CSE241: Digital Systems

Ben Miller

Instructed by: Jennifer Winikus

Contents

1 1/27/2020

1.1 Lecture Notes

- Don't cheat.

1.2 Assigned Readings

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

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

2 1/29/2020

2.1 Lecture Notes

- Analog signals can take any value across a continuous range of current, voltage, etc.
- While digital circuits can be analog too, they don't, because digital works better for their purpose. Digital signals restrict themselves to two discrete values: 0 and 1.
- Systems can be represented digitally. For example, consider an image, which is just thousands of pixels represented by bits.
- Analog signals are our physical reality. Things in analog signals are continuously variable. The design of an analog signal is extremely complex, and stability and accuracy is very difficult.
- In the beginning, we used vacuum tubes to go from analog to digital. However, due to vacuum tubes being inefficient, we eventually moved to transistors.
- Problem 2.1.1: Convert 37 to binary.

37/2	18	R1
18/2	9	R0
9/2	4	R1
4/2	2	R0
2/2	1	R0
1/2	0	R1

Answer: b101001

- Problem 2.1.2: Convert b10111 to decimal.
$1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$
$2^4 + 2^2 + 2^1 + 2^0$
$16 + 4 + 2 + 1$
Answer: 23
- In the hex number system, each group of four becomes a single hex value. For example, b10100111 is xA7.
- Problem: Convert b101110101 to hex.
  - First, start from the right and collect the four rightmost bits: 0101.
  - Then, grab the next four bits: 0111.
  - Finally, grab the remaining bit: 1. We need to put 0's in the front to pad this out to four bits, giving us 0001.
  - Convert these groups to values. 0001 is equivalent to 1, 0111 is equivalent to 7, and 0101 is equivalent to 5.

  Answer: x175

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

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

Answer: Proven. The red numbers signify overflow.

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

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

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

3.2 Assigned Readingss

- In the signed-magnitude system, a number consists of a magnitude and a symbol indicating whether the number is positive or negative. We assume that the sign is positive if no sign is specified. There are two representations of zero, "+0" and "-0", both with the same value.
- The signed-magnitude system is applied to binary numbers using an extra bit position used to represent the sign, called the sign bit. The most significant bit is typically used as the sign bit, with 0 representing a positive value and 1 representing a negative value.
  - $01010101_2 = +85_{10}$
  - $01111111_2 = +127_{10}$
  - $00000000_2 = +0_{10}$
  - $11010101_2 = -85_{10}$
  - $11111111_2 = -127_{10}$
  - $10000000_2 = -0_{10}$
- The signed-magnitude system has an equal number of positive and negative integers. An n-bit signed-magnitude integer lies within the range $-(2^{n-1} - 1)$ through $+(2^{n-1} - 1)$.
- While the signed-magnitude system negates a number by changing its sign, a complement number system negates a number by taking its complement as defined by the system. Taking a complement is more difficult than simply changing the sign, however two numbers in a complement system can be added or subtracted directly without the sign and magnitude checks that have to be done in the signed-magnitude system.
- In a two's complement system, the complement of an n-bit number B is obtained by subtracting it from 2^n. If B is between 1 and $2^n - 1$, thus subtracting produces another number between 1 and $2^n - 1$. If B is 0, the result of the subtraction is 2^n. Because of this, there is only one representation of zero in a two's complement system.

- An unnecessary subtraction operation can be avoided by rewriting 2^n as $(2^n - 1) + 1$ and $2^n - B$ as $((2^n - 1) - B) + 1$. For example, for $n = 8, 100000000_2$ equals $11111111_2 + 1$.

- If we define the complement of a bit b to be the opposite value of the bit, then $(2^n - 1) - B$ is obtained by simply complementing the bits of B. Therefore, the two's complement of a number B is obtained by complementing the number of individual bits in B and adding 1. Again, using $n = 8$ as an example, the two's complement of 01110100 is $10001011 + 1$, or 10001100.

- In the two's complement system, the MSB serves as the sign bit. A number is negative *if and only if* its MSB is 1.

- In a ones' complement system, the complement of an n-bit number B is obtained by subtracting it from $2^n - 1$. This is accomplished by complementing the individual digits of B without adding 1 like in the two's complement system. The MSB acts as the sign, with 0 being positive and 1 being negative. This gives two representations of zero, a positive zero and a negative zero.

