CSE241: Digital Systems

Ben Miller

Instructed by: Jennifer Winikus

Contents

1 1/27/2020 **5**
- 1.1 Lecture Notes . . . 5
- 1.2 Assigned Readings . . . 5

2 1/29/2020 **7**
- 2.1 Lecture Notes . . . 7
- 2.2 Assigned Readings . . . 8

3 1/31/2020 **9**
- 3.1 Lecture Notes . . . 9
- 3.2 Assigned Readingss . . . 10

4 2/3/2020 **13**
- 4.1 Lecture Slides . . . 13
- 4.2 Assigned Readings . . . 16

5 2/5/2020 **18**
- 5.1 Lecture Slides . . . 18
- 5.2 Assigned Readings . . . 20

6 2/7/2020 **22**
- 6.1 Lecture Notes . . . 22
- 6.2 Assigned Readings . . . 23

7 2/10/2020 **27**
- 7.1 Lecture Notes . . . 27

8 2/12/2020 **29**
- 8.1 Lecture Notes . . . 29
- 8.2 Assigned Readings . . . 31

9 2/14/2020 **34**
- 9.1 Lecture Slides . . . 34
- 9.2 Assigned Readings . . . 36

10 2/17/2020 **38**
- 10.1 Lecture Slides . . . 38

11 2/19/2020 **40**
- 11.1 Lecture Slides . . . 40

12 2/21/2020 **42**
- 12.1 Lecture Slides . . . 42
- 12.2 Assigned Readings . . . 45

13 2/24/2020 **50**
- 13.1 Lecture Slides . . . 50

14 2/26/2020 **53**
- 14.1 Lecture Slides . . . 53

15 2/28/2020 **56**
15.1 Lecture Slides 56

16 3/02/2020 **59**
16.1 Lecture Slides 59
16.2 Assigned Readings 64

17 3/04/2020 **68**
17.1 Lecture Slides 68
17.2 Assigned Readings 72

18 3/06/2020 **75**
18.1 Lecture Readings 75

1 1/27/2020

1.1 Lecture Notes

- Don't cheat.

1.2 Assigned Readings

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

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

- 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 in exactly one row of a truth table. Simiarly, a maxterm defined as a sum term that is 0 in exactly one row of a truth table.

- An n-variable minterm can be represented by an n-bit integer, the minterm number. Syntactically, the name minterm i is used to denote the minterm corresponding to row i of the truth table. In minterm i, a particular variable appears complemented if the corresponding bit in the binary representation of i is 0, otherwise it is uncomplemented. A maxterm i is the opposite, with a variable being complemented if the corresponding binary bit i is 1.

- The canonical sum of a logic function is a sum of the minterms corresponding to the truth-table rows for which the function produces a 1 output.

- The canonical product of a logic function is a product of the maxterms corresponding to input combinations for which the function produces a 0 output.

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

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

- 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.

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 $\cdot \to +$ and $+ \to \cdot$. There are no zeros or ones so we don't need to worry about swapping those.
  Answer: Dual of $Y = X \cdot (W' + T) \cdot (W + T')$

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

- With canonical representation, every term in our equation contains every input.

- In SOP, or Sum of Products, each term is the product of the literals and they are all summed together. In a truth table, if the input for a literal for that minterm is a 1, then the literal is itself. Conversely, if the input for a literal for that minterm is a 0, then the literal is a complement of itself. Since SOP is a sum, we represent it with $\sum_{\text{variables in the function}}$.

- POS, or Product of Sums, is when each term is summed together and then is taken as the product. In a truth table, if the input for a literal for that maxterm is a 1, then the literal is a complement of itself. Conversely, if the input for a literal for that maxterm is a 0, then the literal is the complement of itself. We use the capital greek letter pi, or $\prod$, to represent POS.

- Because functions can get long, we use shorthand to make writing them more manageable. When writing a function, just write which minterm or maxterm numbers to include. Furthermore, use $\sum$ for SOP (also known as sigma notation) and $\prod$ for POS (also known as pi notation).

- Problem 9.1.3: Create the truth table for $F = \sum(2, 4, 6, 7)$.

