.
- It is useful to understand the overall HDL design environment before beginning to learn Verilog itself. There are several steps in an HDL-based design process. Such a process is called the *design flow*. These steps are applicable to any HDL-based design process and are described below.
  - The front-end begins with a functional specification of what needs to be designed, followed by figuring out the basic approach for achieving that function at a block-diagram level. Large logic diagrams, like those for software applications, are hierarchical. Verilog provides a good framework for modeling hardware modules and their interfaces and filling in the details later.
  - Next is actually writing HDL code for the modules, their interface, and their internal details. It is best to use the editor included in the HDL's tool suite for this, as it can provide a large amount of functionality that makes writing the code easier.

- After this, you compile the code. The HDL compiler analyzes the code for syntax errors while also checking for compatibility with imported modules. It will also create the internal information necessary for the simulator to process the design later.
- The next step is simulation, which is just a larger part of a step called verification. Simulation allows you to define and apply inputs to your design and to observe the outputs without needing to build a physical circuit. Verification is verifying that the circuit works as desired given a large amount of test cases. There are two dimensions to verification: Functional verification, where we study the circuit's logical operation independent of timing considerations, and timing verification, where we study the circuit's operation including estimated delays.
- Customarily, you perform thorough functional verification before beginning the back-end steps. Timing verification is often limited before back-end work has begun. After verification, we begin the first stage of back-end development: mapping, or where we convert the RTL description into a set of primitives or components that can be assembled in the target technology.
- In the fitting step, a fitter assigns the mapping primitives or components onto available device resources. The designer can specify additional constraints for this stage, such as the placement of modules within a chip and the pin assignments of external input and output pins.
- The final step is the post-fitting timing verification stage. Only at this stage can actual circuit delays due to wire lengths, electrical loading, and other factors be calculated to reasonable precision.

## 19.4 Verilog Terminology

### 19.4.1 Lecture Slides

- List of terms:
  - EDA - Electronic Design Automation
  - Editor - Where you type your content
  - Compilers - Checks the syntax and other related errors
  - RTL - Stands for Register Transfer Level (or sometimes known as Register Transfer Language). This is a clear description of the interconnections in the design that the HDL interprets.
  - Synthesizer - Creates the corresponding circuit and maps the RTL to hardware design.
  - Simulator - Verifies the design's behavior. The simulator also predicts the electrical and functional behavior of the design without needing to physically create the hardware.
  - Test bench - Verilog's version of test statements or unit tests. These allow us to consider timing impacts and monitor behaviors.
  - Module - Everything we design is located in a module. This is not inclusive of functions, which we will not be working with.

- Netlist - Also known as a port list, this describes what connects in and out of a module.
  - Reserved keywords - A list of terms already existing with another purpose being used by the main program.
- Verilog has four main styles/approaches.
  - CMOS
  - Structural/Gate Level
  - Continuous Assignment/Data Flow
  - Procedural/Behavioral
- The CMOS approach is all about transistor level design. It effectively means the already low gate level is reduced to an even lower level.
- With the gate level or structure level approach (these two terms have identical meanings), you design based around only using logic gates, including the primitives: `and`, `not`, `nor`, `or`, `nand`, `xnor`, `buf`, `bufif0`, `bufif1`, and `notif0`. This will be the main style we focus on.
- With the continuous assignment or data flow approach, outputs are automatically updated as inputs change. The signifier of this method is the `assign` statement.
- The procedural or behavioral method is used to implement sequential execution. This is signified by the `always` statement.
- All styles need a starting point. This is the fundamental encapsulation of the module. We enclose the program we are doing with two keywords to symbolize the start of the module and the end of the module. This helps when things expand to include modular design. The beginning of the module is denoted with `module` and the end with `endmodule` (think `\begin{document}` and `\end{document}` in `LATEX`, respectively).
- A couple of considerations can be made to make the language transition easier.
  - A single line comment is made with the “//” characters. For a multi-line comment, you must use “/\* ... \*/”.
  - Brackets are replaced with `begin` and `end`.
  - Code is only executed sequentially when inside an `always` block.
  - Verilog compilers are just as strict as compilers in other languages.
  - Verilog utilizes the line terminator “;”, similarly to Scala.
- Let’s look at some general syntax.

```
module module_name(port_list);
    // declare what items in the port list are inputs
    // declare what items in the port list are outputs
    // declare what initial variables to be used
    internally
    // body of the program
endmodule
```

- Two general data types to know:
  - A *reg* holds information until it is changed.
  - A *wire* is what is used to make a connection between elements.

### 19.4.2 Assigned Reading

- The basic unit of design and programming in Verilog is a *module*, which is a text file containing declarations and statements. A typical Verilog module may correspond to a single piece of hardware in much the same sense as a module in traditional hardware design, in which the hardware is said to be modeled by a single module or by a collection of modules working together.
- A Verilog module has declarations that describe the names and types of the module's inputs and outputs, as well as local signals, variables, constants, and functions used strictly for internal use in the module. These are not visible outside. The rest of the module contains statements that specify or "model" the operation of the module's outputs and internal signals.
- Verilog statements can specify a module's operation behaviorally. They can also specify the module's operation structurally. In this specific case, the statements define instances of other modules and individual components used, such as gates and flip-flops, while also specifying their interconnections, similarly to a logic diagram.
- Verilog modules can use a mix of behavioral and structural modules and may do so hierarchically. Just as procedures and functions in a high-level programming language can "call" others, Verilog modules can "instantiate" other modules. A higher-level module may use a lower-level module multiple times, and multiple top-level modules can use the same lower-level one.
- The scope of signal, constant, and other definitions remains local to each module. Accordingly, values can be passed between modules only by using declared input and output signals.
- This modular approach gives Verilog a great deal of flexibility in designing large systems. For example, a given module can be designed with a rough behavior model during the initial phase of system design so that overall system operation can be checked. Later, it can be replaced with a more intricate behavioral model for synthesis or perhaps even hand-tuned to achieve higher performance.
- Let's now move on to syntax. A simple example module is shown below. Like other high-level languages, Verilog mostly ignores spaces and line breaks, which can be used for better readability. Short comments can be begun with two slashes(//) anywhere in a line and stop at the line's end. Verilog also allows C-style, multi-line long comments that begin anywhere with /\* and end anywhere with \*/.

```

module VrInhibit(X, Y, Z); // also known as 'BUT-NOT
,
    input X, Y; // as in 'X but not Y'
    output Z; // (see [Klir, 1972])

    assign Z = X & ~Y;
endmodule

```

- Verilog defines many special character strings which are called reserved words or reserved keywords. The above example included a few - **module**, **input**, **output**, **assign**, and **endmodule**.
- User-defined identifiers begin with a letter or underscore and can be contain letters, digits, underscores, and dollar sign (note that identifiers with a dollar sign in the front refer to built-in system functions). Identifiers in the above example include **VrInhibit**, **X**, **Y**, and **Z**. Verilog is case-sensitive for both keywords (lowercase only) and identifiers (**XY**, **Xy**, and **xy** are all different).
- The basic syntax for a Verilog module declaration is shown below. It begins with the keyword **module**, followed by an identifier for the module's name and a list of identifiers for the module's input and output ports. These input and output ports are signals that the modules use to communicate with other modules.

```

module module-name (port-name, port-name, ..., port-
    name);
    input declarations
    output declarations
    inout declarations
    net declarations
    variable declarations
    parameter declarations
    function declarations
    task declarations

    concurrent statements
endmodule

```

- Next comes a set of optional declarations that we describe in this and later sections. These declarations can be made in any order. Besides the declarations shown above, there are a few more not used in this book and therefore are omitted. Concurrent statements follow these declarations and the module ends with the **endmodule** keyword.

