Contents

1 1/27/2020 **3**
1.1 Lecture Notes 3
1.2 Assigned Reading 3

2 1/29/2020 **5**
2.1 Lecture Notes 5
2.2 Assigned Reading 6

3 1/31/2020 **7**
3.1 Lecture Notes 7
3.2 Assigned Readings 8

4 2/3/2020 **11**
4.1 Lecture Slides 11
4.2 Assigned Reading 14

5 2/5/2020 **16**
5.1 Lecture Slides 16
5.2 Assigned Reading 18

6 2/7/2020 **20**
6.1 Lecture Notes 20
6.2 Assigned Reading 21

7 2/10/2020 **25**
7.1 Lecture Notes 25

8 2/12/2020 **27**
8.1 Lecture Notes 27
8.2 Assigned Reading 29

9 2/14/2020 **32**
9.1 Lecture Slides 32
9.2 Assigned Reading 34

10 2/17/2020 **36**
10.1 Lecture Slides 36

11 2/19/2020 **38**
11.1 Lecture Slides 38

12 2/21/2020 **40**
12.1 Lecture Slides 40
12.2 Assigned Reading 43

13 2/24/2020 **48**
13.1 Lecture Slides 48

14 2/26/2020 **51**
14.1 Lecture Slides 51

15 2/28/2020 **55**
15.1 Lecture Slides 55

16 3/02/2020 **58**
16.1 Lecture Slides 58
16.2 Assigned Reading 58

1 1/27/2020

1.1 Lecture Notes

- Don't cheat.

1.2 Assigned Reading

- 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.

  $\boxed{\text{Answer: } 0011111110100010}$

2.2 Assigned Reading

- 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

$$\begin{array}{rr} & 101111000 \\ & 10111110 \\ + & 10001101 \\ \hline & 101001011 \end{array} \qquad \begin{array}{rr} & 001011000 \\ & 10101101 \\ + & 00101100 \\ \hline & 11011001 \end{array}$$

$$\begin{array}{rr} & 011111110 \\ & 01111111 \\ + & 00111111 \\ \hline & 10111110 \end{array} \qquad \begin{array}{rr} & 000000000 \\ & 10101010 \\ + & 01010101 \\ \hline & 11111111 \end{array}$$

- Subtraction

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

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

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.

$$\begin{array}{rr} & 1111 \\ & 1011 \\ + & 101 \\ \hline & \text{b}10000 \end{array}$$

$\boxed{\text{Answer: b10000}}$

- Problem 3.1.2: Add b1011 to b1001.

$$\begin{array}{rr} & 1\ 11 \\ & 1011 \\ + & 1001 \\ \hline & \text{b}10100 \end{array}$$

$\boxed{\text{Answer: b10100}}$

- Problem 3.1.3: Subtract b111 from b1101.

$$\begin{array}{rr} & 1101 \\ - & 111 \\ \hline & \text{b}0110 \end{array}$$

$\boxed{\text{Answer: b0110}}$

- Problem 3.1.4: Subtract b1111 from b10101.

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

$\boxed{\text{Answer: No answer.}}$

- 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.

```
        1111
        1101
 +       111
 ------------
      b10100
```

$\boxed{\text{Answer: Proven. The red numbers signify overflow.}}$

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

```
        101
 -      111      Broke it!
 ------------
       ?10
```

$\boxed{\text{Answer: Proven. Writing either “Error” or “Broke it” is acceptable.}}$

3.2 Assigned Readings

- 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.

  - $01010101_2 = +85_{10}$
  - $01111111_2 = +127_{10}$
  - $00000000_2 = +0_{10}$
  - $11010101_2 = -85_{10}$
  - $11111111_2 = -127_{10}$
  - $10000000_2 = -0_{10}$

- 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 it usage in early computer. However, due to the adder 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.

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

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

- 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.

$$\begin{array}{rr} & 0100 \\ - & 0011 \\ \hline \end{array} \qquad \begin{array}{rr} & {\scriptstyle 1} \\ & 0100 \\ + & 1100 \\ \hline & 10001 \end{array}$$

$$\begin{array}{rr} & 0011 \\ - & 1100 \\ \hline \end{array} \qquad \begin{array}{rr} & {\scriptstyle 1} \\ & 0011 \\ + & 0011 \\ \hline & 0111 \end{array}$$

$$\begin{array}{rr} & 0011 \\ - & 0100 \\ \hline \end{array} \qquad \begin{array}{rr} & {\scriptstyle 1} \\ & 0011 \\ + & 1011 \\ \hline & 1111 \end{array}$$

$$\begin{array}{rr} & 1101 \\ - & 1100 \\ \hline \end{array} \qquad \begin{array}{rr} & {\scriptstyle 1} \\ & 1101 \\ + & 0011 \\ \hline & 10001 \end{array}$$

  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 $1 \to 0$ and $0 \to 1$. 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.
  $\boxed{\text{Answer: b00100000}}$

- 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.
    $$\begin{array}{rr} & \text{b11011111} \\ + & 1 \\ \hline & \text{b11100000} \end{array}$$

  $\boxed{\text{Answer: b11100000}}$

- 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.
  $\boxed{\text{Answer: b00001101}}$

- 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.
    $$\begin{array}{rr} & \text{b11110010} \\ + & 1 \\ \hline & \text{b11110011} \end{array}$$

  $\boxed{\text{Answer: b11110011}}$

- With the Excess-B system, the main idea is to moving 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.

$$\begin{array}{rr} & 1111 \\ & \text{b0111} \\ + & \text{b1001} \\ \hline & 10000 \end{array}$$

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

$\boxed{\text{Answer: b0000}}$

- Problem 4.1.12: Add 7 to -8 in 4-bit.
  $\boxed{\text{Answer: Can't be done in 4-bit, 8=1000.}}$
- 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.

$$\begin{array}{rr} & 1111 \\ & \text{b11111000} \\ + & \text{b11110100} \\ \hline & 111101100 \end{array}$$

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

$\boxed{\text{Answer: b11101100}}$

4.2 Assigned Reading

- 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.

$$\begin{array}{rrl} & 1011 & \text{multiplicand} \\ \times & 1101 & \text{multiplier} \\ \hline & 1011 & \\ & 0000 & \\ & 1011 & \text{shifted multiplicands} \\ & 1011 & \\ \hline & 10001111 & \text{product} \end{array}$$

- 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
$\times$	1101	multiplier
	0000	partial product
	1011	shifted multiplicand
	01011	partial product
	0000$\downarrow$	shifted multiplicand
	001011	partial product
	1011$\downarrow\downarrow$	shifted multiplicand
	0110111	partial product
	1011$\downarrow\downarrow\downarrow$	shifted multiplicand
	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 order 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 show below.