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

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

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

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

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

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

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

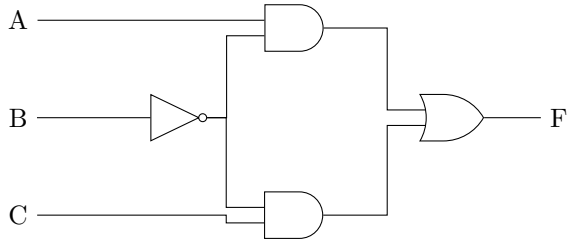
- On-set and off-set is a system used to describe the shorthand list of items. Minterms in shorthand have a list of the rows with 1 as the output, making it on-set. The complementary idea also holds true, with off-set being for maxterms.

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

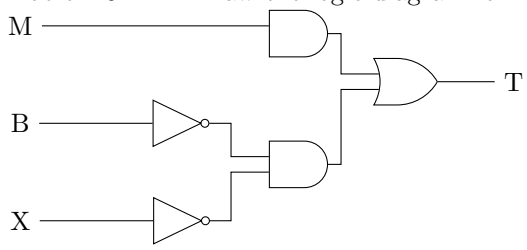
- Logic diagrams are drawings that help with design and preparation for implementation. For now, we will only use computational logic.

- 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 Readings

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

- A logic function described in this way can be designed directly from a given canonical sum or produce expression. For the above prime-number detector, we would have...
  $F = \sum_{N_3,N_2,N_1,N_0}(1,2,3,5,7,11,13)$
  $= N_3' \cdot N_2' \cdot N_1' \cdot N_0 + N_3' \cdot N_2' \cdot N_1 \cdot N_0' + N_3' \cdot N_2' \cdot N_1 \cdot N_0 + N_3' \cdot N_2 \cdot N_1' \cdot N_0 + N_3' \cdot N_2 \cdot N_1 \cdot N_0 + N_3 \cdot N_2' \cdot N_1 \cdot N_0 + N_3 \cdot N_2 \cdot N_1' \cdot N_0$

- 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.

  > ALARM = PANIC + ENABLE $\cdot$ EXITING$'$ $\cdot$ SECURE$'$
  > SECURE = WINDOW $\cdot$ DOOR $\cdot$ GARAGE
  > ALARM = PANIC+ENABLE$\cdot$EXITING$'\cdot$(WINDOW$\cdot$DOOR$\cdot$ 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.

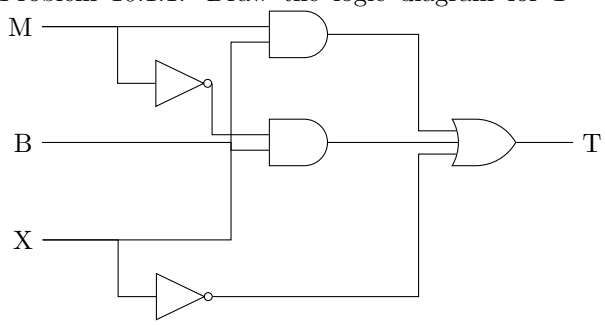
- In general, when we are designing a logic function for an application, it is easier to describe the logic function in words using logical connectives than it is to write a complete truth table (especially if the number of variables is large).

- 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.

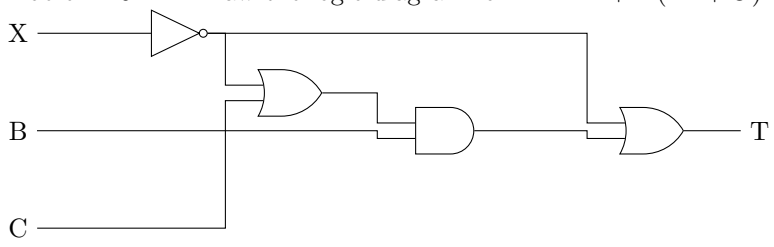
10 2/17/2020

10.1 Lecture Slides

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

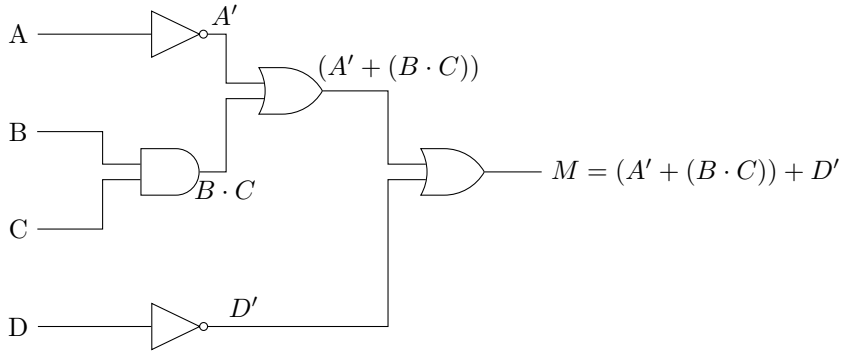


- Problem 10.1.2: Draw the logic diagram for $T = X' + B(X' + C)$.