- Each port that is named at the beginning of the module in the input/output list must have a corresponding input, output, or inout declaration. The simplest form of these declarations is shown in the first three lines below.

```
input identifier , identifier , ... , identifier ;
output identifier , identifier , ... , identifier ;
inout identifier , identifier , ... , identifier ;
```

```
input [msb:lsb] identifier , identifier , ... ,
        identifier ;
output [msb:lsb] identifier , identifier , ... ,
        identifier ;
inout [msb:lsb] identifier , identifier , ... ,
        identifier ;
```

- The keyword **input**, **output**, and **inout** is followed by a comma-separated list of the identifiers for signals (or ports) of the corresponding type. The keyword specifies the signal direction as follows:
  - **input**: The signal is an input to the module.
  - **output**: The signal is an output the module. Note that such a value cannot be “read” inside the module architecture, only by other modules that use it. A “reg” declaration is needed to make it readable.
  - **inout**: The signal can be used as a module input or output. This mode is typically used for three-state input/output pins.
- An input/output signal declared as described above is one bit wide. Multi-bit or “vector” signals can be declared by including a range specification [**msb** : **lsb**], as shown in the last three lines of the above example. Here, **msb** and **lsb** are integers indicating the starting and ending indexes of the individual bits within a vector of signals. The signals are ordered from left to right, with **msb** giving the index of the leftmost signal. A range can be ascending or descending.

## 20 3/29/2020

### 20.1 MSI

- Recall what combinational logic is and what it means. Combinational logic is any system that is purely about its inputs, with no feedback or memory. Previously, we have explored this with systems that have multiple inputs and outputs.
- In combinational logic, the Level of Integration can vary.
  - SSI is Small Scale Integration, and is limited by the pins on the IC. SSI is normally around 10 gates.
  - MSI is Medium Scale Integration, and is normally under 1,000 gates.
  - LSI is Large Scale Integration and can be thousands of gates.
  - VLSI is Very Large Scale Integration and is composed of millions of transistors.
  - ULSI is Ultra Large Scale Integration is anything over one million transistors. ULSI is normally grouped with VLSI unless the number of transistors is truly ridiculous.
- We will primarily focus on MSI (Medium Scale Integration). MSI is about 20 to 200 gates. Note that an AND gate, as an example, only has four gates, so it is SSI.

### 20.2 Shifting

- Shifting is this idea where we shift values from one position to the next. When we exceed the available positions, the extra values fall off and become zero.
- The most common implementation of this is the barrel shifter. The barrel shifter takes  $n$  data in and  $n$  data out. There is a single control set that controls how far the shift will occur.

### 20.3 Rotating

- Rotating is very similar to shifting in that the contents are moving from one location to another. However, instead of values being replaced by zeroes when they exceed the locations available, we follow the idea of “circular shifting” and have values that go to the other side.

### 20.4 Mux Intro

- Multiplexors (also known as a muxes) are the “grand selectors of logic.” Multiplexors are given a set of inputs and the outcome is selected through an additional set of inputs. The second set of inputs looks a lot like a set of switches.
- At the gate level, this is an MSI element. You must focus on the logic of how an outcome is selected, which comes from logic gates and equations.



- A mux circuit is made up of a few specific elements. Its inputs are composed of enables, selectors, and the data the designer wishes to choose from. Its output is a function. Finally, mux circuits are *only* composed of AND, OR, and NOT gates.
- A demultiplexor is really just a decoder, however you can design a multiplexor using a decoder.

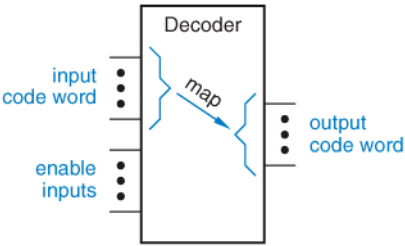
## 21 4/05/2020

### 21.1 Decoders

#### 21.1.1 Lecture Slides

- Decoders are a key MSI device and are critical to computer functionality from processing instructions to memory storage. This is seen a lot with RAM and addressing.
- Decoders, as their name implies, decode encoded information. A decoder has  $n$  data inputs, an enable input, and typically  $2^n$  outputs, only one of which is active at a time.
- Sometimes, it is not optimal to design something that is perpetually operating. This is where the idea of an enable is. An enable acts like a master on/off switch for a component (a collection of gates; MSI or larger). An enable being on is *not* necessarily the same thing as providing power to a chip.

#### 21.1.2 Assigned Reading

- Decoders are designed such that no more than one output is asserted at any given moment. Furthermore, these outputs are asserted only if the corresponding port-select value appears on the input.
- A decoder's input code-word bits are usually called address bits.
- Recall the codes defined in section 5.2, such as 1-out-of-10 code and gray code. This idea of “codes” can be used to define what a decoder is. A decoder is any multiple-input, multiple-output combinational logic circuit that converts or “maps” an input code word with an output code word, where the input and output codes are different.
- With this definition, the general structure of a decoder can be shown, pictured to the right. The enable inputs, if present, must be asserted for the decoder to perform its normal mapping function. Otherwise, the decoder maps all input code words into a single “disabled” output word.
- The most commonly used input code is an  $n$ -bit binary code, where an  $n$ -bit word represents one of  $2^n$  different coded values. Sometimes, an  $n$ -bit binary code is truncated to represent fewer than  $2^n$  values.
- The most commonly used output code is a 1-out-of- $n$  code, which has  $n$  bits, one of which is asserted at any time. Note that  $n$  need not be a power of 2, but usually is.
- The most common decoder circuit is an  $n$ -to- $2^n$  binary decoder, which has an  $n$ -bit binary input code and a 1-out-of- $2^n$  output code. Such a decoder

is used when you need to activate exactly one of the  $2^n$  outputs based on an  $n$ -bit input value.

- The binary decoder's truth table introduces the idea of a "don't-care" notation for input combinations. If one or more input values do not affect the output values for some combination of the remaining inputs, they are marked with an "x" for that input combination. The "x" is used to denote "don't care."
- The more inputs you have, the wider the AND gate will be necessary to complete a decoder. Typically, wide AND gates cannot be realized in a single "level" of transistors. Larger decoders can be designed by cascading multiple smaller decoders. For example, multiple 3-to-8 decoders can be cascaded to create a 5-to-32 decoder.
- In ASICs and custom VLSI, it is often necessary to build decoders with an even larger number of inputs and outputs. These applications use a method called predecoding to accomplish this. Predecoding structures are designed to minimize circuit delays, which can not only be affected by the number of gates in a signal path but by the number of gate inputs connected to each output.
- Decoders can be customized numerous different manners. In HDL-based design, such customization is normally done in the context of a larger module design where decoding functionality is included among other things. Customizations are easy to do and include things such as having a different number of input and data outputs, using active-low inputs (especially enables) or outputs, accessing an output for two or more address-input combinations, or accessing multiple outputs for a single input combination.
- A seven-segment display is a type of display that uses LED or LCD elements to display decimal data. A seven-segment decoder has 4-bit BCD as its input code and the "seven segment code" as its output code. This is the best example of a decoder that is not a binary decoder.
- Previously, we defined a decoder to be any multiple-input, multiple-output combinational logic circuit that converts an input code word into an output code word in a different code. By this definition, a circuit that does the opposite would also be a binary decoder, but these are usually just called binary encoders. A standard binary encoder's output is meaningful if and only if exactly one input is asserted.

