- 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

	101111000			001011000	
	10111110			10101101	
+	10001101		+	00101100	
	101001011			11011001	

	011111110			000000000	
	01111111			10101010	
+	00111111		+	01010101	
	10111110			11111111	

- Subtraction

	001111100			001011000	
	11100101			10101101	
−	00101110		−	00101100	
	10110111			11011001	

	010101010			000000000	
	10101010			11011101	
−	01010101		−	01001100	
	01010101			10010001	

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.

$$\begin{array}{rr} & {\scriptstyle 1111} \\ & 1011 \\ + & 101 \\ \hline & b10000 \end{array}$$

Answer: b10000

- Problem 3.1.2: Add b1011 to b1001.

$$\begin{array}{rr} & {\scriptstyle 1\ 11} \\ & 1011 \\ + & 1001 \\ \hline & b10100 \end{array}$$

Answer: b10100

- Problem 3.1.3: Subtract b111 from b1101.

$$\begin{array}{rr} & 1101 \\ - & 111 \\ \hline & b0110 \end{array}$$

Answer: b0110

- Problem 3.1.4: Subtract b1111 from b10101.

$$\begin{array}{rr} & 10101 \\ - & 1111 \\ \hline & 110 \end{array}$$

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.

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

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

- 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.
  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.

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

  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.

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

  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.

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.

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

  - 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.

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

- 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.

$$\begin{array}{rr} & 0101 \\ \times & 0011 \\ \hline & 0101 \\ & 0101\downarrow \\ & 0000\downarrow\downarrow \\ & 0000\downarrow\downarrow\downarrow \\ \hline & \text{b00001111} \end{array}$$

Answer: Answer: b00001111

- Problem 5.1.2: Multiply b0111 and b0011.

$$\begin{array}{rr} & 0111 \\ \times & 0011 \\ \hline & 0011 \\ & 0011\downarrow \\ & 0011\downarrow\downarrow \\ & 0000\downarrow\downarrow\downarrow \\ \hline & \text{b00010101} \end{array}$$

Answer: b00010101

- Problem 5.1.3: Multiply b0101 and b1011.

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

$$\begin{array}{rr} & 0101 \\ \times & 0101 \\ \hline & 0101 \\ & 0000\downarrow \\ & 0101\downarrow\downarrow \\ & 0000\downarrow\downarrow\downarrow \\ \hline & \text{b0011001} \end{array}$$

$$\begin{array}{rr} & 1100110 \\ + & 1 \\ \hline & \text{b11100111} \end{array}$$

Answer: b11100111

- Problem 5.1.4: Multiply b1101 and b0011.

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

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

$$\begin{array}{rr} & 11110110 \\ + & 1 \\ \hline & b11110111 \end{array}$$

Answer: b11110111

- Problem 5.1.5: Multiply b1101 and b1011.

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

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

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

Answer: b00001111

- Problem 5.1.6: Multiply b1011 and b1011.

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

$$\begin{array}{rr} & 0101 \\ \times & 0101 \\ \hline & 0101 \\ & 0000\downarrow \\ & 0101\downarrow\downarrow \\ & 0000\downarrow\downarrow\downarrow \\ \hline & b00011001 \end{array}$$

Answer: b00011001

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.

$$\begin{array}{rr} & 14 = 00010100 \\ + & 27 = 00100011 \\ \hline & 00111011 \\ + & 0110 \\ \hline & b01000001 \end{array}$$

  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.

$$\begin{array}{rr} & 14 = 1001 \\ + & 27 = 00010011 \\ \hline & 00011100 \\ + & 110 \\ \hline & b00100010 \end{array}$$

  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.
  $6 = 0110$

$.3 \cdot .2 = 0.6$	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 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.

  - 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, 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.

- 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 $F = A'B + ABC'$.

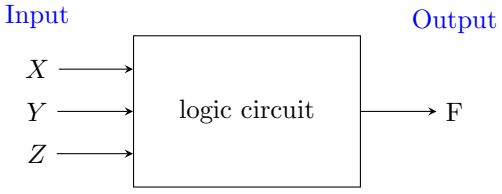
A	B	C	$\overline{A}B$	$AB\overline{C}$	F
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	1	0	1
0	1	1	1	0	1
1	0	0	0	0	0
1	0	1	0	0	0
1	1	0	0	1	1
1	1	1	0	0	0

- 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.



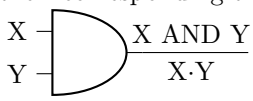
- A logic circuit whose outputs depend only on its current inputs is called a combinational circuit. The operation of such a circuit is fully described by a truth table that lists all combinations of input values and the corresponding output values. Below is an example of a truth table.

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

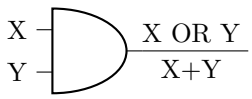
- A circuit with memory whose outputs depend on the current input in addition to the sequence of past inputs are called sequential circuits. Its behavior may be described by a state table, which specifies its output and next state as functions of its current state and input.

- 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.

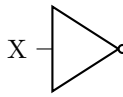
- Just three basic logic functions (AND, OR, and NOT) can be used to build any combinational logic circuit. The graphical symbols, along with their corresponding truth tables, are shown below.



X	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

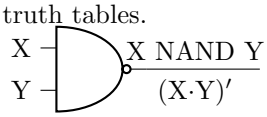
The gates' functions are easily described using words.

- 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.

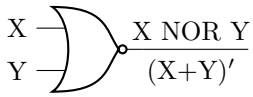
- Take notice of the below circle from the inverter symbol's output.

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.

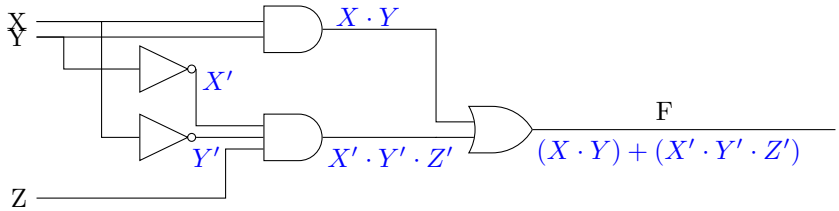


X	Y	X NAND Y
0	0	1
0	1	1
1	0	1
1	1	0



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

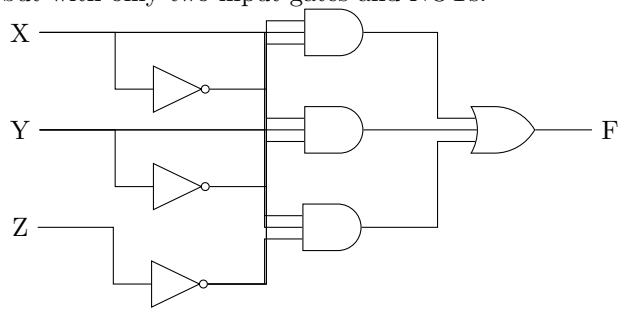
time-varying pattern of various input signals. Time is graphed horizontally and logic values are graphed vertically.

- 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.