	1011	multiplicand
$\times$	1101	multiplier
	00000	partial product
	11011	shifted multiplicand
	111011	partial product
	00000$\downarrow$	shifted multiplicand
	1111011	partial product
	11011$\downarrow\downarrow$	shifted multiplicand
	11100111	partial product
	00101$\downarrow\downarrow\downarrow$	shifted and negated multiplicand
	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.

$$
\begin{array}{rr}
 & 0101 \\
\times & 0011 \\
\hline
 & 0101 \\
 & 0101\downarrow \\
 & 0000\downarrow\downarrow \\
 & 0000\downarrow\downarrow\downarrow \\
\hline
 & \text{b}00001111
\end{array}
$$

$\boxed{\text{Answer: Answer: b00001111}}$

- Problem 5.1.2: Multiply b0111 and b0011.

$$
\begin{array}{rr}
 & 0111 \\
\times & 0011 \\
\hline
 & 0011 \\
 & 0011\downarrow \\
 & 0011\downarrow\downarrow \\
 & 0000\downarrow\downarrow\downarrow \\
\hline
 & \text{b}00010101
\end{array}
$$

$\boxed{\text{Answer: b00010101}}$

- Problem 5.1.3: Multiply b0101 and b1011.

$$
\begin{array}{rr}
 & 0100 \\
+ & 1 \\
\hline
 & 0101
\end{array}
$$

$$
\begin{array}{rr}
 & 0101 \\
\times & 0101 \\
\hline
 & 0101 \\
 & 0000\downarrow \\
 & 0101\downarrow\downarrow \\
 & 0000\downarrow\downarrow\downarrow \\
\hline
 & \text{b}0011001
\end{array}
$$

$$
\begin{array}{rr}
 & 1100110 \\
+ & 1 \\
\hline
 & \text{b}11100111
\end{array}
$$

$\boxed{\text{Answer: b11100111}}$

- Problem 5.1.4: Multiply b1101 and b0011.

$$
\begin{array}{rr}
 & 0010 \\
+ & 1 \\
\hline
 & 0011
\end{array}
$$

$$
\begin{array}{rr}
 & 0011 \\
\times & 0011 \\
\hline
 & 0011 \\
 & 0011\downarrow \\
 & 0000\downarrow\downarrow \\
 & 0000\downarrow\downarrow\downarrow \\
\hline
 & 00001001
\end{array}
$$

$$
\begin{array}{rr}
 & 11110110 \\
+ & 1 \\
\hline
 & \text{b}11110111
\end{array}
$$

$\boxed{\text{Answer: b11110111}}$

- Problem 5.1.5: Multiply b1101 and b1011.

$$
\begin{array}{rr}
 & 0010 \\
+ & 1 \\
\hline
 & 0011
\end{array}
$$

$$
\begin{array}{rr}
 & 0100 \\
+ & 1 \\
\hline
 & 0101
\end{array}
$$

$$
\begin{array}{rr}
 & 0011 \\
\times & 0101 \\
\hline
 & 0011 \\
 & 0000\downarrow \\
 & 0011\downarrow\downarrow \\
 & 0000\downarrow\downarrow\downarrow \\
\hline
 & \text{b}00001111
\end{array}
$$

$\boxed{\text{Answer: b00001111}}$

- Problem 5.1.6: Multiply b1011 and b1011.

$$
\begin{array}{rr}
 & 0100 \\
+ & 1 \\
\hline
 & 0101
\end{array}
$$

$$
\begin{array}{rr}
 & 0101 \\
\times & 0101 \\
\hline
 & 0101 \\
 & 0000\downarrow \\
 & 0101\downarrow\downarrow \\
 & 0000\downarrow\downarrow\downarrow \\
\hline
 & \text{b}00011001
\end{array}
$$

$\boxed{\text{Answer: b00011001}}$

5.2 Assigned Reading

- 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 days the ten decimal digits are represented.

Decimal digit	**BCD (8421)**	**2421**	**Excess-3**	**Biquinary**	**1-out-of-10**
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.

- Similary 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 compliment, 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 somtimes 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 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
  $\boxed{\text{Answer: b10010111}}$

- Problem 6.1.2: Convert 36 to BCD.
  9 in BCD: b0011
  7 in BCD: b0110
  $\boxed{\text{Answer: b00110110}}$

- 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.
  $$\begin{array}{rr} & 14 = 00010100 \\ + & 27 = 00100011 \\ \hline & 00111011 \\ + & 0110 \\ \hline & \text{b01000001} \end{array}$$
  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.
  $\boxed{\text{Answer: b01000001}}$

- Problem 6.1.4: Add 9 and 13 in BCD.
  $$\begin{array}{rr} & 14 = 1001 \\ + & 27 = 00010011 \\ \hline & 00011100 \\ + & 110 \\ \hline & \text{b00100010} \end{array}$$
  $\boxed{\text{Answer: b00100010}}$

- 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

$.3 \cdot .2 = 0.6$	0
$.6 \cdot .2 = 1.2$	1
$.2 \cdot .2 = 0.4$	0
$.4 \cdot .2 = 0.8$	0
$.8 \cdot .2 = 1.6$	1
$.6 \cdot .2 = 1.2$	1
$.2 \cdot .2 = 0.4$	0
$.4 \cdot .2 = 0.8$	0
$.8 \cdot .2 = 1.6$	1
$.6 \cdot .2 = 1.2$	1

Answer: b110.0100110011

6.2 Assigned Reading

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

- 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 of 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 bewteen the truth table and minterms and maxterms. A minterm defined as a product term that is 1 is exactly one row of a truth table. Simiarly, a maxterm defined as a sum term that is 0 is 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. 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 $Y = 1$.
  $\boxed{\text{Answer: } Y = 1}$
- 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 $T = 1$.
  $\boxed{\text{Answer: } T = 1}$
- 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.
  - $x = x * x = x * x * x * x * x$
  - $x = x + x = x + x + x + x + x$
- 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: $T = Y' * F + X * Y$

  $\boxed{\text{Answer: } T = Y' * F + X * Y}$
- : 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: $X + T$