## 21.2 Encoders

- Encoders are an MSI device that allows you to input a lot of information and store it into less bits. Typically, these are in a  $2^n$  to  $n$  design.

## 21.3 Continuous Assignment

### 21.3.1 Lecture Slides

- With the Continuous/Dataflow-style of Verilog, outputs are updated as the inputs change. The signifier of this is the “**assign**” statement.
- The operators used in Continuous Assignment are shown below.
  - Bitwise symbols treat each bit separately.
    - \* NOT: ~
    - \* AND: &
    - \* OR: |
    - \* XOR: ^
    - \* XNOR: ^^or ^^
  - Logical symbols are evaluated from left to right and stop as soon as the result is known.
    - \* NOT: !
    - \* AND: &&
    - \* OR: ||
- Ultimately, the decision to use bitwise or logical depends on what you are doing. When working with 1-bit wide data, however, the choice doesn’t really matter.
- When using parentheses, remember that precedence matters. When in doubt, add parentheses.
- Concatenation is where you place the variables in the order you wish the bits to map in and where the overflow will go.

### 21.3.2 Assigned Reading

- If Verilog exclusively had instance statements, it wouldn’t be much more than a hierarchical net-list description language. “Continuous-assignment statements” allow Verilog to model a combination circuit in terms of the flow of data and operations in the circuit. This style is called a dataflow model or description.
- Dataflow models use continuous-assignment statements, with the basic syntax being shown below. The keyword **assign** is followed by the name of a net, then an = sign, and finally an expression giving the value of the assigned. The left-hand side of the statement can also specify a bit or part of a net vector, or even a concatenation using the standard concatenation syntax.

```
assign net-name = expression;  
assign net-name[bit-index] = expression;  
assign net-name[msb:lsb] = expression;  
assign net-concatenation = expression;
```

- A continuous-assignment statement continuously evaluates the value of its right-hand side and assigns it to the left-hand side. In simulation, this assignment occurs in zero simulated time unless a delay is specified.
- As with instance statements, the order of continuous assignment statements doesn't matter. If the last statement changes a net value used by the first statement, then the simulator will go back to the first statement and update its results according to the net that was just changed. It is therefore possible for a module to perpetually loop until it times out if it has multiple contradicting assignment statements.
- Below is code for a prime-number detector circuit in dataflow style.

```

module Vrprimed (N, F);
input [3:0] N;
output F;

assign F = (~N[3] & N[0]) | (~N[3] & ~N[2] & N[1]) |
           (~N[2] & N[1] & N[0]) | (N[2] & ~N[1] & N[0]);
endmodule

```

- Verilog's continuous-assignment statements are unconditional, but different values can be assigned if the righthand operator uses the conditional operator (?). The below code approaches the same prime-number detection circuit in a completely different manner with this method.

```

module Vrprimec (N, F);
input [3:0] N;
output F;

assign F = N[3] ? (N[0] & (N[1]^N[2])) : (N[0] | (~N
           [2]&N[1]));
endmodule

```

- A dataflow-style example with a more natural and intuitive use of this conditional operator is shown below. This module transfers one of three input bytes to its output depending on which of three corresponding select inputs are asserted. The order of the nested conditional operations determines which byte is transferred if multiple selected inputs are asserted, with A having the highest priority and C having the lowest. If no select input is asserted, the output is 0.

```

module Vrbytesel (A, B, C, selA, selB, selC, Z);
input [7:0] A, B, C;
input selA, selB, selC;
output [7:0] Z;

assign Z = selA ? A : (
           selB ? B : (
           selC ? C : 8'b0));
endmodule

```

## 21.4 Behavioral Verilog

### 21.4.1 Lecture Slides

- Behavioral Verilog is also known as “Procedural Style.” On top of guaranteeing sequential execution, Behavioral Verilog also allows for the use of conditionals.
- Behavioral Verilog is denoted by the **always** operator, followed by either the sequentially executed code or a sensitivity block. The “procedural statements” or stuff inside this always block is written in a style similar to C. “begin” and “end” are used to replace “{” and “}.”
- An example of a behavioral Verilog block is shown below.

```
begin
    procedural-statement
    ...
    procedural-statement
end
```

```
begin: block-name
    variable declarations
    parameter declarations
    procedural-statement
    ...
    procedural-statement
end
```

- There are many styles of behavioral style syntax, some of which are shown below.

```
always @ (signal-name or signal-name or ... or signal-name)
    procedural statement
always @ (signal-name, signal-name, ..., signal-name)
    procedural-statement
always @ (*) procedural-statement
```

```
always @ (posedge signal-name) procedural-statement
always @ (negedge signal-name) procedural-statement
```

```
always procedural-statement
```

- Information provided in the parentheses are the circumstances under which the code will execute. These work similarly to if-statements. These conditions are called sensitivities.
- Variables need to be declared before use. They need to be declared as a **reg** since they hold the same value until they are explicitly changed. Afterwards, you can use an assignment statement, either blocking or non-blocking.

- Let's go over the reg data-type next. A reg is non-volatile, holding the value contained inside until explicitly changed. These are exclusively used in behavioral Verilog, and are used in a similar way to how we used wires for declaring.
- In behavioral Verilog, we have two main styles of execution - blocking and non-blocking.
  - Blocking is the traditional manner of execution, updating the values as you go. For example, the code `x = 5;` is an example of a blocking style of execution.
  - Non-blocking, on the other hand, does all of the calculations immediately for the variable but does not make the assignment to the variable until the very end of the block. `x <= 15;` is an example of this non-blocking style.

#### 21.4.2 Assigned Reading

- The key element of Verilog behavioral modeling is the **always** statement, with the syntax options shown below. This **always** statement is followed by one or more procedural statements, which will be explained shortly. While this syntax table only shows a singular procedural statement, one type of procedural statement is the **begin – end** block that encloses a list of other procedural statements.

```
always @ (signal-name or signal-name or ... or signal-name)
```

```
    procedural statement
```

```
always @ (signal-name, signal-name, ..., signal-name)
```

```
    procedural-statement
```

```
always @ (*) procedural-statement
```

```
always @ (posedge signal-name) procedural-statement
```

```
always @ (negedge signal-name) procedural-statement
```

```
always procedural-statement
```

- Procedural statements in an **always** block execute code sequentially, similarly to software programming languages. However, the **always** block itself executes concurrently with other concurrent statements in the same module (instance, continuous-assignment, and **always**).
- In the first three forms shown above, the @ sign is followed by a parenthesized list of signal names, which is called a sensitivity list. This specifies all of the signals whose values may affect results in an **always** block. The first two forms function identically. The third form of sensitivity list (\*) is shorthand for “every signal that might change a result” and puts the burden on the compiler to determine which signals should be in the list. The fourth and fifth forms are used in sequential circuits. Finally, the last form doesn't specify a sensitivity list. Such an **always** begins running at time zero and indefinitely loops.