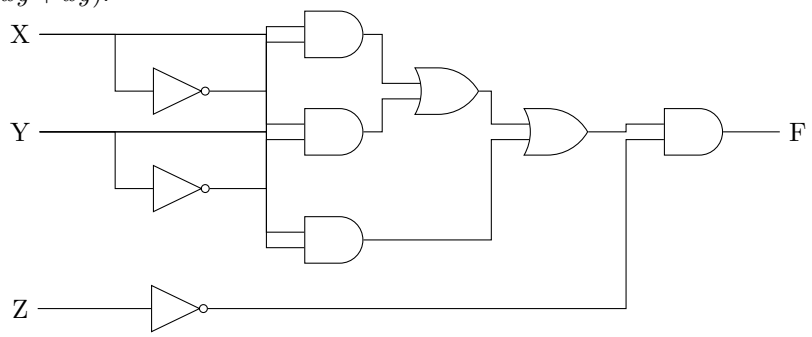
11 2/19/2020

11.1 Lecture Slides

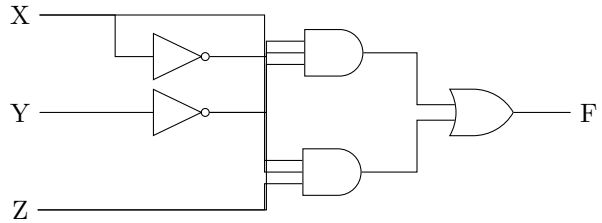
- A manipulator is a fancy way of saying that we can make an equivalent function with different arrangements.
- Problem 11.1.1: Draw the canonical logic diagram for $F = \sum_{x,y,z}(2, 4, 6)$ but with only two input gates and NOTs.



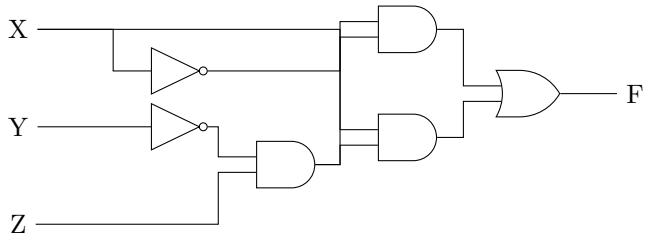
This is a two level, three input gate $F = \overline{x}y\overline{z} + x\overline{yz} + xy\overline{z}$. Using the associative property, we can create a multi-level two input gate $F = \overline{z}(\overline{x}y + x\overline{y} + xy)$.



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

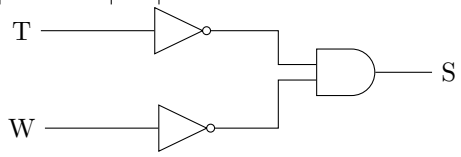
  After asking these questions, you can build the design.

- 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

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

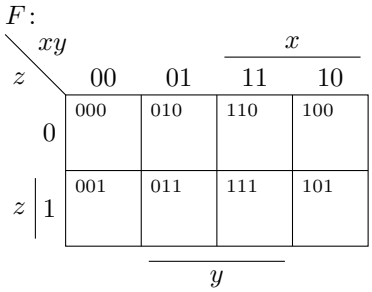


- Problem 11.1.4: Design a system to determine if you are allowed to watch Netflix.
  Inputs: Homework (have^1 or have not^0)→H
  Class (in class^1 or not in class^0)→C
  Output: Watch Netflix (yes^1 or no^0)→N

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

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

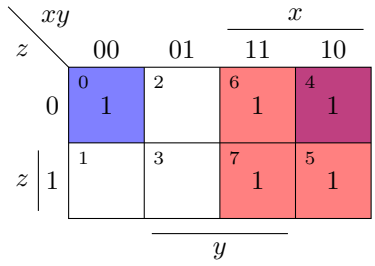
$N = \overline{H} \cdot \overline{C} + H\overline{C} = \overline{C}$



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



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

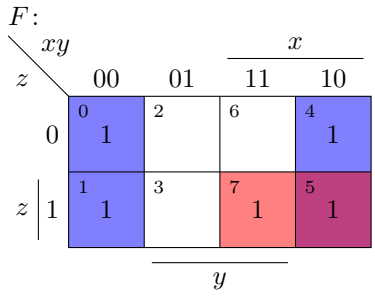
110	xy$\overline{z}$
100	x$\overline{yz}$
111	xyz
101	x$\overline{y}$z

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
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:



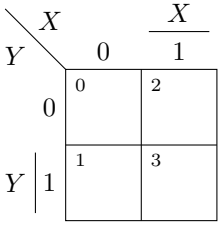
Distinguished 1's: 0, 1, 4, 7
Essential prime implicant(s): $\overline{y}$, xz
Answer: $F = \overline{y} + xz$

12.2 Assigned Readings

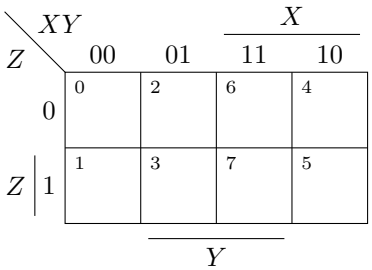
- We have previously described AND, OR, and NOT gates. We also want to use NAND and NOR gates, as these are faster than AND and OR gates in most technologies.

- 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.

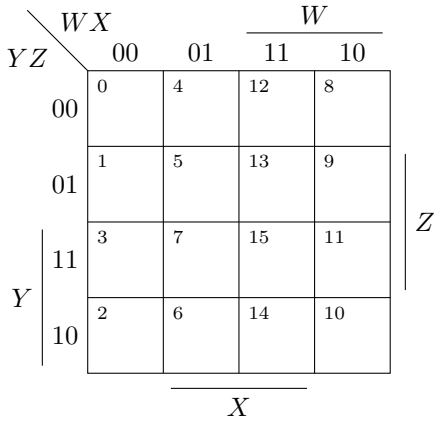
  (a) 2-variable



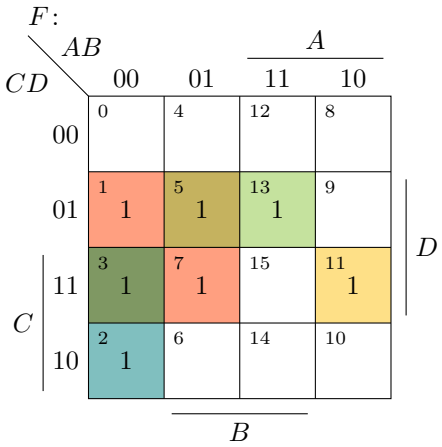
  (b) 3-variable



(c) 4-variable



- Observe the below K-Map.

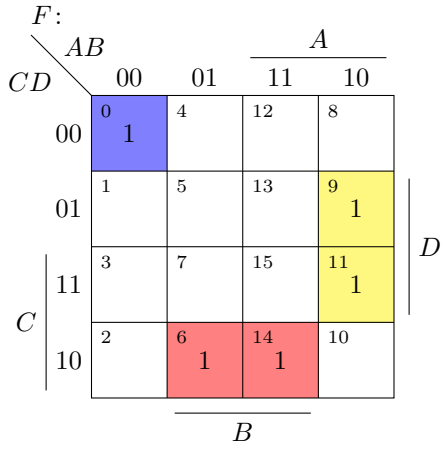


Take note of how adjacent 1 cells were grouped to correspond to their prime implicant, or "product terms that cover only input combinations for which the function has a 1 output, and that would cover at least one input combination with a 0 output if any variable were removed."