  $\boxed{\text{Answer: } Y = X + T}$
- 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: $T = X * Y' + Y$
  - Simplification: $T = Y + X$

  $\boxed{\text{Answer: } T = Y + X}$
- 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.
  $y = \overline{x}T + [\overline{x} \cdot \overline{T}]$
  $y = \overline{x}T + \overline{x}\overline{T}$
  $y = \overline{x}(T + \overline{T})$ (note that $T + \overline{T} = 1$)
  $y = \overline{x} \cdot 1$
  $y = \overline{x}$
  $\boxed{\text{Answer: } y = \overline{x}}$

- 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 $Y = XT + WT' + W'X$.
  $\overline{y} = \text{complement} = [XT + WT' + W'X]'$
  $(\overline{X} + \overline{T})(\overline{W} + \overline{\overline{T}})(\overline{\overline{W}} + \overline{X})$
  $(\overline{X} + \overline{T})(\overline{W} + T)(W + \overline{X})$
  $(\overline{XW} + \overline{X}T + \overline{TW} + \overline{T}T)(W + \overline{X})$
  $\overline{W}W\overline{X} + \overline{X}W\overline{X} + \overline{X}TW + \overline{X}T\overline{X} + \overline{TW}W + \overline{TWX}$
  $\overline{X}W + \overline{X}TW + \overline{X}T + \overline{TWX}$
  $\overline{XW}(1 + T) + \overline{X}WT + \overline{X}T$
  $\overline{XW} + \overline{X}T(W + 1)$
  $\overline{y} = \overline{XW} + \overline{X}T$
  $\boxed{\text{Answer: } \overline{y} = \overline{XW} + \overline{X}T}$

- Problem 8.1.3: Find the complement of $T = AB' + B(A' + C)$.
  $\overline{T} = [AB' + B(A' + C)]'$
  $(\overline{A} + \overline{\overline{B}})(\overline{B} + (\overline{\overline{A}} \cdot \overline{C})$
  $(\overline{A} + B)(\overline{B} + (A\overline{C}))$
  $\overline{AB} + \overline{A}(AC) + B\overline{B} + BA\overline{C}$
  $\overline{T} = \overline{AB} + BA\overline{C}$
  $\boxed{\text{Answer: } \overline{T} = \overline{AB} + BA\overline{C}}$

- 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.

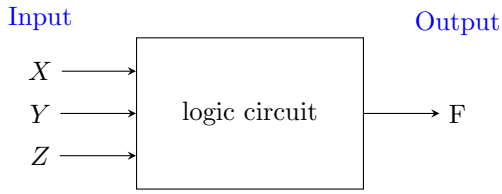
- A self dual is when you can take a dual and the resulting function generates the same outcomes as the original. Not all functions that have a dual can produce a self dual (most can't, in fact).

- 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 Reading

- 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.



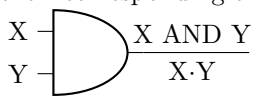
- A logic circuit whose outputs depend only on its current inputs is called a combinational circuit. The operation of such a circuit is fully described by a truth table that lists all combinations of input values and the corresponding output values. Below is an example of a truth table.

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

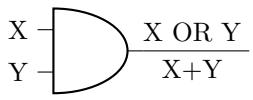
- A circuit with memory whose outputs depend on the current input in addition to the sequence of past inputs are called sequential circuits. Its behavior may be described by a state table, which specifies its output and next state as functions of its current state and input.

- 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.

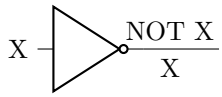
- Just three basic logic functions (AND, OR, and NOT) can be used to build any combinational logic circuit. The graphical symbols, along with their corresponding truth tables, are shown below.



X	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



X	NOT X
0	1
1	0

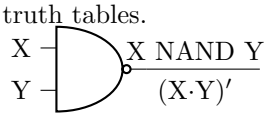
The gates' functions are easily described using words.

- 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.

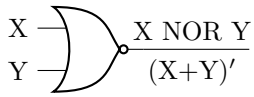
- Take notice of the below circle from the inverter symbol's output.

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.

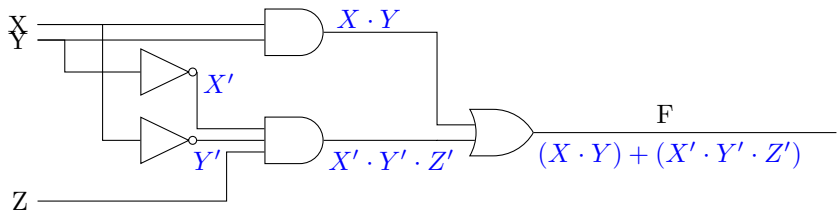


X	Y	X NAND Y
0	0	1
0	1	1
1	0	1
1	1	0



X	Y	X NOR Y
0	0	1
0	1	0
1	0	0
1	1	0

- 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 hows 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.
  $\boxed{\text{Answer: Dual of } Y = X \cdot (W' + T) \cdot (W + T')}$

- 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.
  $\boxed{\text{Answer: Dual of } F = (A' + B) \cdot (A + B + C')}$

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

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

- 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.
  $\boxed{\text{Answer: } F = (x + y + \overline{z})(\overline{x} + y + z)(\overline{x} + \overline{y} + \overline{z})(\overline{x} + \overline{y} + z)}$

- 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.
  $\boxed{\text{Answer: } F = (x + y + z)(\overline{x} + y + z)(\overline{x} + \overline{y} + z)}$

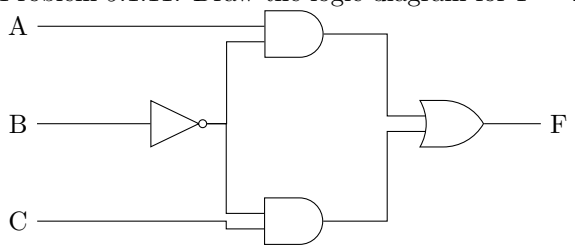
- On-set and off-set is a system used to describe the shorthand list of items. Minterms in shorthand have a list of the rows with 1 as the output, making it on-set. The complementary idea also holds true, with off-set being for maxterms.

- Logic gates are used to physically represent functions and can be drawn. AND, OR, and NOT are said to create a complete logic set.