- An instance or continuous-assignment statement also has a sensitivity list, albeit an implicit one. All of the input signals in an instantiated component or module are on the instance statement's implicit sensitivity list. Similarly, all of the signals on the right-hand side of a continuous-assignment statement are on its implicit sensitivity list.
- In simulation, a Verilog concurrent statement such as an **always** block is always either executing or suspended. A concurrent statement is initially suspended. Whenever any signal in its sensitivity list changes value, it resumes execution. A resumed **always** block starts with the first procedural statement and continues executing them sequentially until reaching its end. If any signal in the sensitivity list changes value as a result of executing the concurrent statement, it re-executes. This loop continues until the statement executes without any signals changing value.
- Verilog has several different kinds of procedural statements used within an **always** block. They are: assignment, **begin** – **end** blocks, **if**, **case**, **while**, and **repeat**.
- Procedural statements are written in a style similar to programming languages like C. Every value assigned to a variable is preserved until it is changed in the current or subsequent execution of an **always** block.
- The first two procedural statements we will go over are blocking and non-blocking assignment, with the syntax shown below.

```
variable-name = expression;    // blocking assignment

variable-name <= expression;   // nonblocking
                               assignment
```

The lefthand side of either procedural assignment statement must be a variable, but the righthand side can be any expression that produces a compatible value and can include both nets and variables.

- A blocking statement looks and acts like an assignment statement in any other procedural language, like C. A nonblocking assignment looks and acts a little differently. It evaluates its left side immediately but doesn't assign the resulting value to the lefthand side until an infinitesimal delay after the entire **always** block has executed. Thus, the "old" value of the lefthand side continues to be available for the remainder of the **always** block. You can read a nonblocking assignment as "*variable-name* eventually gets *expression*."
- So when should you use one over the other? Always use blocking assignments in an **always** block intended to create combinational logic. Always use nonblocking assignments in **always** blocks intended to create sequential logic. Do not mix the two assignments in the same **always** block, and do not make assignments to the same variable in two different **always** blocks.
- Below is the basic syntax for a **begin** – **end** block, which is simply a list of one or more procedural statements enclosed by the keywords **begin** and **end**.



```

begin
    procedural-statement
    ...
    procedural-statement
end

begin: block-name
    variable declarations
    parameter declarations
    procedural-statement
    ...
    procedural-statement
end

```

A **begin** – **end** block can have its own local parameters or variables. If it does, the block *must* be named to track these items during simulation or synthesis. A **begin** – **end** block can be named even if it doesn’t have any local parameters or variables.

- Note that the procedural statements contained within a **begin** – **end** block execute sequentially, not concurrently like the previously covered instance, continuous-assignment, and **always** statements.
- Other procedural statements beyond simple assignment and **begin** – **end** blocks give designers much more powerful ways to model the behavior of a circuit. The most familiar of these is probably the **if** statement. In the first and simplest form of this statement, a condition is tested. If this condition is true, a procedural statement is executed.
- In the **if** – **else** form, we add the “else” clause with another procedural statement if the above condition evaluates to anything not true. Although an **if** – **else** statement can contain two semicolons, it is still just a single statement, an important fact to note. Thus, an **if** – **else** statement can be used anywhere that a single statement can be used. The syntax for an **if** and an **if** – **else** statement is shown below.

```

if ( condition ) procedural-statement

if ( condition ) procedural-statement
else procedural-statement

```

As in other languages, an **if** or an **if** – **else** statement can be nested.

- When two or more variables must be tested to determine different outcomes, a nested series of **if** statements is generally the correct approach. However, if all of the **if** statements would be testing the same variable, it is often more clear to use a **case** statement, which we will cover next.
- Below is the syntax for a **case** statement. It begins with the **case** keyword and a parenthesized “selection expression,” usually one that evaluates to a bit-vector value of a certain width. Next is a series of case items, each of which has a comma-separated list of “choices” and a procedural statement.

If there is only one choice, the comma is omitted. A single “default” case item may be included. The statement ends with the **endcase** keyword.

```
case (selection-expression)
    choice , ... , choice: procedural-statement
    ...
    choice , ... , choice: procedural-statement
    default: procedural-statement
endcase
```

- The operation of a **case** statement is simple. It evaluates the selection expression, finds the first one of the choices that matches the expression’s value, and then executes the corresponding procedural statement. Once again, the **case** statement executes just one procedural statement, corresponding to the first match.
- Choices in a **case** statement are usually just constant values compatible with the selection expression, but they can be more complex expressions. If some of the choices overlap, it is important to understand that only the *first* matching choice is executed. When the choices do not overlap, they are said to be “mutually exclusive.” Standard Verilog coding practice is to avoid overlapping choices.
- Often, the listed choices in a **case** statement are not “all-inclusive,” or don’t include all of the possible values of the selection expression. In these scenarios, the **default** keyword can be used in the last case item to denote all selection values that have not been covered. Even if you believe your listed choices are all-inclusive, it is standard practice to still include a **default** case. A **case** statement that is all-inclusive is called a full case. Standard Verilog practice avoids using nonfull cases.
- Another important class of procedural statements are looping statements. The most commonly used of these is the **for** statement or the **for** loop, with the syntax shown below. Here, the loop-index is a register variable, typically an integer or a bit vector. first-expr is an expression giving a value that is assigned to loop-index when the **for** loop begins execution.

```
for (loop-index = first-expr; logical-expression;
    loop-index = next-expr)
    procedural-statement
```

```
for (loop-index = first; loop-index <= last; loop-
    index = loop-index + 1;)
    procedural-statement
```

After loop-index is initialized, the **for** loop begins executing the enclosed procedural-statement for a certain number of iterations. At the beginning of each iteration, it evaluates logical-expression. If the value is false, the **for** loop stops execution. If it is true, it executes procedural-statement and at the end of the iteration it assigns next-expr to loop-index. Iterations continue until logical-expression is false.

- The remaining Verilog looping statements are **repeat**, **while**, and **forever**, with syntax shown below.

```
repeat (integer-expression)
    procedural-statement
```

```
while (logical-expression)
    procedural-statement
```

```
forever
    procedural-statement
```

A **repeat** statement repeats a procedural-statement a number of times given by integer-expression. A **while** statement repeats a procedural-statement until logical-expression is false. Lastly, a **forever** statement repeats “forever.”

## 21.5 Test Benches

### 21.5.1 Lecture Slides

- Test benches are the Verilog equivalent to test statements or j-unit tests. These allow us to consider timing impacts and monitor behaviors, on top of the ability to see if the program runs.
- A test bench is composed of multiple key components, listed below.
  - Compiler directive to declare the time unit (optional)
  - Variable declarations
  - Function call
  - Variable initializations
  - Evaluation pattern
  - Analysis tools: waveform file, log display, file output
  - Complete runtime
- There are many different compiler directives, distinguished by the “`” mark. **timescale** allows for declaration of the specific units of time and the observed precision.
- When naming a test bench module, you take the module being tested and add “\_tb.” For example, a test bench for **fav1** would be called **fav1\_tb**. Test bench modules do not contain a port list. The variables used as inputs are declared as **reg** values and the values being tested are declared as wires.
- Calling the tested module is just the instantiation. We can use the **..()** form to get away from knowing the portlist order so long as we know the variable names, which should be in the documentation anyways.
- When testing, we need a starting point, because we need values to start or nothing exists and nothing can be executed. We use the **initial** keyword to denote procedural-statements that only need to be run once.

- When testing, run through the cases you want to test. `._#` is the number of time units you want to pass before executing a line.
- Since we are not using an IDE, we need a file that can be read by a waveform reader. `$dumpfile(_)` designates a file for the destination. `$dumpvars` then puts the data in.
- We can also use the log/console to display information as to what is currently going on. `$display` creates text that happens once. `$monitor` creates outputs anytime something changes. The syntax here is exactly like C.
- The difference between `$write` and `$display` is whether or not a new line is printed after the statement completes.
- When determining how long the tests should run, run them as long as necessary. Use `#_` to tell the program how long you wish for a test to run. The `always` block will repeat until the amount of time units are reached.