13 2/24/2020

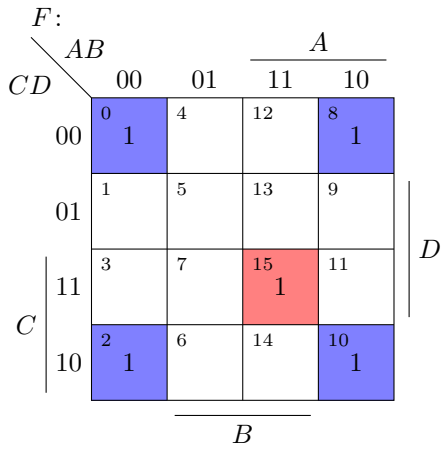
13.1 Lecture Slides

- 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): $\overline{ABCD}$, $A\overline{B}D$, $BC\overline{D}$
Answer: $F = \overline{ABCD} + A\overline{B}D + BC\overline{D}$

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

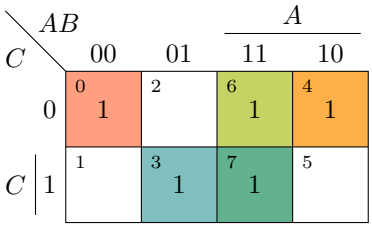


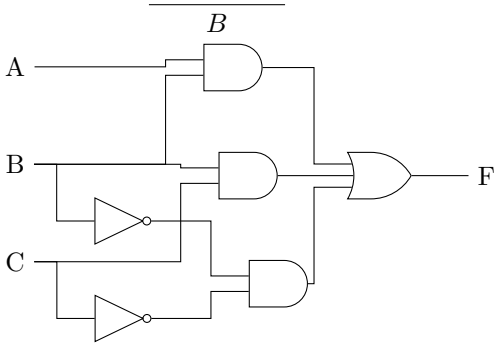
Distinguished 1's: 0,2,8,10,15
Essential prime implicant(s): $ABCD, \overline{BD}$
Answer: $F = ABCD + \overline{BD}$

- 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:

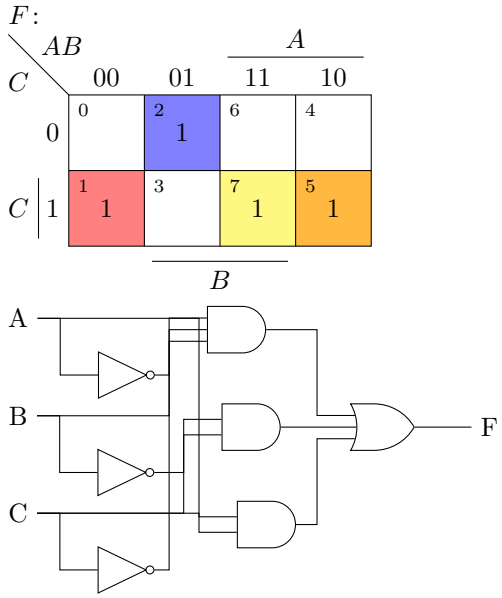




Distinguished 1's: 0, 3
Essential prime implicant(s): $\overline{BC}, BC$
Answer: $F = \overline{BC} + BC + A\overline{C}$

- 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

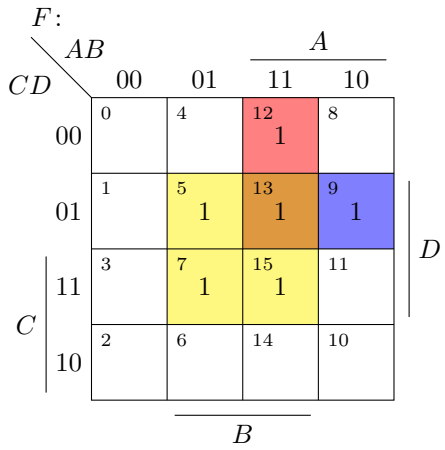


Distinguished 1's: 1, 2, 7

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

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.



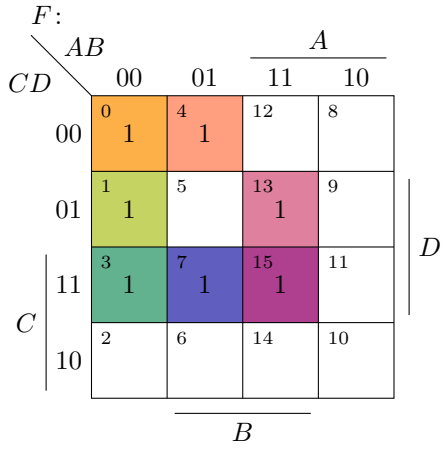
Distinguished 1's: 5, 7, 12, 15, 9

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

14 2/26/2020

14.1 Lecture Slides

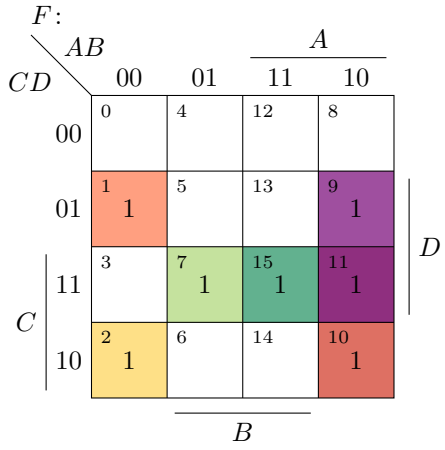
- Equipment costs money, so we need to have a method to describe the cost of a function. This is done by counting the number of gates. Due to standard practice, initial NOT gates are not counted.
- Problem 14.1.1: Draw the K-Map for $F = \sum_{A,B,C,D}(0, 1, 3, 4, 7, 13, 15)$ and state the cost.



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

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

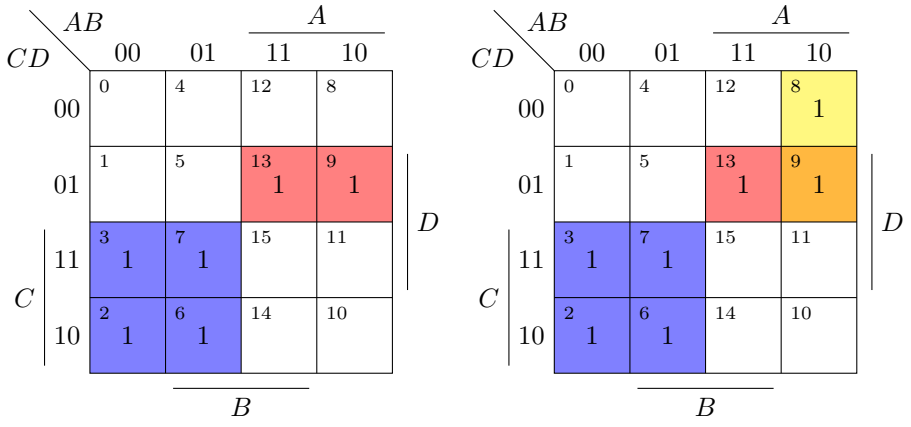
- Problem 14.1.2: Draw the K-Map for $F = \sum_{A,B,C,D}(1, 2, 7, 9, 10, 11, 15)$ and state the cost.



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

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

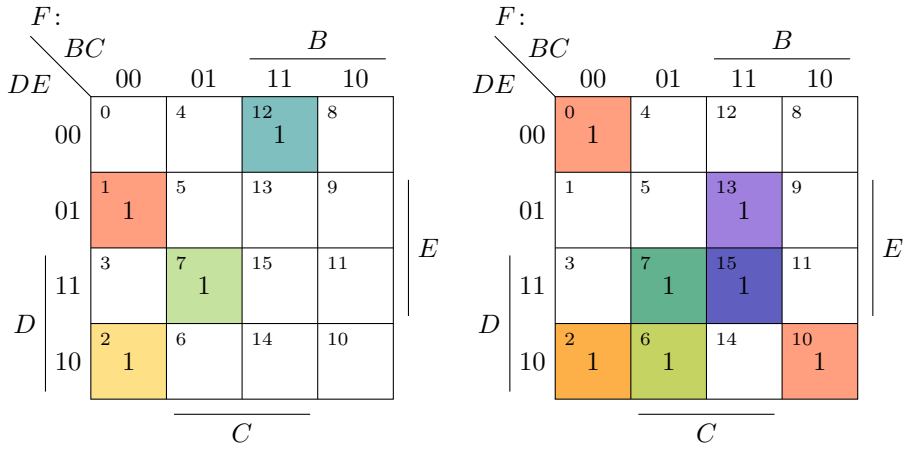
- 5-variable K-Maps take the form of $f(A, B, C, D, E)$. To solve these, we use two 4-variable K-Maps, one with $A = 0$ (for 0-15) and one with $A = 1$ (for 16-31). We combine adjacent 1's in three directions: vertically, horizontally, and out of the page. Below is an example of a 5-variable K-Map.