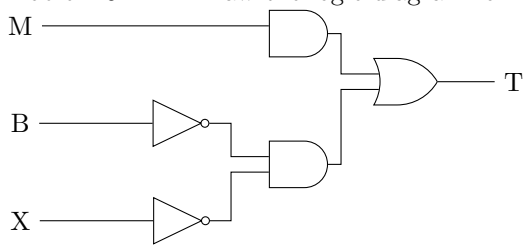
- Logic diagrams are drawings that help with design and preparation for implementation. For now, we will only use computational logic.

- 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 Reading

- 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...
  $F = \sum_{N_3,N_2,N_1,N_0}(1,2,3,5,7,11,13)$
  $= N_3' \cdot N_2' \cdot N_1' \cdot N_0 + N_3' \cdot N_2' \cdot N_1 \cdot N_0' + N_3' \cdot N_2' \cdot N_1 \cdot N_0 + N_3' \cdot N_2 \cdot N_1' \cdot N_0 + N_3' \cdot N_2 \cdot N_1 \cdot N_0 + N_3 \cdot N_2' \cdot N_1 \cdot N_0 + N_3 \cdot N_2 \cdot N_1' \cdot N_0$

- 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.

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

- 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.

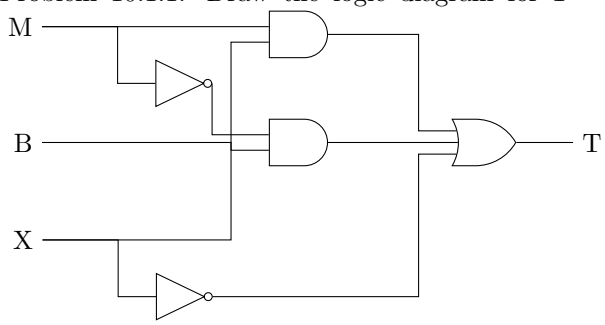
- In general, when we are designing a logic function for an application, it is easier to describe the logic function in words using logical connectives than it is to write a complete truth table (especially if the number of variables is large).

- 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.

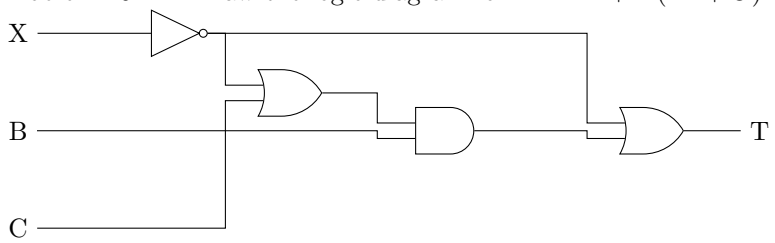
10 2/17/2020

10.1 Lecture Slides

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

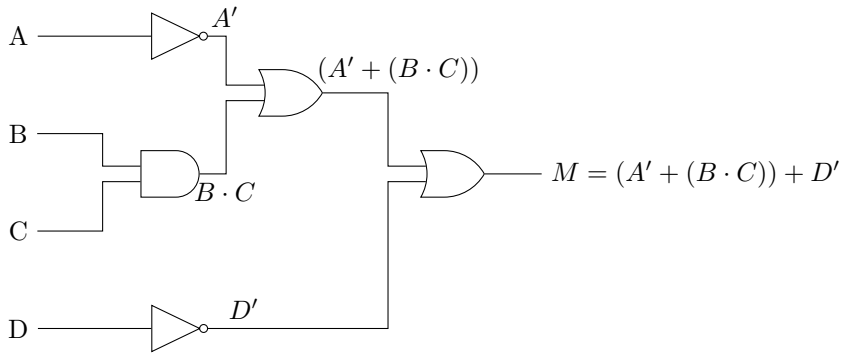


- Problem 10.1.2: Draw the logic diagram for $T = X' + B(X' + C)$.

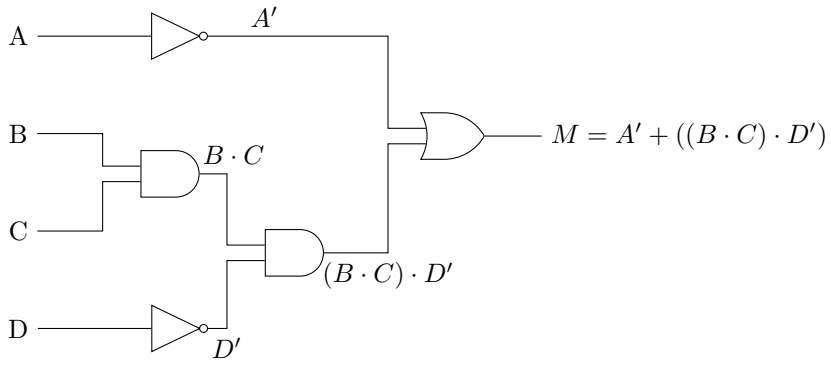


- You can always follow the path of connections in a diagram to see the behavior that it is implementing. This is done by labeling the function after each gate. See the below problems for examples.

- 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.

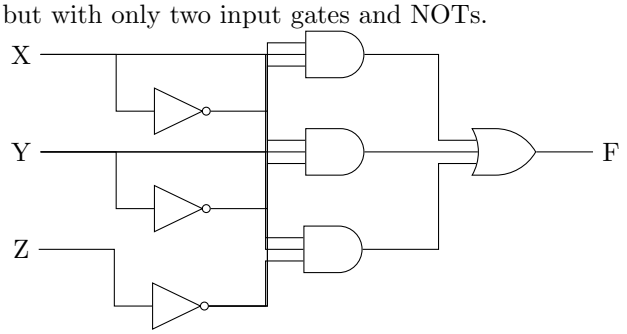


- Combinational logic synthesis is the process that specifies the required function and creates the details for implementation. Combinational logic design is the broader overview of the entire process, which includes logic synthesis.