### 21.5.2 Assigned Reading

- A test bench specifies a sequence of inputs that can be applied by the simulator to an HDL model, such as a Verilog module. The entity being tested is often called the unit under test, or the UUT.
- Typically, Verilog test benches use the `initial` block, the syntax for which is shown below. Similarly to an `always` block, it contains one or more procedural statements but lacks a sensitivity list.

```
initial
    procedural-statement

initial begin
    procedural-statement
    ...
    procedural-statement
end
```

An `initial` block executes just once at the simulated time zero. Just like in an `always` block, the `begin – end` block can be named and have its own local variable and parameter declarations.

- Normally, it is worth expending a little more effort when writing test benches to make their output more user friendly. For example, `$write` and `$display` can be used to print a result to the system console.

## 22 4/12/2020

### 22.1 What Is Sequential Logic?

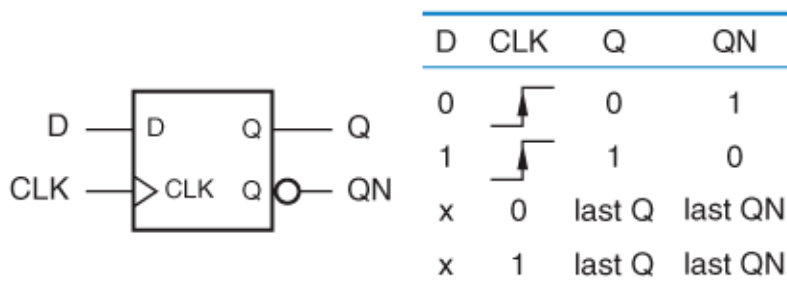
#### 22.1.1 Lecture Slides

- Some key terms:
  - State Memory - These will have  $n$  flip-flops or latches in these systems.
  - Tick - A clock update.
  - Next state logic - The inputs to the flip-flops or latches. These are also the function of the current state and the output.
  - Output logic - The function output.
- Present state is the information currently in memory. Next state is the information we are going to update to. In this class, we denote next state as  $Q^*$  and present state as  $Q$ . The main idea is to associate the memory input and output.
- We can't design these systems without understanding the hardware used to implement it. These are the basics of this theory, and are actually based on combinational logic.

#### 22.1.2 Assigned Reading

- “The state of a sequential circuit is a collection of state variables whose values at any one time contain all the information about the past necessary to account for the circuit's future behavior.”
- In a digital circuit, state variables are binary values corresponding to certain logic signals in the circuit. A circuit with  $n$  binary state variables has  $2^n$  possible states. While  $2^n$  can indeed get rather large, it is important to understand that *finite* value, never infinite. For this reason, sequential circuits are sometimes called finite-state machines (or FSMs) or more often simply state machines.
- State variables need not have any direct physical significance. Furthermore, there are an unlimited number of ways to choose them to describe a particular sequential circuit, each making sense in some particular context.
- Why do state changes happen? In most sequential circuits, they can occur only at times specified by a free-running clock signal. By convention, a clock signal is active high if state changes occur at the clock's rising edge (transitioning from LOW to HIGH) and active low if they occur at the falling edge. The edge at which state changes occur can be called the active or triggering edge.
- The clock period is the time between successive transitions in the same direction and the clock frequency is the reciprocal of the clock period.
- The triggering edge is often called a clock tick.

- The duty cycle is the percentage of time that the clock signal is at its asserted level (i.e. HIGH for an active-high clock). State changes only occur at the triggering clock edge. Between the triggering states, the state is stable.
- Most sequential circuits (along with almost all state machines) use a particular type of element to store their state elements, specifically an edge-triggered D flip-flop. Below is the logic symbol for a positive-edge-triggered D flip-flop, along with its function table. The circuit's inputs are D and CLK and its outputs are Q and the optional QN, with QN being the complement of Q. The outputs may change only at the rising, or positive, edge of the controlling CLK signal. When the CLK signal transitions from LOW to HIGH, the circuit samples its D input and places the current value of D on the Q output, simultaneously placing the complement of that value on QN if present. Between LOW-to-HIGH clock transitions, the flip-flop maintains the value previously stored in Q (and QN).



- There are other types of sequential circuits besides this D flip-flop. A feedback sequential circuit, for example, uses ordinary gates and feedback loops to obtain memory in a logic circuit resulting in the creation of sequential-circuit building blocks such as D flip-flops.

## 22.2 Latches

- Every device has a table used to describe its behavior. These are often referred to as the Characteristic Table or the Next State Table since they describe the characteristics or the behavior of the device. If we have a table, we can form an equation, a fact that will be useful later.
- The first sequential logic element we will cover is the S-R Latch, with the name coming from the idea of set-reset. An S-R Latch is constructed from two NOR gates.
- In some cases, we may not have a “last” Q or QN. This is called a “complication.” In most cases, the bitstable idea of a random assignment that allows for some form of a default start.
- Most sequential logic devices natively output its normal output and the output's complement (Q and QN, respectively). When designing with these devices, use QN over a NOT gate on the signal for efficiency.

- Q and QN are common for present and next states, but there are many variations of these terms. One that we use is adding the \* after the variable name.
- The  $\bar{S}$ - $\bar{R}$  Latch uses NAND gates instead of NOR gates. This is also an active low system, so to *set* we need to have a zero. Just know that this exists.
- Another sequential logic device is D. D is often considered as data. D only has one input. Notably, this is the first time we extend our sequential block by adding more gates to configure.
- Depending on the material, ENABLE or CONTROL can also be referred to as G. When G is high with active high design, your latch is on. This is called open or transparent. When G switches off, it holds its value.

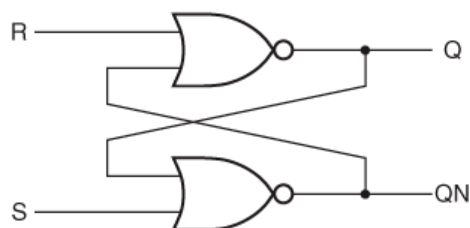
## 22.3 Flip-Flops

### 22.3.1 Lecture Slides

- Latches and flip-flops are both core devices. The main difference between the two is that with a latch, the outputs change with the inputs, while flip-flops need to explicitly be triggered.
- To produce a D flip-flop, we use two D latches in a master-slave configuration, a method that exploits time to properly function.
- To know which side is posedge or negedge, look at the name or the schematic symbol.

### 22.3.2 Assigned Reading

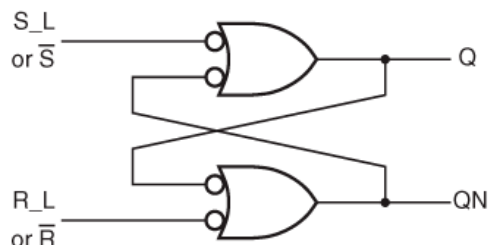
- Latches and flip-flops are the most basic building blocks for most sequential circuits. Latches and flip-flops in typical digital systems are functionally specified devices prepackaged in a standard integrated circuit.
- All digital designers use the name flip-flop for a sequential device that normally samples its inputs and changes its outputs only when a clocking signal is changing. Conversely, most digital designers use the name latch for a sequential device that continuously watches its inputs and changes its outputs at any time.
- The simplest sequential circuit that has control inputs can be built from just two NOR gates, as shown to below. This is called an S-R (set reset) latch. The circuit has two inputs, S and R, and two outputs, Q and QN. As noted previously, QN is the complement of Q. QN is sometimes labeled as  $\bar{Q}$  or Q.L. These S-R latches are used to detect events using the S input to “set” the latch when the event occurs and using R to “reset” it later.