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

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) $A = 1$

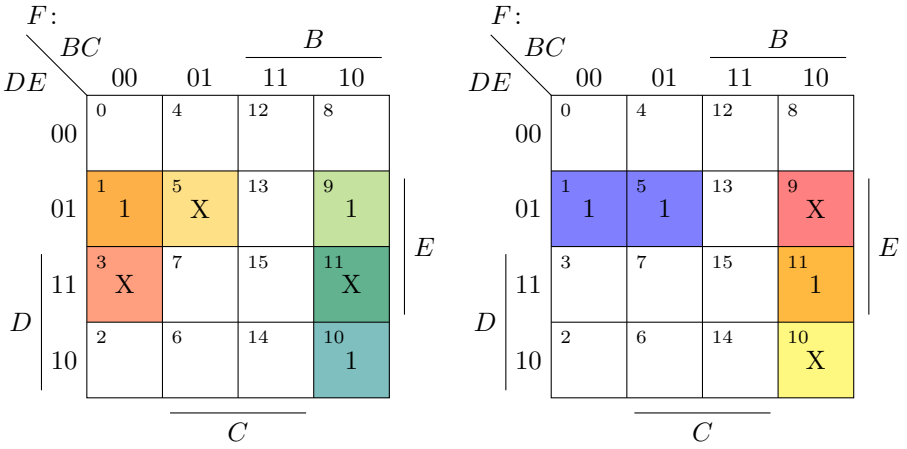
Distinguished 1's: 1, 12, 2, 7, 16, 26, 29
EPIs: $\overline{A}\overline{B}CDE, \overline{A}B\overline{C}\overline{D}\overline{E}, \overline{B}C\overline{D}\overline{E}, \overline{B}CDE, A\overline{B}\overline{C}E, ABCE, A\overline{C}\overline{D}\overline{E}$

$F = \overline{A}\overline{B}CDE + \overline{A}B\overline{C}\overline{D}\overline{E} + \overline{B}C\overline{D}\overline{E} + \overline{B}CDE + A\overline{B}\overline{C}E + ABCE + A\overline{C}\overline{D}\overline{E}$

- 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.

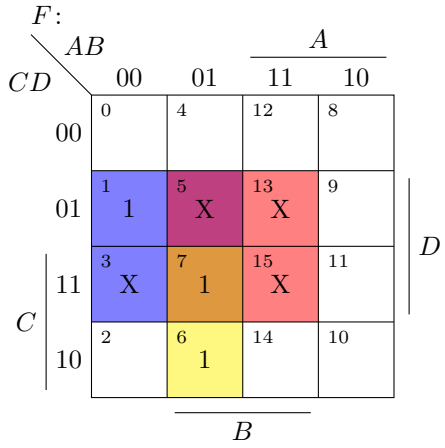
- When forming prime implicant's, treat don't cares as a 1, but ignore them when actually selecting prime implicants. Never select a prime implicant only to cover a don't care.
- Problem 14.1.4: Draw the 5-variable K-Map $F(A,B,C,D,E) = \sum(1,9,10,17,21,27) + d(3,5,11,25,26)$.



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

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

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

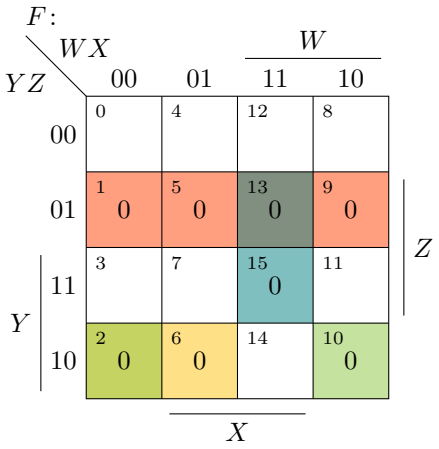


Distinguished 1's: 1, 6
Essential prime implicant(s): $\overline{A}D, \overline{A}BC$
Answer: $F = \overline{A}D + \overline{A}BC$

15 2/28/2020

15.1 Lecture Slides

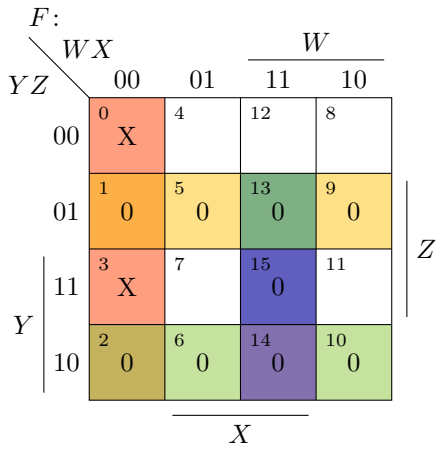
- Thus far, we have been focusing on minimization with SOP, but what about POS? POS minimization is important to understand for considering other designs in the future. It has similar rules to SOP minimization, but flipped.
- Recall the "recipe" for SOP minimization from page 42. We will now construct 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): $(W + \overline{Y} + Z), (X + \overline{Y} + Z), (Y + \overline{Z}), (\overline{W} + \overline{X} + \overline{Y})$

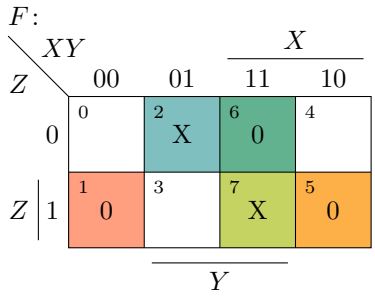
Answer: $F = (W + \overline{Y} + Z)(X + \overline{Y} + Z)(Y + \overline{Z})(\overline{W} + \overline{X} + \overline{Y})$

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



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

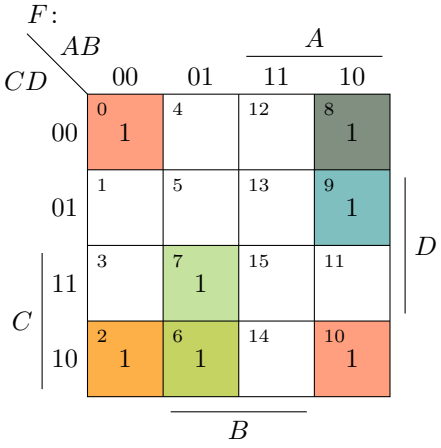
- Problem 15.1.3: Find the simplified POS for $F = \prod_{x,y,z}(1, 5, 6) + d(2, 7)$.



Distinguished 0's: 1
Essential prime implicant(s): $(Y + \overline{Z})$
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. $F = \overline{A}B\overline{C}\overline{D}+\overline{A}BC\overline{D}+\overline{A}B\overline{D}+\overline{A}BC+\overline{A}BCD+A\overline{B}\overline{C}D+A\overline{B}\overline{C}+A\overline{B}\overline{D}$.