- While positive number representations are the same for ones' and two's complement, negative numbers differ by one.

- A weight of $(2^{n-1} - 1)$, rather than -2^{n-1} is given to the most significant bit when computing the decimal equivalent of a ones' complement number.

- The main advantages a ones' complement system has is its symmetry and ease of complementation, given it usage in early computer. However, due to the adder design of ones' complement being more complicated and there being two representations of zero, two's complement is more widely used presently.

- In an excess-B representation, an m-bit string whose unsigned integer value is $M (0 \leq M \leq 2^m)$ represents the signed integer $M - B$, where B is the bias of the number system.

- For example, an excess-2^{m-1} system represents any number X in the range between -2^{m-1} through $+2^{m-1} - 1$. The range of this representation is exactly the same as that of m-bit two's complement numbers. In fact, the range of the two systems are identical with the sole difference being that the sign bits are always opposite. Excess representation is mostly used in floating point number systems.

- Because ordinary addition is just an extension of counting, two's complement numbers can be added using ordinary binary addition, ignoring any carries beyond the MSB. This result will always be accurate so long as the range of the number system is not exceeded.

- If an addition operation produces a result that exceeds the range of the number system, overflow is said to have occurred. Addition of two numbers with different signs will never produce overflow, but addition with like signs can, as shown below.

5.2 Assigned Readings

- A set of n-bit strings in which different bit strings represent different numbers or other things is called a code. A particular combination of n 1-bit values is called a code word.

- There may not necessarily be a mathematical relationship between a particular code word and what it is supposed to represent. Furthermore, a code using n-bit strings doesn't need to contain 2^n valid code words.

- Listed below are some common days the ten decimal digits are represented.

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

- The most natural of these is the binary-coded decimal system, or BCD. This system encodes the digits 0 through 9 by their 4-bit unsigned binary representations. Because of this, conversions between BCD and decimal representations are trivial.

- Some computer programs place two BCD digits into one 8-bit byte in the packed-BCD representation. In this system, one byte may represent the values from 0 to 99 versus 0 to 255 for a normal, unsigned 8-bit binary number. BCD numbers for any desired value can be obtained by using one byte for every two digits.

- Similary to binary numbers, there are many possible representations of negative BCD values. Signed BCD numbers have one extra digit position for the sign, and both the signed-magnitude and 10's complement representations are used in BCD arithmetic. In signed-magnitude BCD, the encoding of the sign bit string is arbitrary, while in 10's compliment, 0000 indicates a positive value and 1001 indicates a negative value.

- Addition of BCD digits function similarly to adding 4-bit unsigned numbers, with the sole difference being that a correction must be made if the result exceeds 1001, in which case the result is corrected by adding six.

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

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

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

- All axioms are stated as a pair. This is a characteristic of axioms in switching algebra called "duality."
- An inverter is a logic circuit whose output signal level is the opposite, or complement, of its input signal level. We use a prime tick ($'$) to denote an inverter function.
- This prime tick is an algebraic operator, and a statement such as X' is an expression.
- A 2-input AND gate is a circuit whose output is 1 if both of its inputs are 1. The function of a 2-input AND gate is sometimes called logical multiplication and is symbolized algebraically by a multiplication dot ($\cdot$). Some mathematicians and logicians use the wedge ($\wedge$) to denote logical multiplication.
- A 2-input OR gate is a circuit whose output is 1 if either of its inputs are 1. The function of a 2-input AND gate is sometimes called logical addition and is symbolized algebraically by a plus sign ($+$). Some mathematicians and logicians use the vee ($\vee$) to denote logical addition.
- By convention, logical multiplication has a higher precedence than logical addition.
- Switching algebra theorems are statements known to always be true that allow us to manipulate algebraic expressions for simpler analysis. For example, the theorem $X + 0 = X$ allows us to substitute every occurrence of $X + 0$ in an expression with just X. A list of theorems involving one variable are shown below.