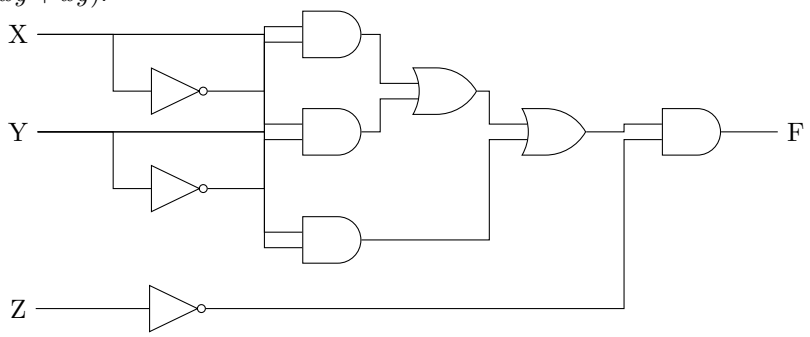
11 2/19/2020

11.1 Lecture Slides

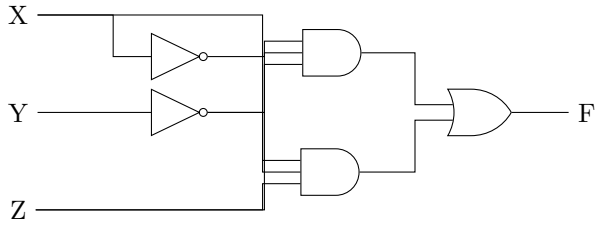
- A manipulator is a fancy way of saying that we can make an equivalent function with different arrangements.
- Problem 11.1.1: Draw the canonical logic diagram for $F = \sum_{x,y,z}(2,4,6)$ but with only two input gates and NOTs.



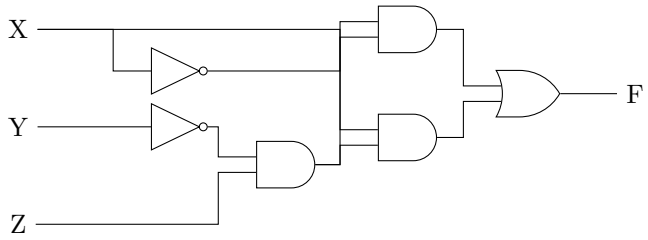
This is a two level, three input gate $F = \overline{x}y\overline{z} + x\overline{y}\overline{z} + xy\overline{z}$. Using the associative property, we can create a multi-level two input gate $F = \overline{z}(\overline{x}y + x\overline{y} + xy)$.



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



- 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?

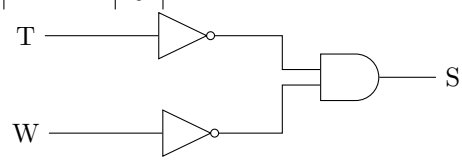
  After asking these questions, you can build the design.

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

  Sprinkler: On^1 or off ^{0} = output →S

T	W	S
0	0	1
0	1	0
1	0	0
1	1	0

$S = \overline{T} \cdot \overline{W}$



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

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

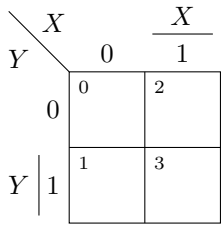
$N = \overline{H} \cdot \overline{C}$
$\overline{H} \cdot \overline{C} = N$

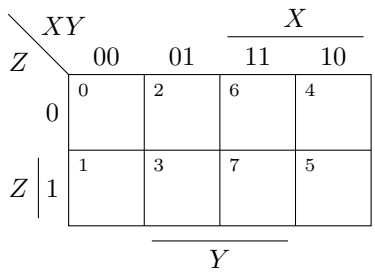
$N = \overline{H} \cdot \overline{C} + H\overline{C} = \overline{C}$

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 $F = \prod_{x,y,z}(1,2,5,6)$.
  $F = (x + y + \overline{z})(x + \overline{y} + z)(\overline{x} + y + \overline{z})(\overline{x} + \overline{y} + z)$
  $F = (\cancel{xx}^{x} + x\overline{y} + xz + yx + \cancel{y\overline{y}}^{0} + yz + \overline{z}x + \overline{z}x + \cancel{\overline{z}z})^{0}$
  $(\cancel{\overline{x}x}^{x} + \overline{x}\overline{y} + y\overline{x} + \cancel{y\overline{y}}^{0} + yz + \overline{z}\overline{x} + \overline{z}\overline{y} + \cancel{\overline{z}z}^{0})$
  $F = x\overline{x} + x\overline{x}\overline{y} + x\overline{x}z + xy\overline{z} + xyz + x\overline{z}\overline{x} + x\overline{z}\overline{y}$
  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.





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

W (11, 10); Z (01, 11); Y (11, 10); X (01, 10)

- 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:

$y \backslash x$	0	1
0	0	2
1	1	3

x (1); y (1)

- 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: z \ xy	00	01	11	10
0	000	010	110	100
1	001	011	111	101

x (11, 10); y (01, 11)

- 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.
  1. Place all of the minterms 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.
  3. Determine the distinguished 1's. Using these distinguished 1'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 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.
  $F = \overline{x}\overline{y}\overline{z} + x\overline{y}\overline{z} + x\overline{y}z + xy\overline{z} + xyz$
  $F = \overline{y}\overline{z}(\cancel{\overline{x} + \overline{x}}^{1}) + x\overline{y}z + xy(\cancel{\overline{z} + \overline{z}}^{1})$
  $F = \overline{y}\overline{z} + x\overline{y}z + xy$
  $\boxed{\text{Answer: } F = \overline{y}\overline{z} + x\overline{y}z + xy}$

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

F:

$z \backslash xy$	00	01	11	10
0	0: 1	2	6: 1	4: 1
1	1	3	7: 1	5: 1

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

110	xy$\overline{z}$
100	x$\overline{yz}$
111	xyz
101	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
$\boxed{\text{Answer: } F = \overline{yz} + x}$

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

F:

$z \backslash xy$	00	01	11	10
0	0: 1	2	6	4: 1
1	1: 1	3	7: 1	5: 1

Distinguished 1's: 0, 1, 4, 7
Essential prime implicant(s): $\overline{y}$, xz
$\boxed{\text{Answer: } F = \overline{y} + xz}$

12.2 Assigned Reading

- 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.

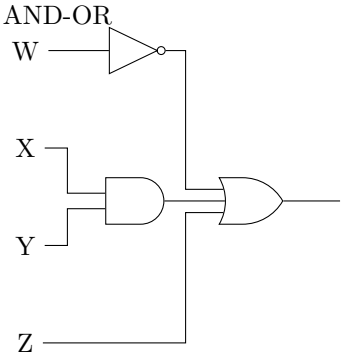
- NAND and NOR connectives aren't very logically intuitive, however. For example, you wouldn't say "I won't date you if you're not clean or not wealthy and also you're not smart or not friendly." You would instead say "I'll date you if you're clean and wealth or if you're smart and friendly." To produce 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.