S	R	Q	QN
0	0	last Q	last QN
0	1	0	1
1	0	1	0
1	1	0	0

Observe the S-R latch's function table shown above. When S and R are both zero, the circuit behaves like a bi-stable element. A result of this is that we make a feedback loop retaining one of two logic states:  $Q = 0$  or  $Q = 1$ . Either S or R can be asserted to force the feedback loop to the desired state. S *sets* or *presets* the Q output to 1 while R *resets* or *clears* it to 0.

- The propagation delay is the time it takes for a transition on an input signal to produce a transition on an output signal. A given latch or flip-flop may have several different propagation-delay specifications, one for each pair of input and output signals. Additionally, the propagation delay may be different depending on whether the output makes a LOW-to-HIGH or HIGH-to-LOW transition.
- An  $\bar{S}$ - $\bar{R}$  latch (read as “S-bar-R-bar latch”) has active-low set and rest inputs built from NAND gates, as shown in the below figure.  $\bar{S}$ - $\bar{R}$  latches are used more often than S-R latches because NAND gates are preferred over NOR gates for reasons of size, speed, or maybe even both.

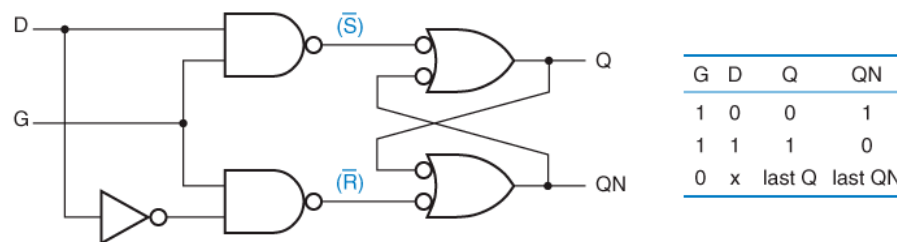


S_L	R_L	Q	QN
0	0	1	1
0	1	1	0
1	0	0	1
1	1	last Q	last QN

As shown by the function table, the operations of an  $\bar{S}$ - $\bar{R}$  latch are similar to an S-R latch, with only two key differences. First,  $\bar{S}$  and  $\bar{R}$  are active low, so the latch only remembers the previous state when  $\bar{S} = \bar{R} = 1$ . Second, when both  $\bar{S}$  and  $\bar{R}$  are asserted simultaneously, both latch outputs got to 1, versus 0 like in an S-R latch. Besides these two differences, an  $\bar{S}$ - $\bar{R}$  latch performs identically to an S-R latch.

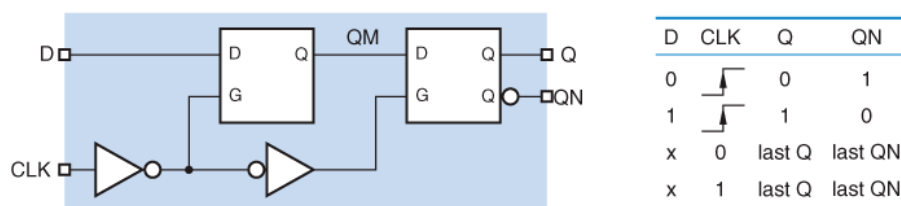
- While S-R latches are useful in control applications, we often need latches to simply store bits of information. In these cases, a D latch may be used. The figure for a D latch is shown below.





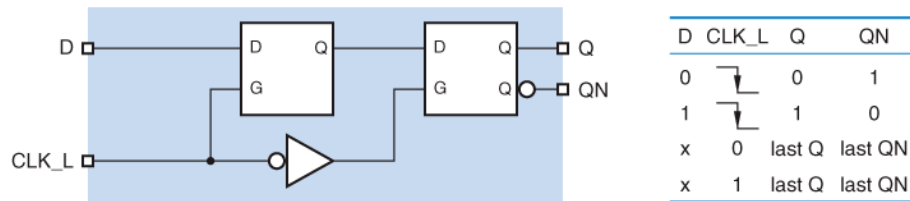
The right-hand side of the logic diagram is just an  $\bar{S}\text{-}\bar{R}$  latch. Two additional NAND gates are inserted on the left-hand side to ensure that either  $\bar{S}$  or  $\bar{R}$  is asserted when the control input  $G$  is asserted, eliminating the scenario of a normal S-R latch when both S and R are asserted simultaneously.  $G$  is sometimes also called ENABLE, CLK, or C. Additionally note that  $G$  is sometimes active-low in some D latch designs.

- Let's go over an example of D latch behavior. When the  $G$  input is asserted, the  $Q$  output follows the input. In this situation, the latch is said to be "open" and the resulting view from the  $Q$  output to the  $D$  input is "transparent." As a result, this type of circuit is often called a transparent latch. When the  $G$  input is negated, the latch "closes." When this happens, the  $Q$  output retains its last value and no longer changes in response to  $D$  as long as  $G$  remains negated.
- Previously, we introduced the positive-edge-triggered D flip-flop as the most commonly used sequential element for storing state variables of a state machine. A D flip-flop doesn't necessarily need to be a part of a formal state machine - it can also be used to simply store data. It is distinguished from a D latch by its edge-triggered behavior.
- Below is a figure showing how a positive-triggered D flip-flop can be built from a pair of D latches. The first latch is called the master, opening and following the inputs when CLK is 0. When CLK goes to 1, the master latch closes and its output is transferred to the second latch, called the slave. The slave latch is open all the while that CLK is 1 but changes only at the beginning of this interval.



The triangle on the D flip-flop's CLK input indicates edge-triggered behavior and is called a dynamic-input indicator.

- A negative-edge-triggered D flip-flop simply inverts the clock input so that all of the action takes place on the falling edge of CLK.L. By convention, a falling-edge trigger is considered to be active low. This flip-flop is shown below.



- Some D flip-flops have asynchronous inputs that may be used to force the flip-flop to a particular state independent of D and CLK inputs. These inputs are typically labeled PR (preset) and CLR (clear) and behave like the set and reset inputs on an S-R latch.
- A commonly desired function in D flip-flops is the ability to hold the last value stored rather than load a new value at the clock edge. This can be accomplished by adding an enable input, called EN or CE (clock enable).
- A T (toggle) flip-flop changes state on every tick of the clock. These are most frequently used in counters and frequency dividers. In many applications of T flip-flops, the flip-flop need not be toggled on every clock tick. Such applications can use a T flip-flop with enable, in which the flip-flop changes state at the triggering edge of the clock only if the enable signal is asserted.

## 22.4 Sequential MSI Devices

### 22.4.1 Lecture Slides

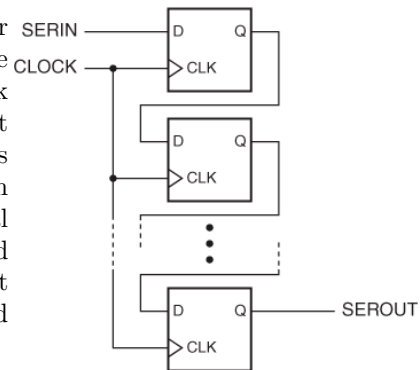
- A sequential logic device is a logic device made into an ASIC implementation. These must have memory and/or storage.
- A register is “a collection of two or more D flip-flops with a common clock input.” Registers are essentially our barebones storage device. The amount of information a register can store is dependent on the number of D flip-flops it is composed of, with the register storing  $n$  bits for  $n$  total flip-flops.
- A shift register is a design that allows for conversion between serial and parallel inputs such that data can be worked with, such as with arithmetic or addressing.