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

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

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

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

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

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

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

- The information obtained in a truth table can also be conveyed algebraically. To do so, a few terms must be defined.
  - A literal is a variable or the complement of a variable. Examples: X, Y, X′, Y′.
  - A product term is a single literal or a logical product of two or more literals. Examples: $Z', W \cdot X \cdot Y, X \cdot Y' \cdot Z, W' \cdot Y' \cdot Z$.
  - A sum-of-product expression is a logical sum of product terms. Examples: $Z' + W \cdot X \cdot Y + X \cdot Y' \cdot Z + W' \cdot Y' \cdot Z$.
  - A sum term is a single literal of a logical sum of two or more literals. Examples: $Z', W + X + Y, X + Y' + Z, W' + Y' + Z$.
  - A normal term is a product or sum term in which no variable appears more than once. A non-normal term can always be simplified to a constant or a normal term using one of theorems T3, T3′, T5, or T5′. Examples of non-normal terms: $W \cdot X \cdot X \cdot Y', W + W + X' + Y, X \cdot X' \cdot Y$. Examples of normal terms: $W \cdot X \cdot Y', W + X' + Y$.

- Associativity is the idea that the order in which segments of an equation are executed do not matter.
- Covering refers to some terms being excluded when a single term "covers" all possible cases.
- Redundancy is a simplification trick when simplifying algebraic expressions.
- General idempotency is simply the idea of applying impotence repeatedly.
  - $x = x * x = x * x * x * x * x$
  - $x = x + x = x + x + x + x + x$
- Problem 7.1.3: Simplify $T = Y' * F + X * (X' + Y)$.
  - Distributive property: $T = Y' * F + X * X' + X * Y$
  - Identity property: $T = Y' * F + 0 + X * Y$
  - Simplify: $T = Y' * F + X * Y$

  $\boxed{\text{Answer: } T = Y' * F + X * Y}$
- : Problem 7.1.4: Simplify $Y = X * X * T + T * T + X$.
  - Idempotency: $Y = X * T + T + X$
  - Distributive property: $X(T + 1) + T$
  - Simplification: $X(1) + T$
  - Simplification: $X + T$

  $\boxed{\text{Answer: } Y = X + T}$
- Augustus DeMorgan came up with a theorem that describes factoring and distributing inverted functions. His theorems are listed below.
  - $(x * y)' = x' + y'$
  - $(x + y)' = x' * y'$
  - $x + x' * y = x + y$
  - $x * (x' + y) = x * y$
- Problem 7.1.5: Simplify $T = X * (Y + X)' + Y$.
  - DeMorgan's Law: $T = X * (Y' * X) + Y$
  - Distributive Law: $X * Y + X * X' + Y$
  - Simplification: $T = X * Y' + Y$
  - Simplification: $T = Y + X$

  $\boxed{\text{Answer: } T = Y + X}$
- The main idea behind Shannon's expansion theorem is configuring an expression to contain more variables than the hardware would normally allow.

8 2/12/2020

8.1 Lecture Notes

- Problem 8.1.1: Simplify $Y = X'T + (X + T)'$.
  Use DeMorgan's.
  $y = \overline{x}T + [\overline{x} \cdot \overline{T}]$
  $y = \overline{x}T + \overline{x}\overline{T}$
  $y = \overline{x}(T + \overline{T})$ (note that $T + \overline{T} = 1$)
  $y = \overline{x} \cdot 1$
  $y = \overline{x}$
  $\boxed{\text{Answer: } y = \overline{x}}$

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

- Problem 8.1.2: Find the simplified complement of $Y = XT + WT' + W'X$.
  $\overline{y} = \text{complement} = [XT + WT' + W'X]'$
  $(\overline{X} + \overline{T})(\overline{W} + \overline{\overline{T}})(\overline{\overline{W}} + \overline{X})$
  $(\overline{X} + \overline{T})(\overline{W} + T)(W + \overline{X})$
  $(\overline{X}\overline{W} + \overline{X}T + \overline{T}\overline{W} + \overline{T}T)(W + \overline{X})$
  $\overline{W}W\overline{X} + \overline{X}W\overline{X} + \overline{X}TW + \overline{X}T\overline{X} + \overline{T}\overline{W}W + \overline{T}\overline{W}\overline{X}$
  $\overline{X}W + \overline{X}TW + \overline{X}T + \overline{T}\overline{W}\overline{X}$
  $\overline{X}\overline{W}(1 + T) + \overline{X}WT + \overline{X}T$
  $\overline{X}\overline{W} + \overline{X}T(W + 1)$
  $\overline{y} = \overline{X}\overline{W} + \overline{X}T$
  $\boxed{\text{Answer: } \overline{y} = \overline{X}\overline{W} + \overline{X}T}$

- 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.