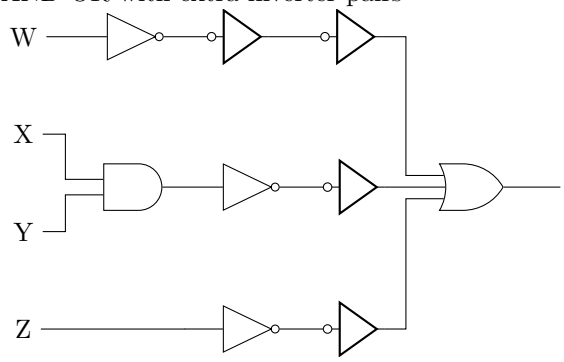
- We can insert a pair of inverters between each AND-gate output and the corresponding OR-gate input in a two-level AND-OR circuit. These inverters, per T4 (see page 21) have no effect on the output function of a circuit.

- 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.

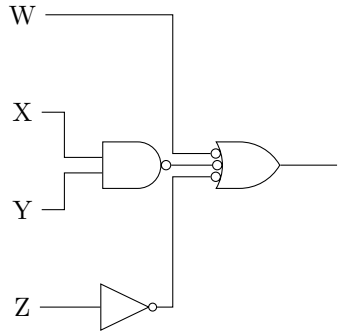
a. AND-OR



b. AND-OR with extra inverter pairs

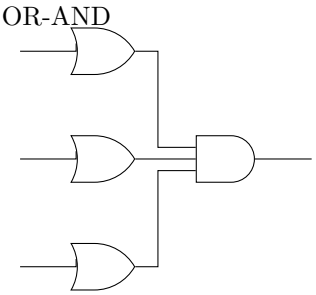


c. NAND-NAND

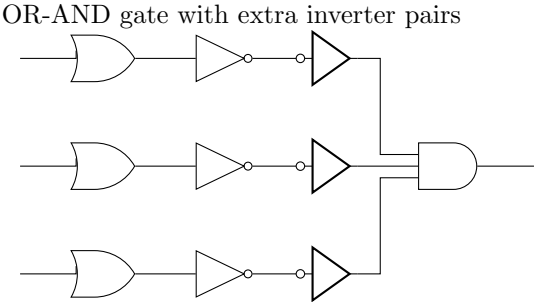


- The dual of this idea also holds true, in that any product-of-sums expression can be realized as an OR-AND circuit or as a NOR-NOR circuit. Below is a realization of a product-of-sums circuit.

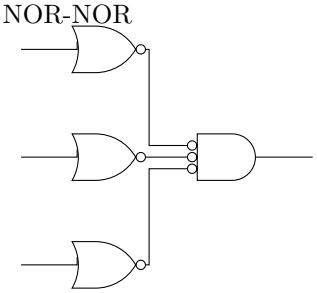
a. OR-AND



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

$Y \backslash X$	0	$\overline{X}$ 1
0	0	2
$Y \mid 1$	1	3

  (b) 3-variable

$Z \backslash XY$	00	01	$\overline{X}$ 11	10
0	0	2	6	4
$Z \mid 1$	1	3	7	5
		$\overline{Y}$		

(c) 4-variable

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

W (11, 10); Z (01, 11); Y (11, 10); X (01, 11)

- Observe the below K-Map.

F:

$CD \backslash AB$	00	01	11	10
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

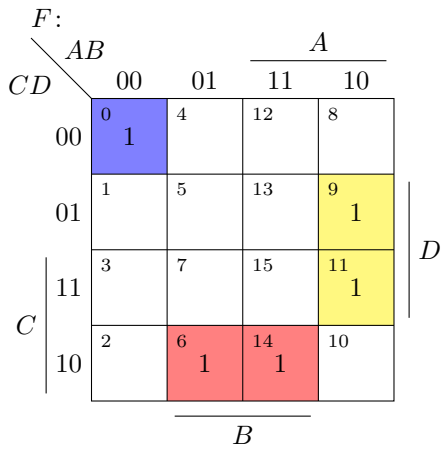
A (11, 10); D (01, 11); C (11, 10); B (00, 01)

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

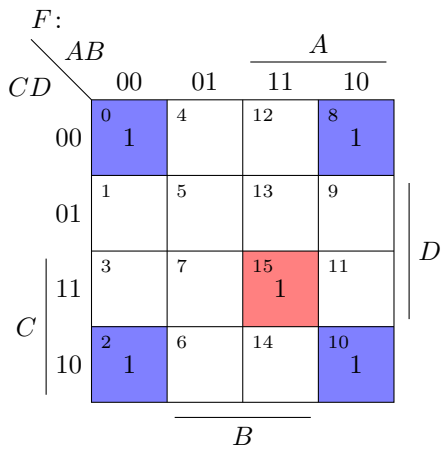
13.1 Lecture Slides

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



Distinguished 1's: 0,6,9,11,14
Essential prime implicant(s): $\overline{ABCD}$, $A\overline{B}D, BC\overline{D}$
$\boxed{\text{Answer: } F = \overline{ABCD} + A\overline{B}D + BC\overline{D}}$

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



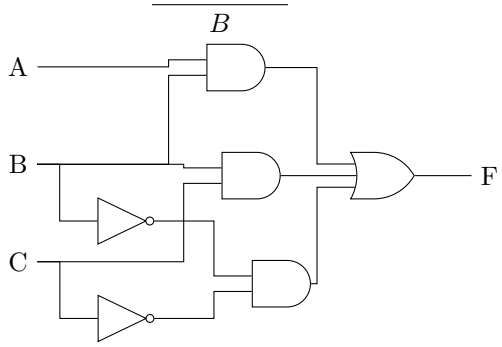
Distinguished 1's: 0,2,8,10,15
Essential prime implicant(s): $ABCD, \overline{BD}$
$\boxed{\text{Answer: } F = ABCD + \overline{BD}}$

- 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:

$C \backslash AB$	00	01	11	10
0	0: 1	2	6: 1	4: 1
1	1	3: 1	7: 1	5

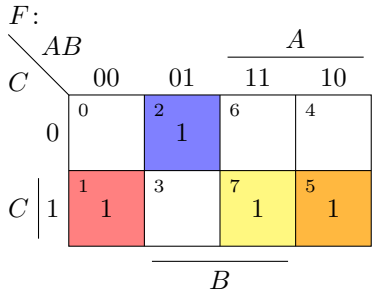


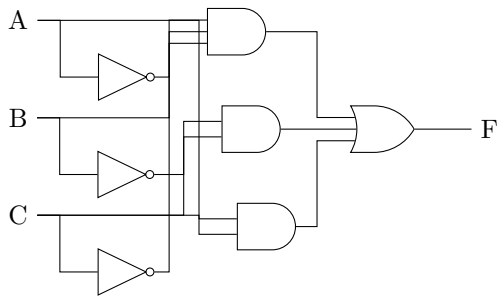
Distinguished 1's: 0, 3
Essential prime implicant(s): $\overline{BC}, BC$

$\boxed{\text{Answer: } F = \overline{BC} + BC + A\overline{C}}$

- 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



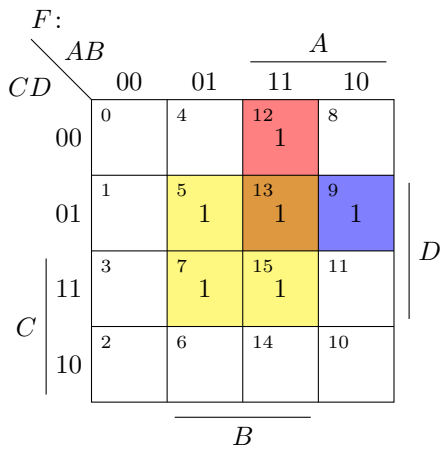


Distinguished 1's: 1, 2, 7

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

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

- 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.



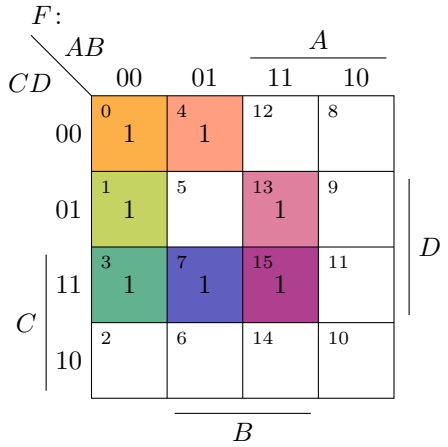
Distinguished 1's: 5, 7, 12, 15, 9
Essential prime implicant(s): $AB\overline{C}, A\overline{C}D, BD$

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.

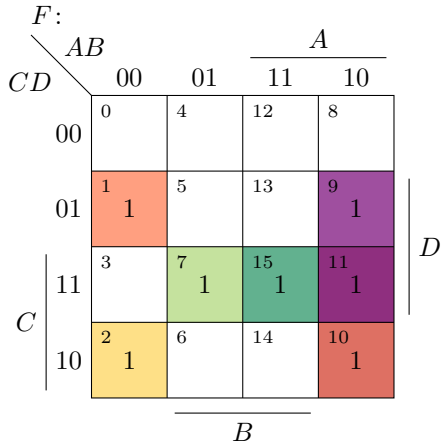
- Problem 14.1.1: Draw the K-Map for $F = \sum_{A,B,C,D}(0, 1, 3, 4, 7, 13, 15)$ and state the cost.



Distinguished 1's: 4, 13
Essential prime implicant(s): $\overline{ACD}, ABD$

$\boxed{\text{Answer: } F = \overline{ACD} + ABD + \overline{AB}D + BCD \therefore \text{ Cost is } 4}$

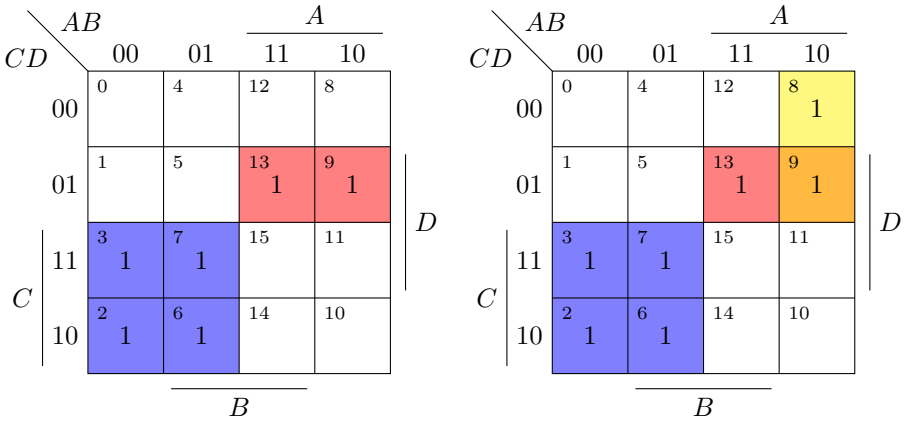
- Problem 14.1.2: Draw the K-Map for $F = \sum_{A,B,C,D}(1, 2, 7, 9, 10, 11, 15)$ and state the cost.



Distinguished 1's: 1, 2, 7
Essential prime implicant(s): $\overline{BC}D, \overline{B}C\overline{D}, BCD$

$\boxed{\text{Answer: } F = \overline{BC}D + \overline{B}C\overline{D} + BCD + A\overline{B}D + A\overline{B}C \therefore \text{ Cost is } 5}$

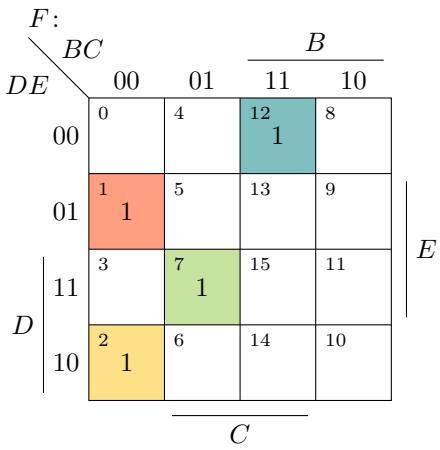
- 5-variable K-Maps take the form of $f(A, B, C, D, E)$. To solve these, we use two 4-variable K-Maps, one with $A = 0$ (for 0-15) and one with $A = 1$ (for 16-31). We combine adjacent 1's in three directions: vertically, horizontally, and out of the page. Below is an example of a 5-variable K-Map.