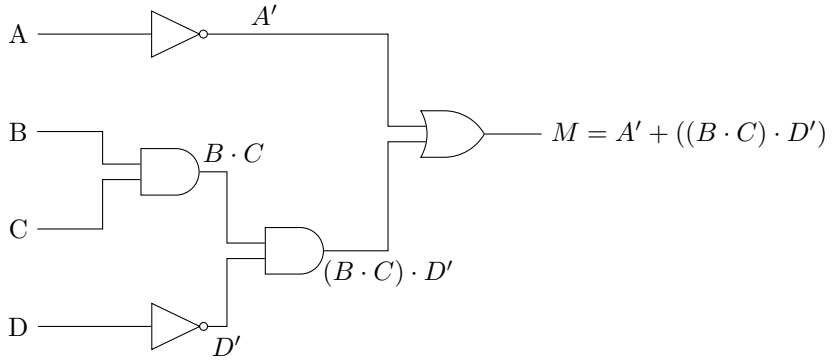


- You can always follow the path of connections in a diagram to see the behavior that it is implementing. This is done by labeling the function after each gate. See the below problems for examples.

- 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.

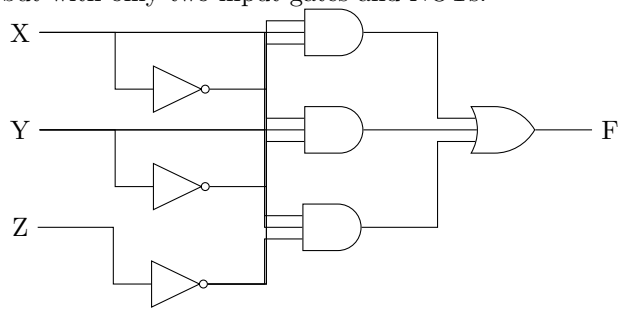


- Combinational logic synthesis is the process that specifies the required function and creates the details for implementation. Combinational logic design is the broader overview of the entire process, which includes logic synthesis.

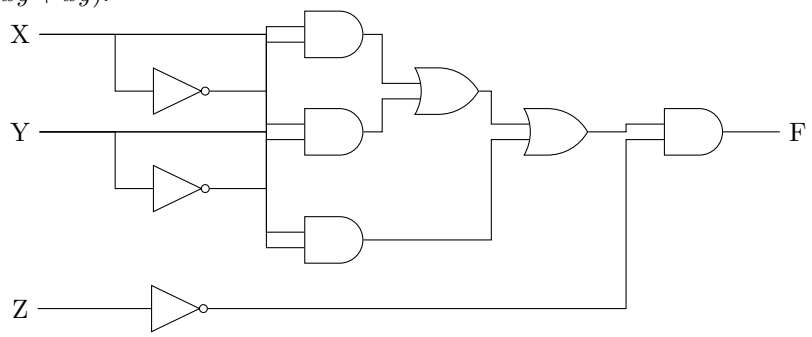
11 2/19/2020

11.1 Lecture Slides

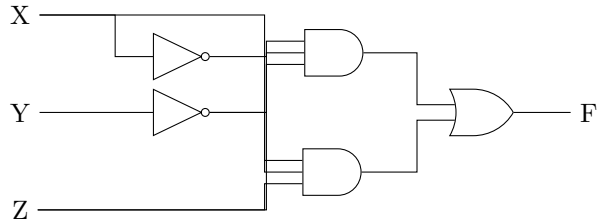
- A manipulator is a fancy way of saying that we can make an equivalent function with different arrangements.
- Problem 11.1.1: Draw the canonical logic diagram for $F = \sum_{x,y,z}(2, 4, 6)$ but with only two input gates and NOTs.