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

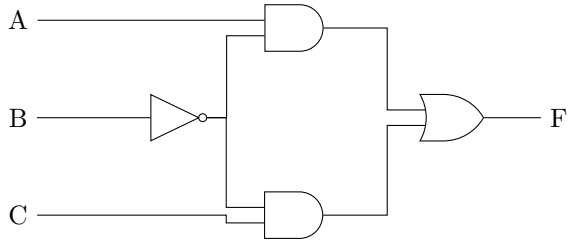
- On-set and off-set is a system used to describe the shorthand list of items. Minterms in shorthand have a list of the rows with 1 as the output, making it on-set. The complementary idea also holds true, with off-set being for maxterms.

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

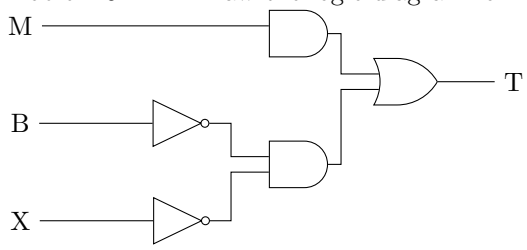
- Logic diagrams are drawings that help with design and preparation for implementation. For now, we will only use computational logic.

- 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.

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

- A circuit realizes an expression if its output function equals that expression. If a circuit realizes an expression, it is called a realization of the function. In place of realization, the term "implementation" is sometimes used. Recognize both - they are both used in practice.

- Once we have any expression, we can do more than just build a circuit from the expression. We can, for example, manipulate the expression to get different circuits. The aforementioned ALARM expression can be multiplied out to get a sum-of-products circuit, as an example. Alternatively, if the number of variables isn't very large, we can create a truth table for the expression.

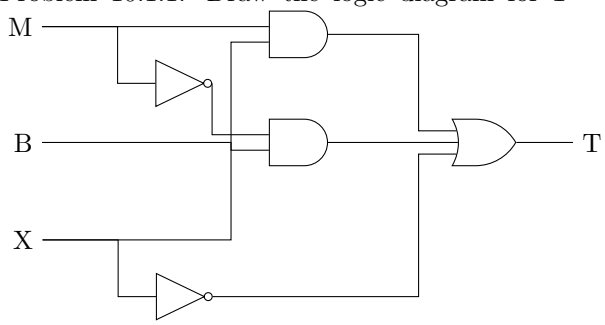
- In general, when we are designing a logic function for an application, it is easier to describe the logic function in words using logical connectives than it is to write a complete truth table (especially if the number of variables is large).

- Sometimes however we start with imprecise word descriptions of logic functions, such as the sentence "The ERROR output should be 1 if the GEARUP, GEARDOWN, and GEARCHECK inputs are inconsistent." In these cases, a truth-table approach is more optimal because it allows us to determine the output required for every input. Using a logic expression in this case might make it difficult to notice "corner cases" and handle them appropriately.

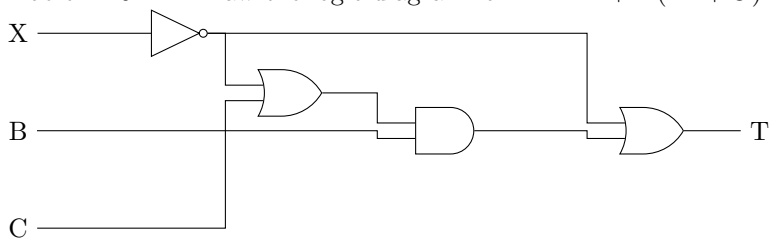
10 2/17/2020

10.1 Lecture Slides

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

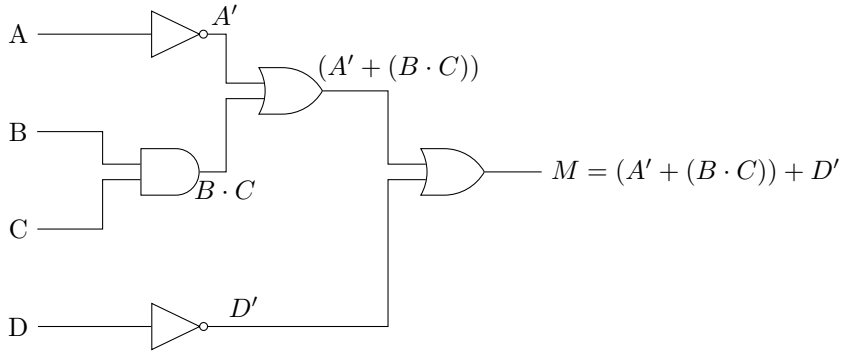


- Problem 10.1.2: Draw the logic diagram for $T = X' + B(X' + C)$.