### 22.4.2 Assigned Reading

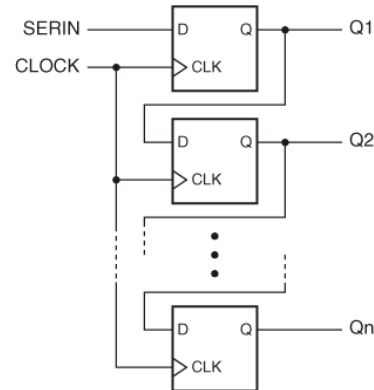
- A counter is generally any clocked sequential circuit whose state diagram contains a single cycle. The modulus of a counter is the number of states in the cycle. A counter with  $m$  states is called a modulo- $m$  counter or sometimes a divide-by- $m$  counter.
- The most commonly used counter type is an  $n$ -bit binary counter, with such a counter having  $n$  flip-flops and  $2^n$  states, visited in the sequence  $0, 1, 2, \dots, 2^n - 1, 0, 1, \dots$ . Each of these states is encoded as the corresponding  $n$ -bit binary integer.

- An  $n$ -bit binary counter can be constructed with just  $n$  flip-flops and no other components for any value  $n$ . Because T flip-flops only change state on every rising edge of their clock cycle, this counter toggles if and only if the immediately preceding bit changes from 1 to 0. This counter is called a ripple counter because the carry information ripples from the less significant bits to the more significant bits, one bit at a time.
- Ripple counters require significantly fewer components than other types of binary counters, but at a cost, being significantly slower as well. Because of this, ripple counters are rarely used in actual practice, with the rare exception such as a digital watch.
- A synchronous counter connects all of its flip-flop clock inputs to a common CLK signal allowing all flip-flop outputs to simultaneously change. This can be done using T flip-flops with enable inputs.
- It is additionally possible to provide a master counter-enable signal called CNTEN. Each T flip-flop toggles if and only if CNTEN is asserted and all of the lower-order counter bits are 1. This is sometimes called a synchronous serial counter because the combinational enable signals propagate serially from the least significant to most significant bits.
- The fastest binary counter structure is the synchronous parallel counter, created by driving each EN input from a synchronous serial counter with a dedicated AND gate.
- Although most counters are designed with enable inputs, counters are often used in a free-running mode in which they are enabled continuously.
- A binary counter can be combined with a decoder to obtain a set of 1-out-of- $m$ -coded signals, where one signal is asserted in each counter state. This is particularly useful when counters are used to control a set of devices where a different device is enabled in each counter state. Through this approach, each output of the decoder enables a different device.
- Occasionally, some decoder outputs can contain “glitches” on state transitions when two or more counter bits change despite the counter outputs being glitch free themselves and the decoder outputs not having any static hazards. This is because in synchronous counters, the outputs don’t change at exactly the same time. More specifically, the different signal paths in a decoder can have different delays. Thus, even if a decoder input changes simultaneously in theory, in practice it may be a multi-step process. This is an example of a function hazard.
- In most applications, the decoder output signals would be used as function inputs in devices that would sample the inputs on a clock edge (devices such as a register or a counter). In these cases, the glitches aren’t necessarily a problem if all devices use the same clock signal since the glitches occur after the clock edge. However, if these glitches were applied to an asynchronous control input, such as the inputs of an  $\bar{S}$ - $\bar{R}$  latch, they *would* be a problem.

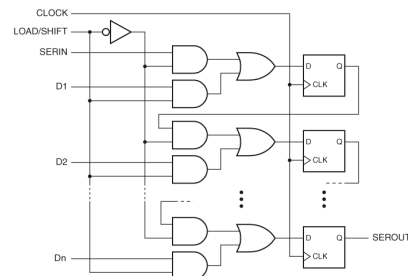
- A shift register is an  $n$ -bit structure register with a provision for shifting its stored data by one bit position at the end of every clock tick. To the right is the structure of a shift register. The serial input, SERIN, specifies a new bit to be shifted into one end at each clock tick. This bit will appear at the serial output, SEROUT, after  $n$  clock ticks and is lost one tick later. Subsequently, an  $n$ -bit serial-in serial-out shift register can be used to delay a signal by  $n$  clock ticks.



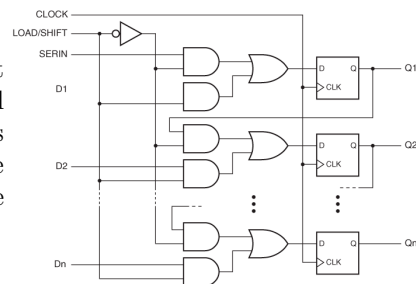
- This idea of a shift register can be further expanded upon. A serial-in, parallel-out serial register, shown to the right, has outputs for all of its stored bits. This allows its outputs to be made available to all other circuits. Such a shift register can be used to perform serial-to-parallel conversion.



- A parallel-in, serial-out shift register is shown to the right. Here, at each clock tick the register loads new data from its inputs or shifts its current contents, depending on the current value of the LOAD/SHIFT control input. This is achieved by internally utilizing a 2-input multiplexor on each flip-flop's D input.



- Lastly, there is the parallel-in, parallel-out serial register. This is an extremely general register applicable to any of the applications described with previous registers. These are formed by providing outputs for all of the stored bits in a parallel-in register.



- All of the shift registers described thus far have been unidirectional shift registers because they shift in only a single direction. A bidirectional shift register, as the name implies, has the capability to shift in either direction depending on the value of a control input.
- While serial/parallel conversion is a “data” application, shift registers have

“non-data” applications as well. A shift register can be combined with a combinational logic circuit to form a state machine whose state diagram is cyclic. Such a circuit is called a shift-register counter. In contrast to a normal binary counter, shift register counters do not count in an ascending or descending binary sequence. We will now go over different approaches to counting sequences with a shift-register counter.

- The most simple shift-register counter uses an  $n$ -bit shift register to obtain a counter with  $n$  states and is called a ring counter. Ring counter’s have a very major problem, however, in their lack of robustness. If a single 1 output is lost due to a temporary hardware problem, the counter goes to state 0000 indefinitely. Likewise, an extra 1 output will result in the counter producing an indefinite incorrect counter cycle.
- The problems of a ring counter are resolved with a self-correcting counter, which is specifically designed such that all abnormal states have transitions leading into normal states. These are desirable for the same reasons minimal-risk approaches are recommended, in that if something unexpected happens a counter or state machine should go to a “safe” state.
- The major appeal of a ring counter for control applications is that its states appear in 1-out-of- $n$  decoded form directly from the flip-flop’s outputs. In other words, one and only one flip-flop output is asserted per state. Furthermore, these outputs are effectively glitch-free.
- An  $n$ -bit shift register with the complement of the serial output fed back into the serial input is a counter with  $2n$  states and is called multiple things, such as a twisted-ring, a Moebius, or a Johnson counter. The decoded outputs of a Johnson counter are glitch free. Because an  $n$ -bit Johnson counter has  $2^n - 2n$  abnormal states, these suffer from the same robustness problems as a ring counter.
- Up until now, we have only seen shift registers with less than the maximum of  $2^n$  normal states. An  $n$ -bit linear feedback shift-register (LFSR) counter can have  $2^n - 1$  states, almost reaching the maximum. Such a counter is also sometimes called a maximum-length sequence generator. The design of these LFSR counters is based on the theory of finite fields.
- Using the finite-field theory, it can be shown that for any value of  $n$ , there exists at least one feedback equation that makes the counter go through all  $2^n - 1$  nonzero states before repeating. This is called a maximum-length sequence.
- The states of an LFSR counter are not visited in binary counting order. Typically, LFSR counters are used where this characteristic is an advantage, such as generating test inputs for logic circuits. Designers capitalize on the “psuedo-randomness” of LFSR circuits, which are more likely to detect errors than a normal binary counter.