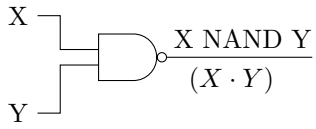
Distinguished 1's: 0, 7, 10, 10
Essential prime implicant(s): $A\overline{B}\overline{C}, \overline{B}\overline{D}, \overline{A}BC$

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

16 3/02/2020

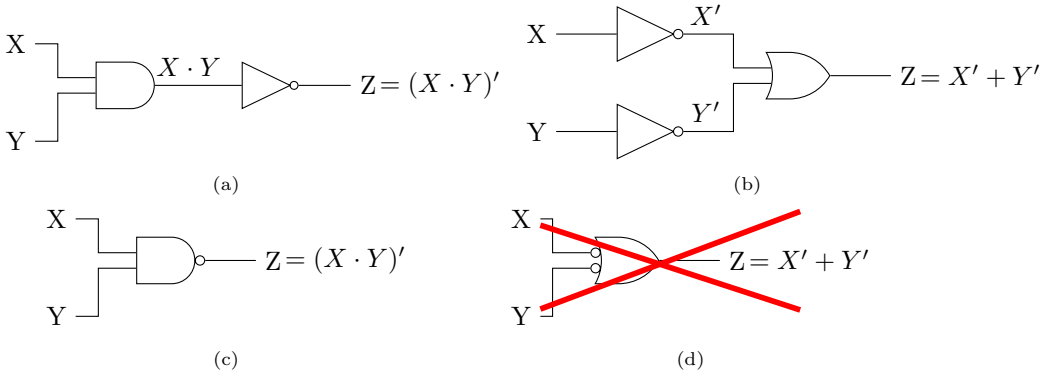
16.1 Lecture Slides

- 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.

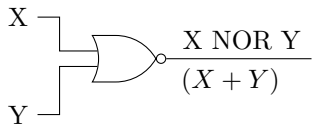


X	Y	X NAND Y
0	0	1
0	1	1
1	0	1
1	0	0

- Below are some other perspectives of a NAND gate.

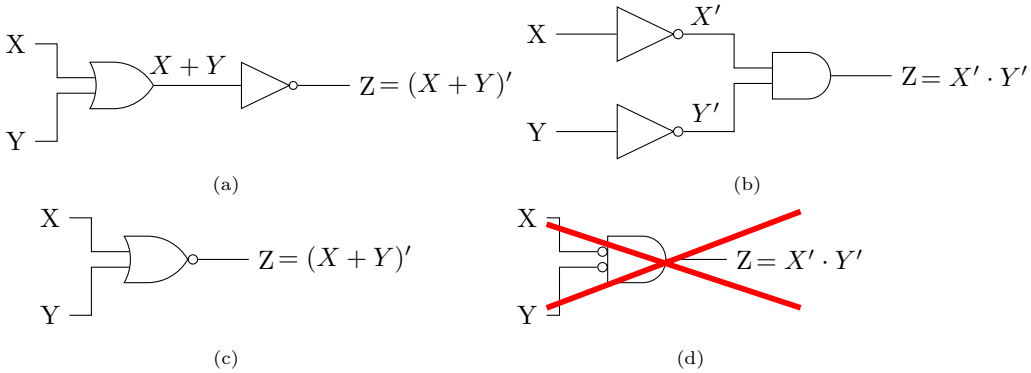


- The next gate is the NOR gate, or a Not OR gate.

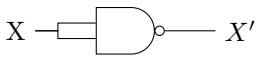


X	Y	X NAND Y
0	0	1
0	1	0
1	0	0
1	0	0

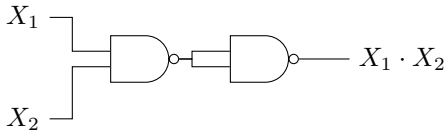
- Below are some other perspectives of a NOR gate.



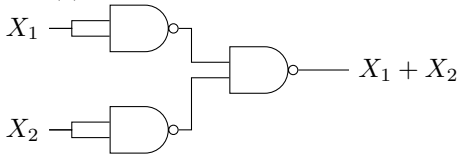
- Why do we use these new gates? The answer is simple: These are complete sets. One gate can be used to create a complete logic gate set (AND, OR, and NOT). Not having to mess around with other types of configurations makes setup much easier.
- In the end, it's all about mapping.



(a) Replacing a NOT with a NAND

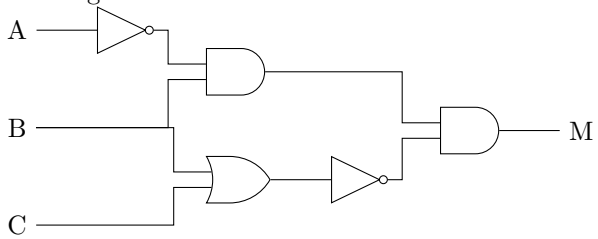


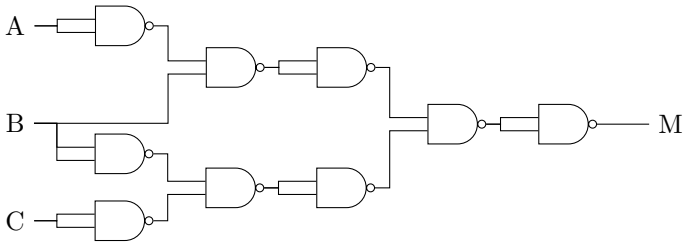
(b) Replacing an AND with a NAND



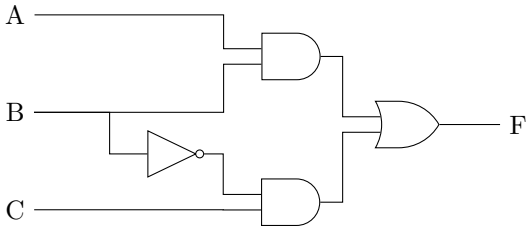
(c) Replacing an OR with a NAND

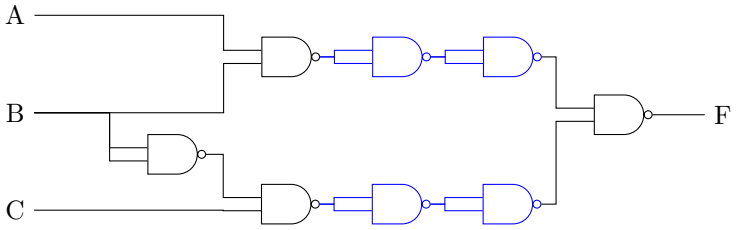
- So why does this work? In short, we are adapting DeMorgan's Law and the Involution Theorems.
- Problem 16.1.1: Convert the given logic diagram into one only using NAND gates.



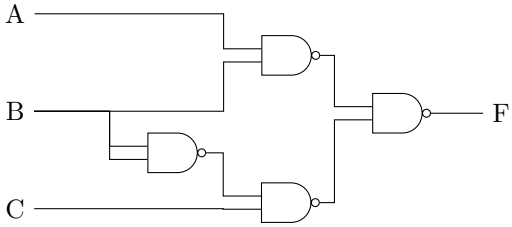


- Problem 16.1.2: Convert the given logic diagram into one only using NAND gates.

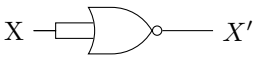




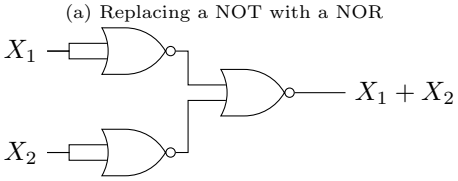
Using involution, we can cross out the blue elements, giving us...



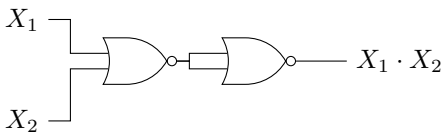
- We also have mappings for NOR gate conversions.