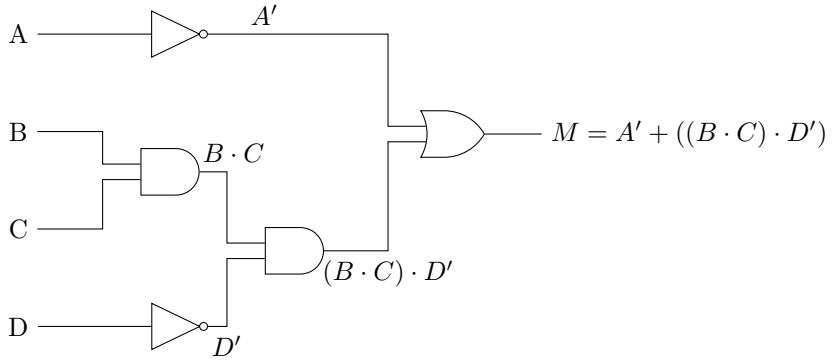


- You can always follow the path of connections in a diagram to see the behavior that it is implementing. This is done by labeling the function after each gate. See the below problems for examples.

- Problem 10.1.3: Develop the equation for the logic diagram. Do not simplify the solution.



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

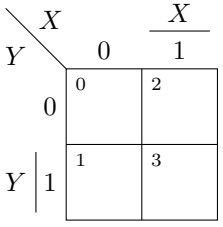


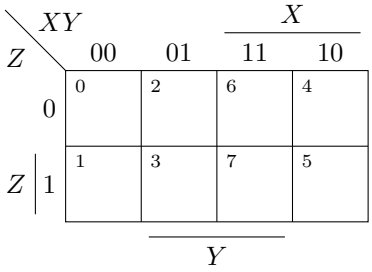
- Combinational logic synthesis is the process that specifies the required function and creates the details for implementation. Combinational logic design is the broader overview of the entire process, which includes logic synthesis.

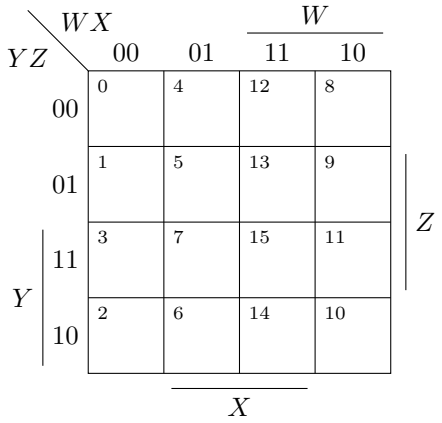
12 2/21/2020

12.1 Lecture Slides

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



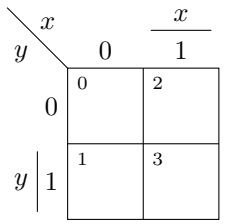




- There are a few things to note about K-Maps. First, working with more than four variables generally isn't worth it, at which computer aided design begins to be used. Second, the order of the numbers is in gray code.
- Why gray code? Take the two terms $A'BC + ABC$. If we apply the distributive property 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:

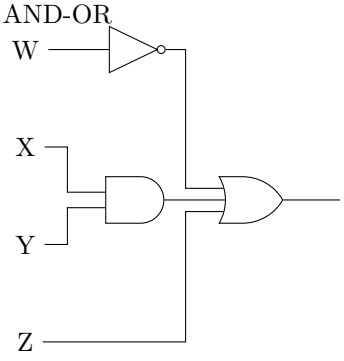


- Problem 12.1.3: How does the below truth table map to a K-Map?