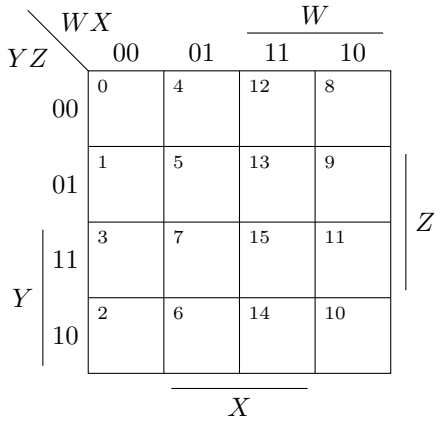
This is a two level, three input gate $F = \overline{x}y\overline{z} + x\overline{yz} + xy\overline{z}$. Using the associative property, we can create a multi-level two input gate $F = \overline{z}(\overline{x}y + x\overline{y} + xy)$.



- 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.



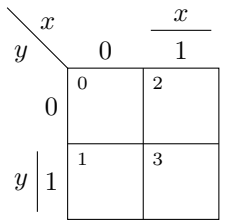
This above logic diagram is for $F = \overline{xy}z + x\overline{y}z$ and is incorrect. The below correct logic diagram takes advantage of the distributive property. $F = \overline{x}(\overline{y}z) + x(\overline{y}z)$.



- 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?

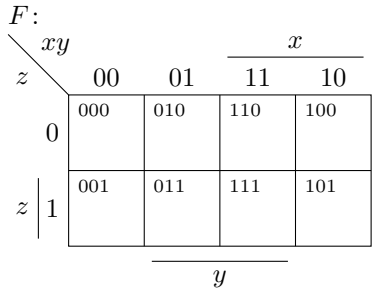
x	y	F
0	0	A
0	1	B
1	0	C
1	1	D

F:



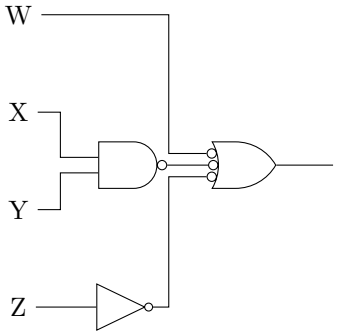
- Problem 12.1.3: How does the below truth table map to a K-Map?

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



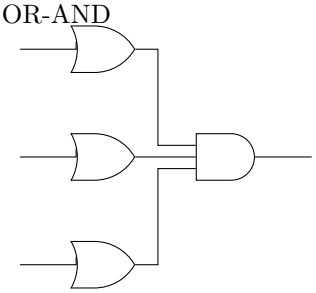
- To use the K-Map for reduction, some vocabulary terms need to get defined.
  - Implicants (or a covers) are power of 2 circles/rectangles that go around neighboring 1's in SOP and 0's in POS. The prime implicant, or PI, is the largest cover possible.
  - Distinguished 1's are an easy way of checking 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.
  $F = \overline{x}\overline{y}\overline{z} + x\overline{y}\overline{z} + x\overline{y}z + xy\overline{z} + xyz$
  $F = \overline{y}\overline{z}(\overline{x} + \cancel{x^1}) + x\overline{y}z + xy(\overline{z} + \cancel{z^1})$
  $F = \overline{y}\overline{z} + x\overline{y}z + xy$
  Answer: $F = \overline{y}\overline{z} + x\overline{y}z + xy$

c. NAND-NAND

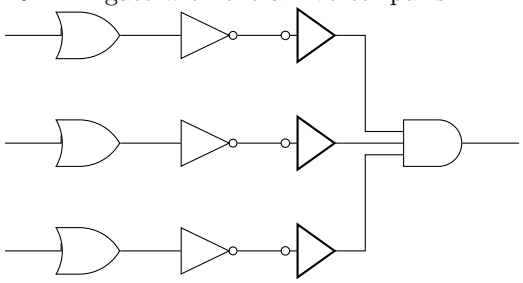


- The dual of this idea also holds true, in that any product-of-sums expression can be realized as an OR-AND circuit or as a NOR-NOR circuit. Below is a realization of a product-of-sums circuit.

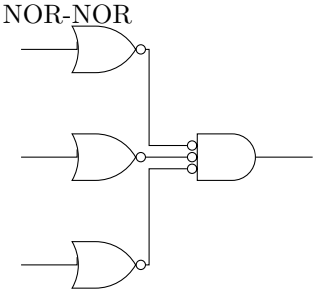
a. OR-AND



b. OR-AND gate with extra inverter pairs

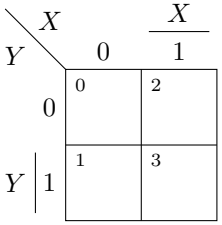


c. NOR-NOR

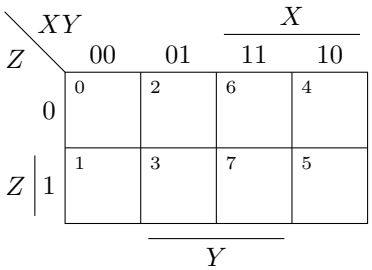


- Often, it is uneconomical or inefficient to realize a logic circuit directly from the first logic expression you think of. Canonical sum and product expressions are particularly expensive because the number of possible minterms and maxterms grows exponentially with the number of variables. We need 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.