## 23 4/19/2020

### 23.1 What Is an FSM?

- State machines are known as Finite State Machines in this course. These are additionally called Finite State Automata in computer science theory. These are called finite because we work with a finite number of states.
- FSMs represent a system of behaviors with states and paths. At any given state, all inputs must be considered. With a few exceptions, you describe these states with names representing a majority of the processes working.
- There are two types of FSM design styles: Mealy, where the output also depends on the input logic, and Moore (what we will focus), where the output logic depends solely on its present state.

### 23.2 FSM Design Algorithm

- An algorithm is just a recipe for a system. There are many steps of the process, listed below.
  - Figuring out the problem
  - State diagram
  - State table
  - State assignments
  - Transition table
  - Transition/Next state equations
  - Output equations
  - Logic diagrams
  - Logic schematic
  - Verilog
- A state diagram is a visual representation of the relationships between the states, inputs, and outputs, with the location of the output being based on if it is Moore or Mealy and with states being represented with circles. Transitions are drawn with arrows between the states and labeled based on whether or not the input is to make that transition. Lastly, the output label is dependent on the FSM's type. If it is Moore, the output is labeled in the circle with the state name. If it is Mealy, it is labeled on the arrow with the transition.
- The state table represents the paths between the current state and the next state based on the input. This table has a section for the output as well. States are represented in assignment form (letters). The form (of two) being drawn in this class has three segments: present state, next state, and output. Next state and output have multiple columns to represent each input combination, with each column itself containing a column for the state variables and a column for the output variables. Present state simply has a number of columns corresponding to the number of state variables.

- State assignment is when we take state letters and change them to numeric values (0 and 1) that we can use to develop equations. There are a few types that are common, including binary, gray code, and one hot. Gray code is most useful when simplifying a K-Map and One Hot is useful for determining functions by inspection (think decoder logic). This step is also known as encoding.
- Excitation is the change that occurs when inputs are inserted into memory. Excitation logic is what is happening between the input and current states and is where you consider FF characteristics. An excitation equation is simply the equations for this excitation logic.
- To get from the transition table to something that can actually be implemented, we need one of these excitation equations. These are the equations that represent the determination of the next state value as a function of the input values and additionally the values available from the flip-flops/latches. Excitation equations are usually developed through K-Maps.
- The order of these steps doesn't necessarily matter, however you *must* begin with understanding the problem, or your resulting work no matter the order will be a mess.
- Sometimes, not all of the steps are done, such as creating the diagrams, schematics, and Verilog implementation.

## 24 4/26/2020

### 24.1 Implementation With DFF Review

- One of the most frequent uses of FSMs is pattern matching, where the aim is to find three “1’s” in a row using a Moore-type FSM. These FSMS accept a single variable  $x$  as their input as either 0 or 1. Additionally, they output a single variable  $z$ , which is 0 at all points where the pattern *hasn’t* been found and 1 where the pattern *has*.

### 24.2 FSM Verilog Part 2

- There are many ways to design FSMs in Verilog, such as the currently covered next-state equations or modified FFs and conditional Verilog.
- Conditionals in Verilog function similarly to other high-level programming languages, with Verilog utilizing if, else if, and else blocks. If the statement for one of these is more than one statement, you must have a begin and an end.
- A case statement looks at specific inputs, such as an expression. This expression is then sent to an alternative component of the case statement to complete execution.
- A parameter is a modifier denoting something as a constant. We use this as a shortcut to state assignments. A parameter modifier combines variables, numbers, and constants.

### 24.3 Implementation With TFF

- Excitation is the change that occurs when inputs are applied to memory. Accordingly, the excitation logic is what is happening between the inputs and the current state. Excitation equations are the equations for excitation logic.

### 24.4 Memory Devices

- Memory is “something remembered from the past” and is the process used to reproduce or recall something that has previously been learned. Memory follows sequential logic.
- ROM stands for Read Only Memory and is a type of flash memory that is exclusively read only. These are also examples of combinational logic because memory as a device doesn’t need to be sequential. ROM is non-volatile because it’s logic is fixed and won’t change when power is removed.
- ROM is configured with  $2^n$  inputs and  $b$  outputs. These inputs and outputs are configured as a 2D array, with each row corresponding to a  $b$ -bit output. The result of this is that the inputs select the row, forcing us to read only the table currently configured by the ROM.
- We use ROM quite a bit, from decoders to representing the behavior of truth tables to predefined processes of a CPU.



- Programmable devices can, as implied by their name, be manufactured and configured (or programmed). The first programmable device was ROM, but this was a slow and expensive solution. We now have a logic system called Programmable Logic Array, or PLA. These are two-level AND-OR configurations, meaning they are quite good with hazards. Eventually, this logic system was reimagined into Programmable Array Logic, or PAL, which eventually became colloquially known as PLD, or Programmable Logic Devices.
- Let's expand upon this idea of PLAs. As previously mentioned, these are two level AND-OR configured devices and are represented in SOP form. These sound like decoders, but in reality aren't, as while the starting configuration is the same fuses are blown to program PLAs to a very specific configuration. These fuses are what make PLAs non-volatile.
- PALs are similar to PLAs but in these the OR gate is fixed, meaning we can't program that part of the device. These rely on the inputs selecting a set of connections that rely quite heavily on inversion, and as a result aren't particularly popular.
- PLDs evolved from PALs and are very popular, in contrast.
- Over time, we began scaling these devices further and further. PLDs were originally limited by their 2 level AND-OR configurations, so complex PLDs called CPLDs were developed through connecting many individual PLDs. Additionally, Field Programmable Gate Arrays, or FPGAs, took many CPLDs and smaller devices called Configurable Logic Blocks (CLBs) to create an architecture commonly used in phototyping.
- RAM is Random Access Memory and is memory that is simultaneously readable and writable. Technically, ROM is a type of RAM.
- Static RAM is by default asynchronous. When RAM is asynchronous, the control and data signals are not coordinated by an independent clock. This has its occasional use, but it can result in errors.
- Dynamic RAM uses D-Latches with between four to six transistors. The values are precharged and then assigned as they get periodically refreshed.
- We have so many different types of RAM because each type offers different capabilities in its speed, flexibility, stability, accuracy, power, and capacity.
- DDR stands for double data rate and allows for the transmission of data on both edges of the clock.
- FPGAs were made first available commercially in 1985 by Xilinx. Overall, these are a collection of configurable logic blocks (CLBs), local and global routing resources, I/O blocks (IOBs), programmable I/O buffers, and SRAM based configurable memory.
- Lookup tables (LUTs) are small units of ROM that a truth table is decomposed to. These share the same benefits as decoders. LUTs have a configuration of  $2^n \times 1$ , with  $n$  being between four and six in most cases. If this becomes too large, we can divide a LUT into numerous smaller LUTs and multiplex them.

- IOs are inputs and outputs to the FPGA. These have a buffering property and control the speed of transfer without modifying the overall device.
- Xilinx Spartan is a class of FPGA architecture. The IOB is the I/O block that allows for peripheral connections.
- We configure FPGAs using an HDL language, such as the familiar Verilog.