(a) Replacing a NOT with a NOR

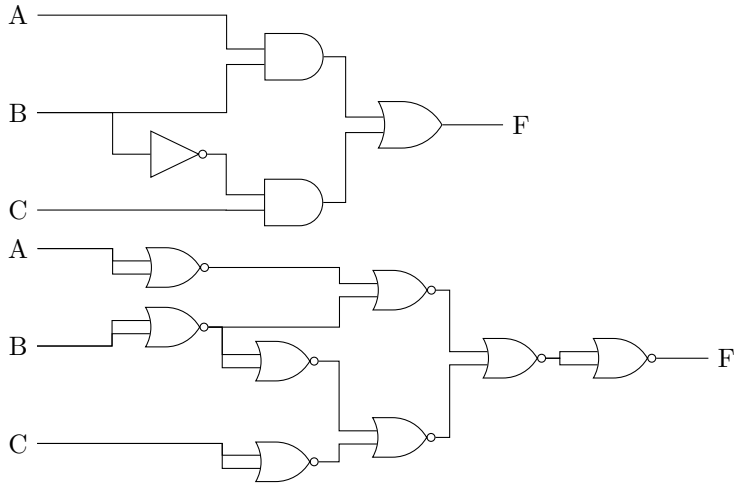


(b) Replacing an AND with a NOR

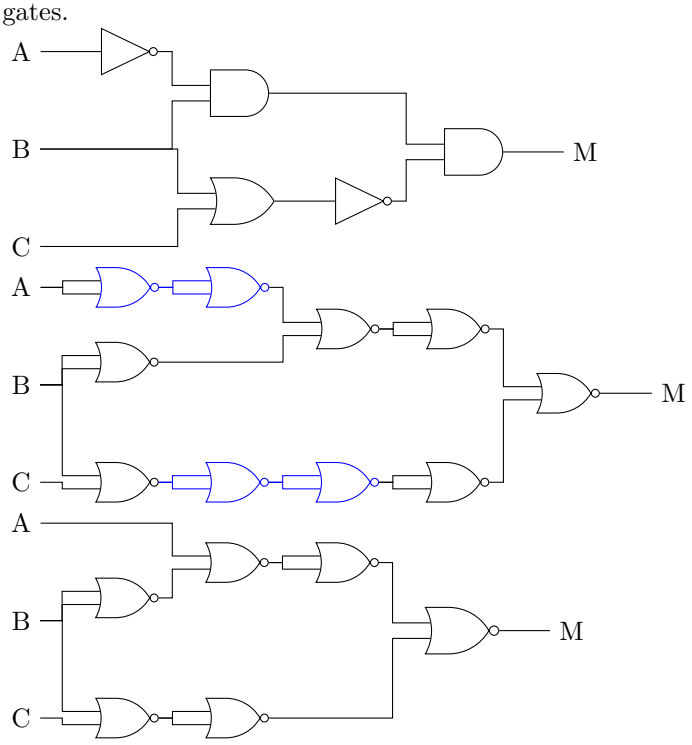


(a) Replacing an OR with a NAND

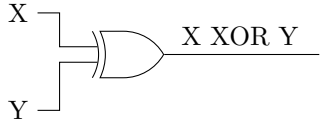
- Problem 16.1.3: Convert the given logic diagram into one only using NOR gates.



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

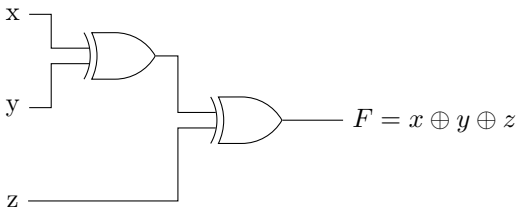


- Another new gate is the XOR gate.

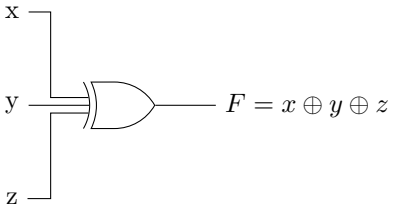


X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

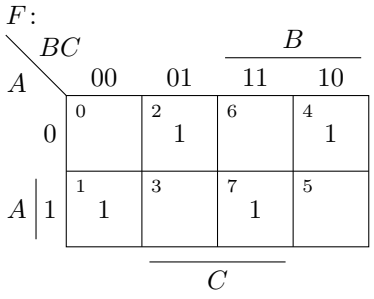
- It can be tricky to recognize when XOR can be used. A general rule of thumb is not to use this type of gate when told to only use specific gates. You can use truth tables, arithmetic, k-maps, and inspection to otherwise find when to use the XOR gate.



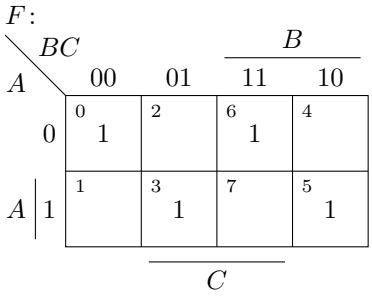
(a) Using 2-input gates



(b) Using 3-input gate

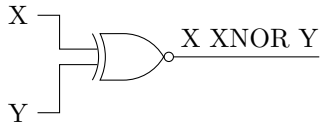


(a) Odd function $F = A \oplus B \oplus C$



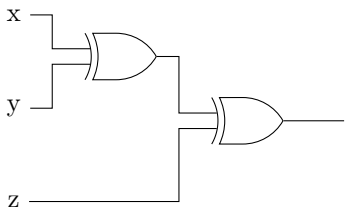
(b) Even function $F = (A \oplus B \oplus C)'$

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

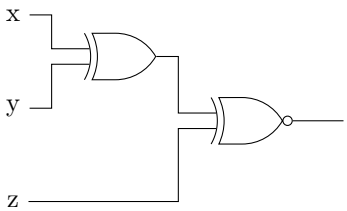


X	Y	X XNOR Y
0	0	1
0	1	0
1	0	0
1	1	1

- 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:

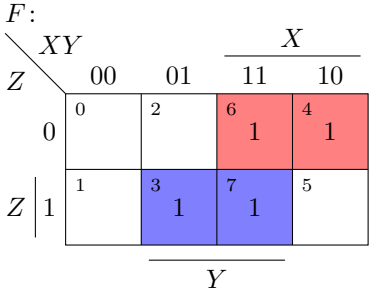
  > "A static-0 hazard is a pair of input combinations that: (a) differ in only one input variable and (b) both give a 0 output; such that is possible for a momentary 1 output to occur during a transition in the differing input variable."

- 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.

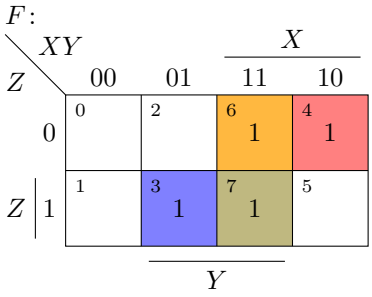
- A properly designed two-level SOP (AND-OR) circuit has no static-0 hazards. A static-0 hazard would only exist in the circuit if both a variable and its complement were connected to the same AND gate. However, these can have static-1 hazards.

- 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.

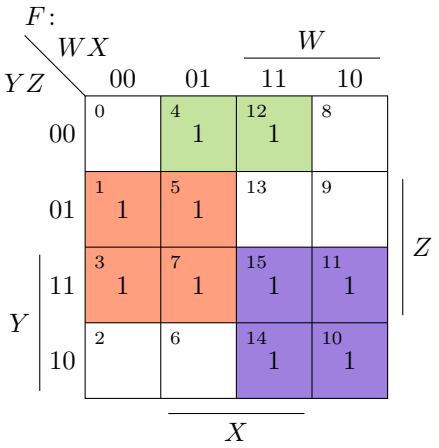


  It should be immediately obvious that there is no single product term that can cover both combinations 111 and 110. Thus, it is theoretically possible for the output to momentarily "glitch" to 0 if the AND gate output that covers one of the combinations goes to 0 before the other can go to 1. This is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

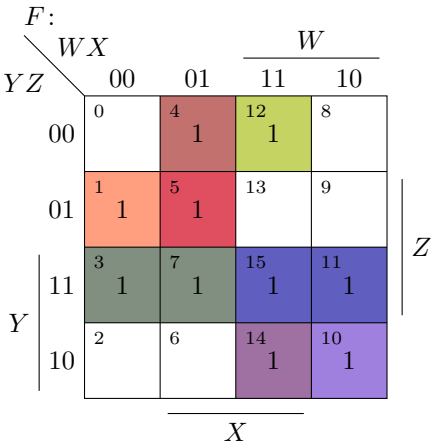


The extra product term is called the consensus, which is what we add to eliminate hazards.