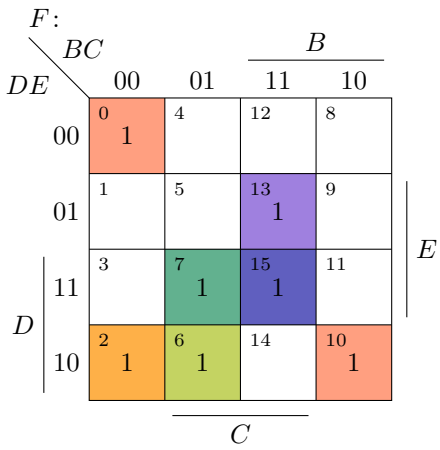
(a) $E = 0$ (b) $E = 1$

$$\boxed{\text{Answer: } F = AC + A\overline{C}D + A\overline{BC}E}$$

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



(a) $A = 0$



(b) $A = 1$

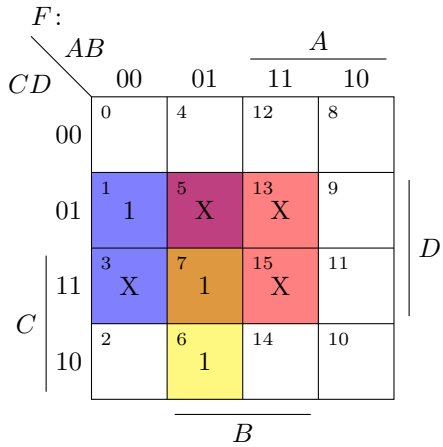
Distinguished 1's: 1, 12, 2, 7, 16, 26, 29

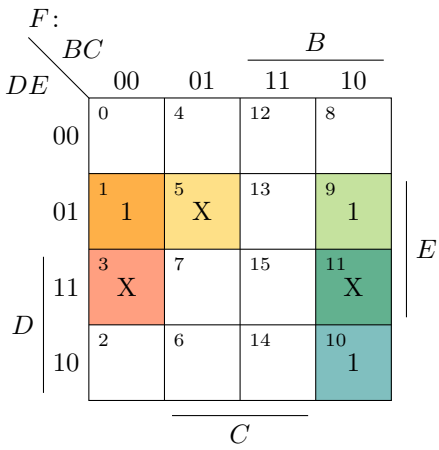
Essential prime implicants:

$\overline{ABCD}E, \overline{A}BC\overline{DE}, \overline{BC}D\overline{E}, \overline{B}CDE, A\overline{BCE}, ABCE, A\overline{C}D\overline{E}$

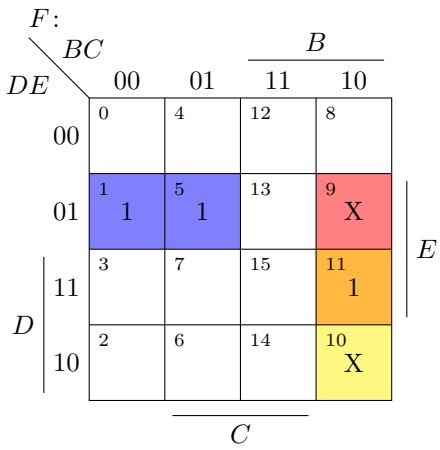
Answer: $F = \overline{ABCD}E + \overline{A}BC\overline{DE} + \overline{BC}D\overline{E} + \overline{B}CDE + A\overline{BCE} + ABCE + A\overline{C}D\overline{E}$

- 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 $F(A, B, C, D, E) = \sum(1, 9, 10, 17, 21, 27) + d(3, 5, 11, 25, 26)$.
- Problem 14.1.5: Draw the K-Map for $F(A, B, C, D) = \sum(1, 6, 7) + d(3, 5, 13, 15)$.





(a) $A = 0$



(b) $A = 1$

Distinguished 1's: 1, 9, 10, 17, 21
Essential prime implicants: $\overline{B}\overline{D}E, B\overline{C}E, B\overline{C}D$

$\boxed{\text{Answer: } F = \overline{B}\overline{D}E + B\overline{C}E + B\overline{C}D}$

Distinguished 1's: 1, 6
Essential prime implicant(s): $\overline{A}D, \overline{A}BC$

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

15 2/28/2020

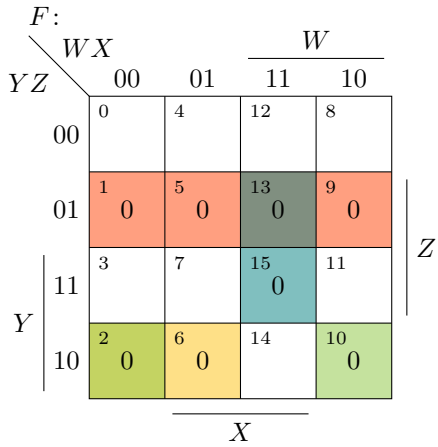
15.1 Lecture Slides

- 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 $F = \prod_{w,x,y,z}(1, 2, 5, 6, 9, 10, 13, 15)$.



Distinguished 0's: 1, 5, 6, 9, 10, 15
Essential prime implicant(s): $(W + \overline{Y} + Z), (X + \overline{Y} + Z), (Y + \overline{Z}), (\overline{W} + \overline{X} + \overline{Y})$

$\boxed{\text{Answer: } F = (W + \overline{Y} + Z)(X + \overline{Y} + Z)(Y + \overline{Z})(\overline{W} + \overline{X} + \overline{Y})}$

- 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: YZ \ WX	00	01	11	10
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 0	10 0

W X Y Z

Distinguished 0's: 5, 6, 9, 10
Essential prime implicant(s): $(Y + \overline{Z}), (\overline{Y} + Z)$

$\boxed{\text{Answer: } F = (Y + \overline{Z})(\overline{Y} + Z)(\overline{W} + \overline{X} + \overline{Z})}$

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

F: Z \ XY	00	01	11	10
0	0	2 X	6 0	4
1	1 0	3	7 X	5 0

X Y Z

Distinguished 0's: 1
Essential prime implicant(s): $(Y + \overline{Z})$

$\boxed{\text{Answer: } F = (Y + \overline{Z})(\overline{X} + \overline{Y})}$

- 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. $F = \overline{A}\overline{B}C\overline{D} + \overline{A}\overline{B}C\overline{D} + \overline{A}\overline{B}\overline{D} + \overline{A}BC + \overline{A}BCD + A\overline{B}\overline{C}D + A\overline{B}\overline{C} + A\overline{B}\overline{D}$.

F:

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

A, B, C, D

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

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

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

16 3/02/2020

16.1 Lecture Slides

16.2 Assigned Reading