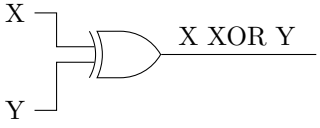
  (a) 2-variable



  (b) 3-variable

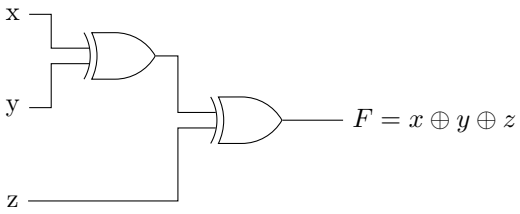


- Another new gate is the XOR gate.

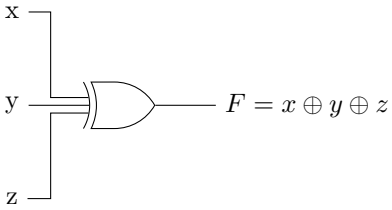


X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

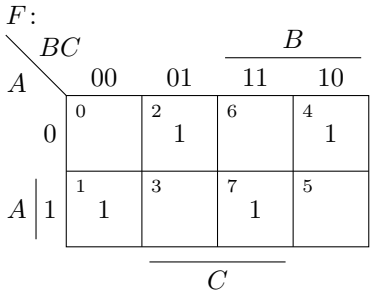
- It can be tricky to recognize when XOR can be used. A general rule of thumb is not to use this type of gate when told to only use specific gates. You can use truth tables, arithmetic, k-maps, and inspection to otherwise find when to use the XOR gate.



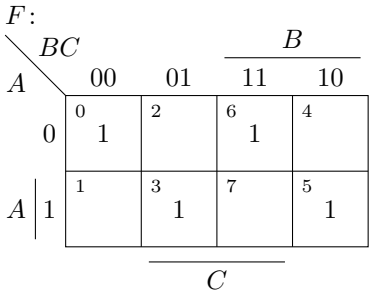
(a) Using 2-input gates



(b) Using 3-input gate

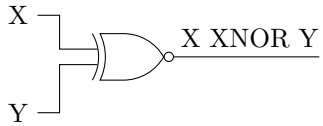


(a) Odd function $F = A \oplus B \oplus C$



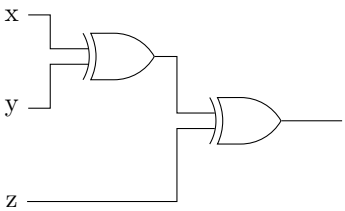
(b) Even function $F = (A \oplus B \oplus C)'$

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

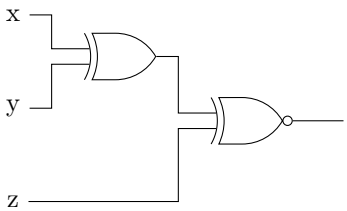


X	Y	X XNOR Y
0	0	1
0	1	0
1	0	0
1	1	1

- 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.

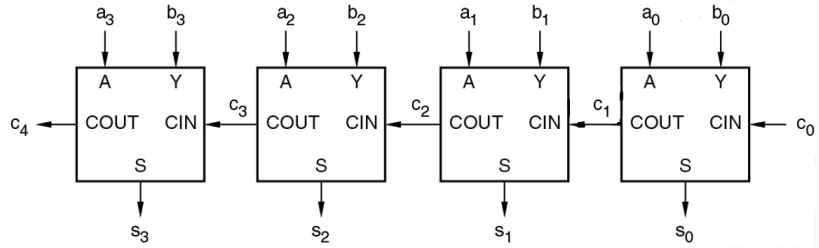
- Depending on how a circuit's output is produced, a system's operation might not even be adversely impacted by a glitch. When a glitch is harmful, however, it is up to the logic designer to be prepared to eliminate the hazards, or the possibilities of glitches occurring.

- 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.