- 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.



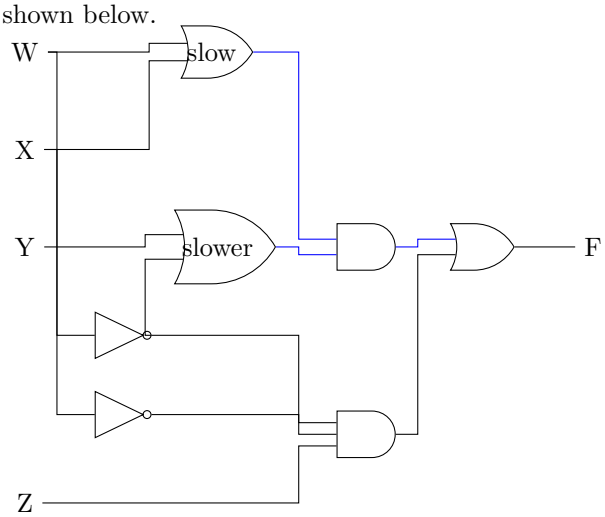
(a) As originally designed



(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.

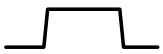
- Everything that has been said about AND-OR circuits applies to NAND-NAND circuits, and everything said about OR-AND circuits applies to NOR-NOR circuits.

- 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.

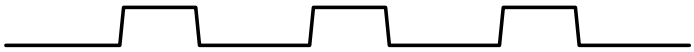
17 3/5/2020

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.

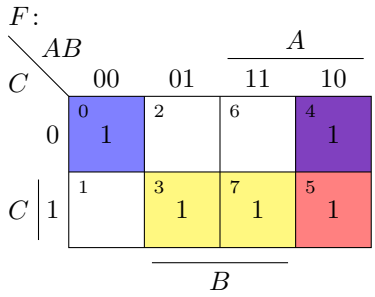


- While we do operate in 0 and 1, the values always exist and are continuous. Timing diagrams are what we actually work with and show behaviors such as verifying functionality and time impacts.

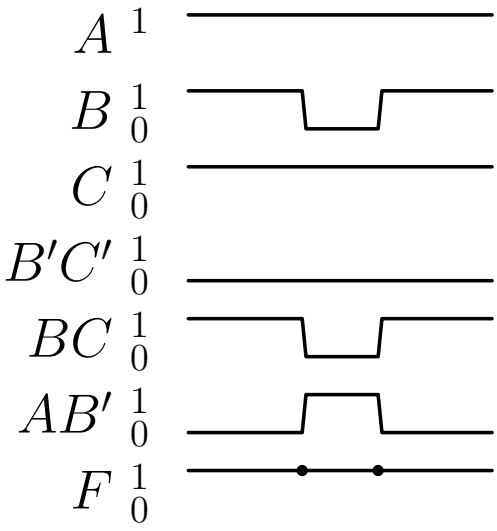


- Three variables aren't enough to draw a timing diagram for a function such as $F = B'C' + BC + AB'$. Those three variables only have eight possible combinations, which isn't enough. To draw the timing diagram, we need to look at all of the possible combinations.
- Determining all of the potential transitions 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.

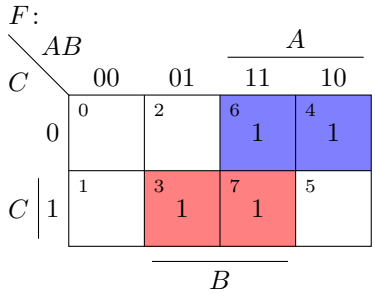
- Problem 17.1.1: Draw the timing diagram for $F = B'C' + BC + AB'$.

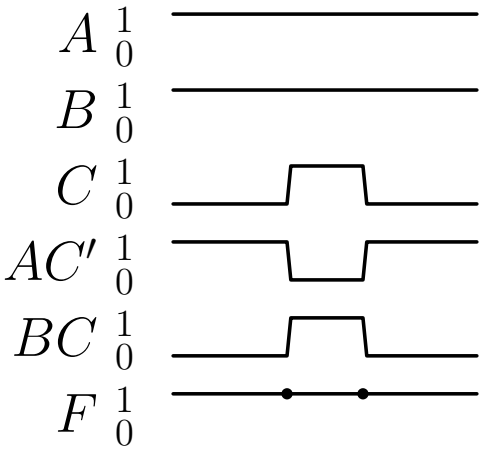


The transition between 111 $\leftrightarrow$ 101 is left uncovered (Note: Yes, you can draw implicants connecting these two but here she only drew the original function)!



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



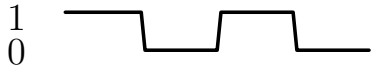


- Transition times in gates and other electronical technologies result in hazards. Let's first go over static hazards, which exist when two adjacent one's in a K-Map are not covered by a prime implicant in the resulting minimal function. To fix it, include additional prime implicants whenever there are 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) $1 \to 1$

(b) $0 \to 0$

(a) Static hazard

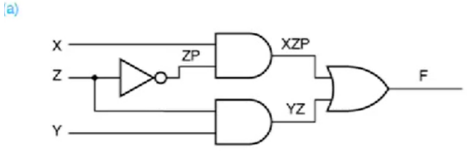


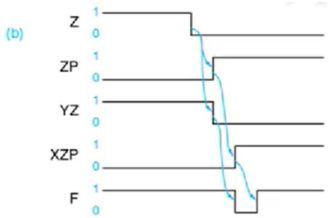
(a) $1 \to 0$

(b) $0 \to 1$

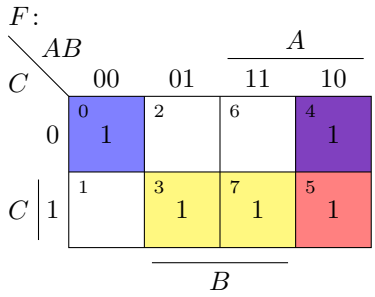
(b) Dynamic hazard

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

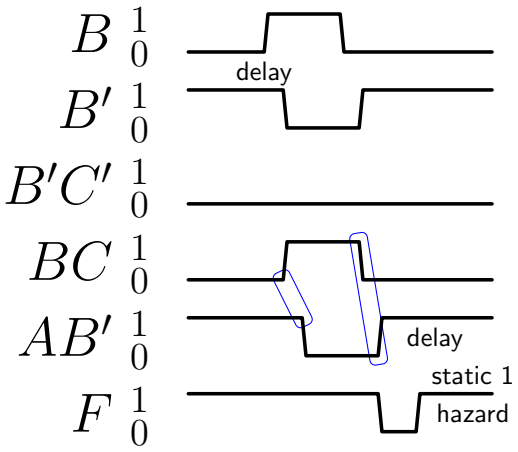




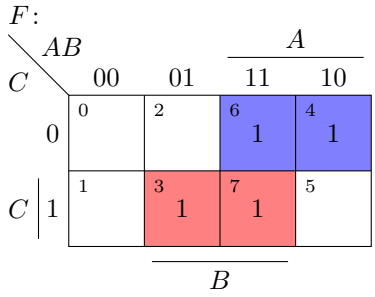
- Problem 17.1.2: Identify the hazard for $F = B'C' + BC + AB'$.