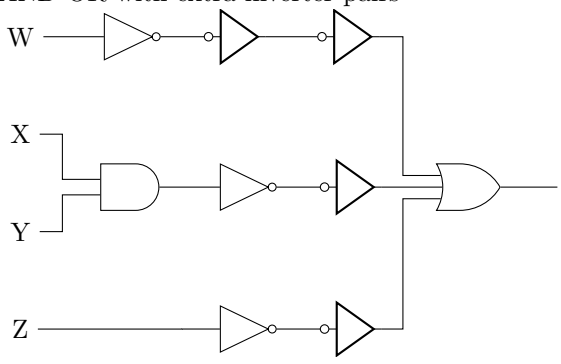
x	y	z	F
0	0	0	A
0	0	1	B
0	1	0	C
0	1	1	D
1	0	0	E
1	0	1	F
1	1	0	G
1	1	1	H

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

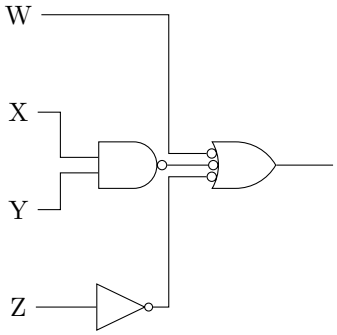
a. AND-OR



b. AND-OR with extra inverter pairs

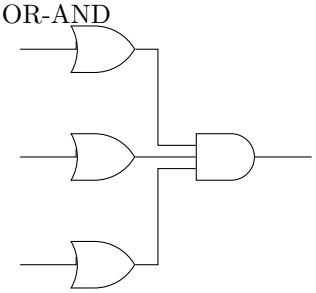


c. NAND-NAND

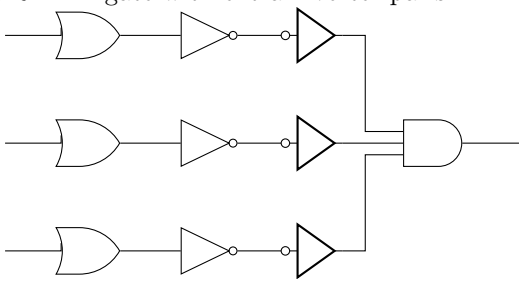


- The dual of this idea also holds true, in that any product-of-sums expression can be realized as an OR-AND circuit or as a NOR-NOR circuit. Below is a realization of a product-of-sums circuit.

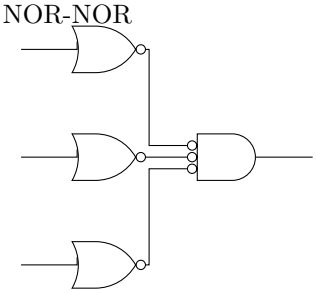
a. OR-AND



b. OR-AND gate with extra inverter pairs

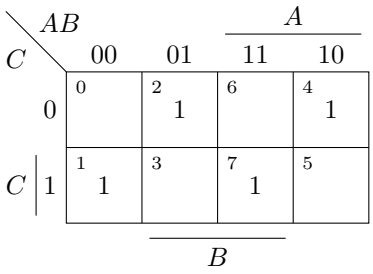


c. NOR-NOR

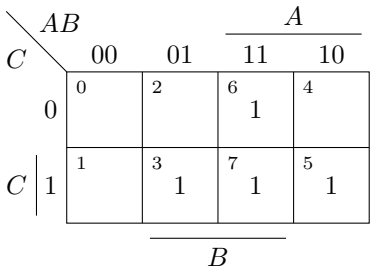


- So what does a full adder look like when implemented?

$A+B+C_{in}$				
A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

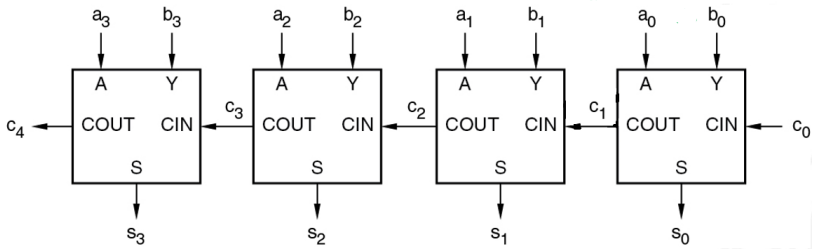


(a) S



(b) C

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



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