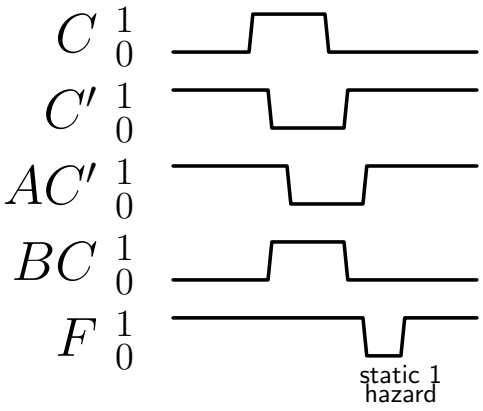


If you specify that B is the only factor (it is the only thing that changes in $111 \leftrightarrow 101$), you don't have to draw A and B in the timing diagram.

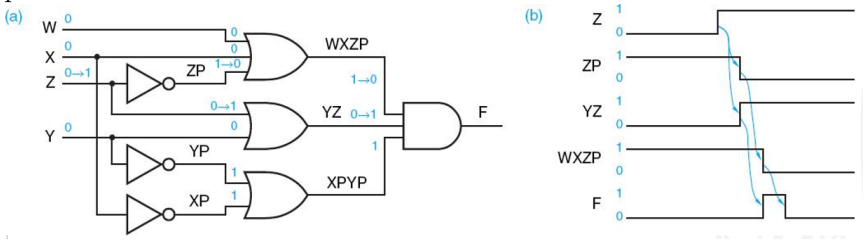


- Problem 17.1.4: Identify the hazard for $F = AC' + BC$.





- 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.

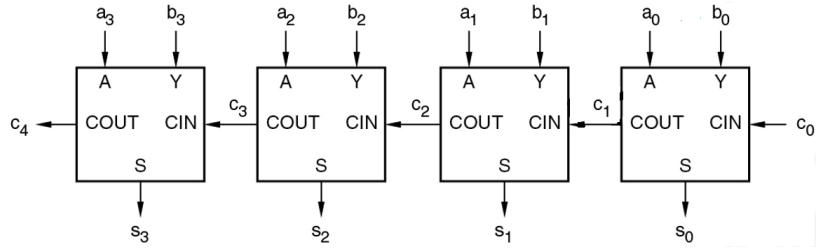
$$\begin{aligned} HS &= A \oplus B \\ &= A \cdot B' + A' \cdot B \\ CO &= A \cdot B \end{aligned}$$

- 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 most 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:

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

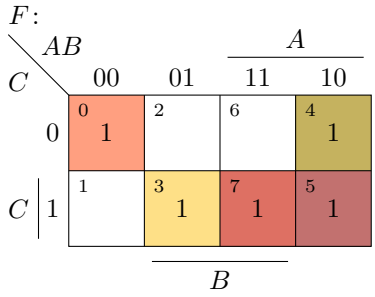
- Any n-bit adder circuit can function as a subtractor by complementing the subtrahend and treating the carry-in and carry-out signals as borrows with the opposite active level.

- 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 eveh 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.

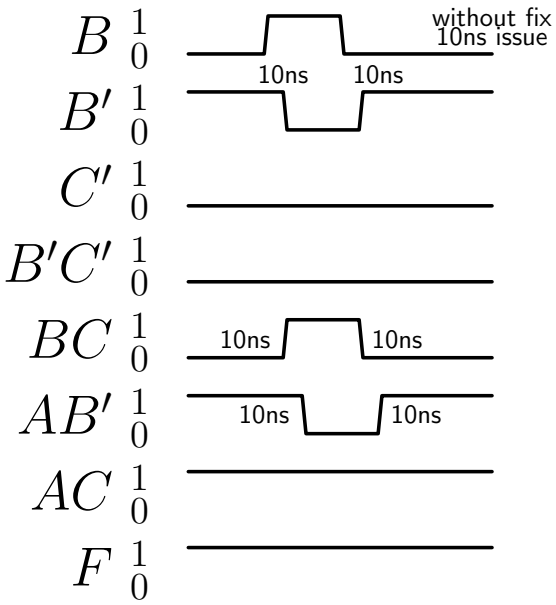
18 3/6/2020

18.1 Lecture Readings

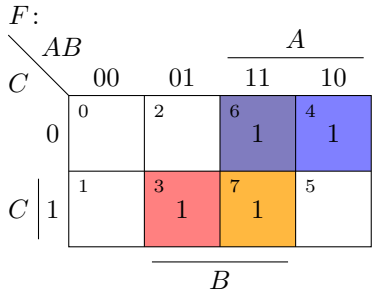
- Problem 18.1.1: Fix the hazard for $F = B'C' + BC + AB'$.



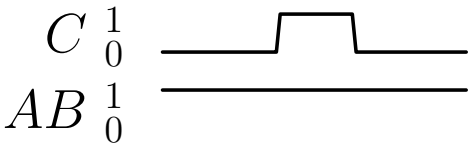
The hazard occurs at $111 \leftrightarrow 101$, so we can add AC to fix the hazard.



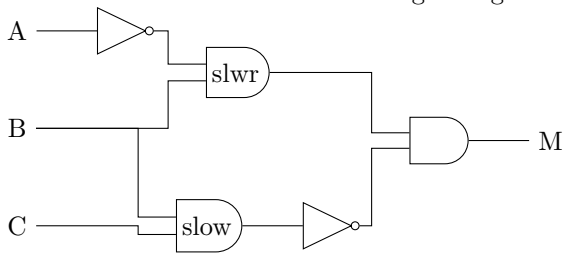
- Problem 18.1.2: Fix the hazard for $F = AC' + BC$.



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.

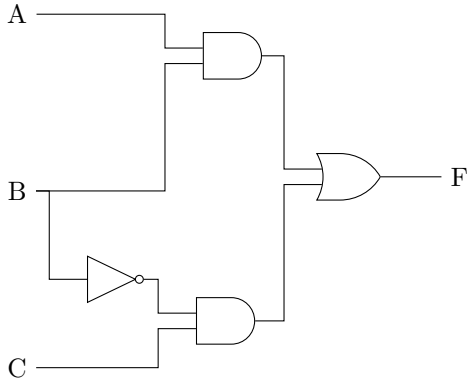


- Dynamic hazards occur when the output is supposed to change but oscillates for a brief period of time before settling down. These occur because of race conditions. We identify these through timing diagrams, logic gate inspections, and analysis.
- Problem 18.1.3: Does the below logic diagram have a 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 ($A \rightarrow$ and $\rightarrow$ or $\rightarrow F$) goes through only one logic gate, the and gate. The second path ($B \rightarrow$ not $\rightarrow$ and $\rightarrow$ or $\rightarrow F$), however, goes through two, which will introduce greater delay and thus brings a race condition.

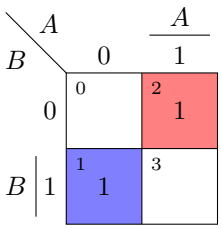
- So how do we solve these dynamic hazards? It's complicated. You should first try a two level design, described below. If that doesn't work, you can also try clocked synchronization.
  - If SOP: Check your not gates $\rightarrow$ and gates $\rightarrow$ or gates.
  - If POS: Check your not gates $\rightarrow$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

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

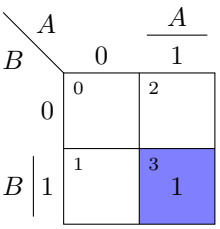
- MSI stands for medium scale integration and involves from 20 to about 200 logic gates. Our AND gate IC has four gates, making it SSI. MSI is the direction of complexity we will soon be approaching.
- The MSI combinational logic devices we will be looking at are adders, subtractors, multiplexors, decoders, 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.