2022 Spring Term (1) Fundamentals of Computer Systems CSC 215 5000[3335] (Borough of Manhattan CC)

     Digital Design

Digital Design

Principles of K-Map Minimization

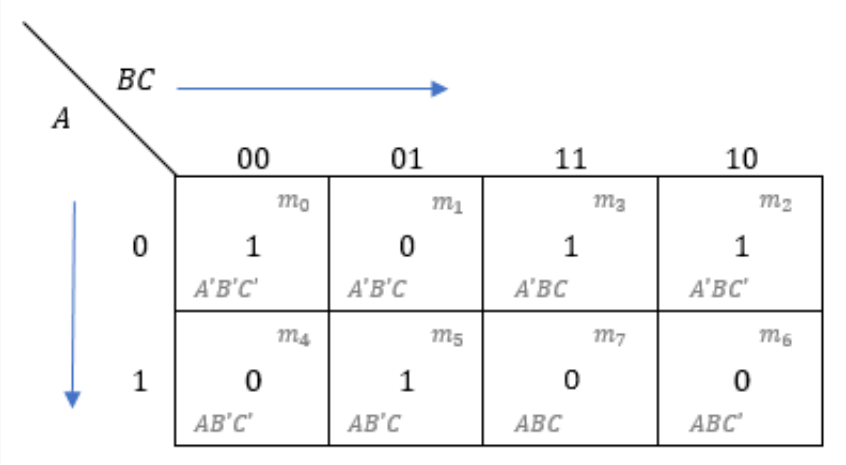
K-Map minimization is possible because terms in each adjacent cell in the map differ by only one literal. This is because the Gray code differs in only one bit position. Completing a K-Map is determined by the output of the Boolean function it represents. For example, consider the Boolean function

$$F(A, B, C) = \sum (0, 2, 3, 5)$$

The truth table for the function is:

				Sum of Products	
A	B	C	Output	Term	Designation
0	0	0	1	$A'B'C'D'$	m_0
0	0	1	0	$A'B'C'D$	m_1
0	1	0	1	$A'B'CD'$	m_2
0	1	1	1	$A'B'CD$	m_3
1	0	0	0	$A'BC'D'$	m_4
1	0	1	1	$A'BC'D$	m_5
1	1	0	0	$A'BCD'$	m_6
1	1	1	0	$A'BCD$	m_7

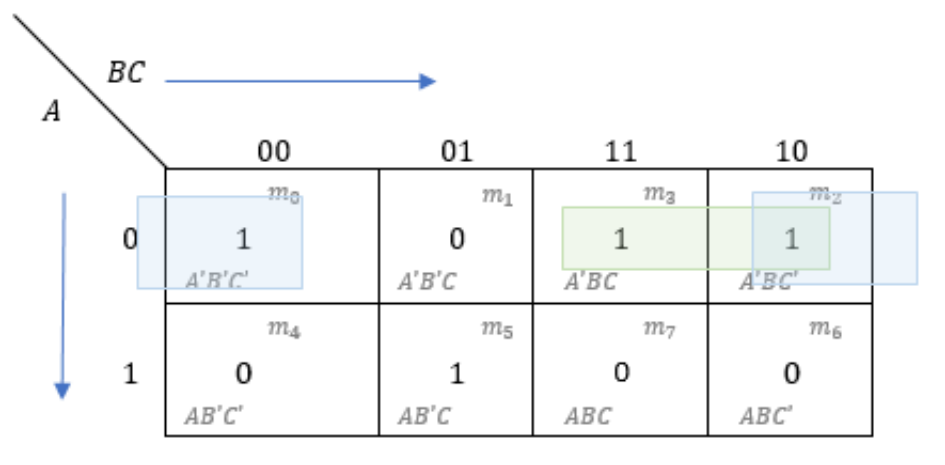
The completed associated K-Map is:



Principles of simplifying Boolean expressions using a K-Map

1. Cells containing the number 1 are grouped into groups of 2**n cells of (1, 2, 4, 8, ..). Create groups as large as possible.
2. Groups may not contain a 0
3. In each group, eliminate the variable(s) that vary
4. Groups may wrap around the K-Map

Grouping the K-Map



Minimizing

Blue Group

Term	Notation	
A'B'C'	m_o	A' B' C'
AB'C'	m_2	A' B C'

The dislike terms cancel out. The resulting term is

A'C'

Green Group

Term	Notation	
A'BC	m_3	A' B C
A'BC'	m_2	A' B C'

The dislike terms cancel out. The resulting term is

A'B

The last 1, that is not part of a group is considered by itself. The final, minimizaed expression is:

$$A'B + A'C' + AB'C$$

Grouping Hints

In a 4 × 4 K-Map:

- One square represents one minterm, yielding a term of four literals
- Two adjacent squares represent two minterms, yielding a term of threeliterals
- Four adjacent squares represent four minterms, yielding a term of twoliterals
- Eight adjacent squares represent eight minterms, yielding a term of oneliteral
- Sixteen adjacent squares represent sixteen minterms, always equal to 1

In a 3 × 3 K-Map:

- One square represents one minterm, yielding a term of three literals
- Two adjacent squares represent two minterms, yielding a term of twoliterals
- Four adjacent squares represent four minterms, yielding a term of oneliteral
- Eight adjacent squares represent eight minterms, always equal to 1

Don't Care Conditions

We have considered that the logical sum of the minterms associated with a Boolean function specifies the conditions when the function equals or returns a 1and that the

function is equal to 0 for the remaining minterms. But in application,some functions, referred to as incompletely specified functions, are functions that have unspecified outputs for some input combinations. It doesn't matter what the input to these minterms are and these unspecified minterms of a function are referred to as don't care conditions. A don't care minterm is a combination of variables whose logical value is not specified and is represented in K-Maps with an X.

When minimizing a Boolean function using a K-Map, cells marked with an X can be treated as a 1 or 0 and should be considered such that the simplest expression is achieved.

Implicants

In a Boolean function, an implicant is a minterm or maxterm used in the function.For example, int the function $F = AB + ABC + BC$, AB, ABC, and BC are implicants. Implicant types that might be handy to know are:

- Prime Implicant: Confusing definitions exist for this term; one source defines any grouping found in a K-Map as prime implicant while the text defines prime implicant as the largest grouping in a K-Map.
- Essential Prime Implicant: Source 1: Those groups that cover at least one minterm that can't be covered by any other prime implicant. The text: A prime implicant is essential if it is the only prime implicant that covers theminterm.
- Redundant Prime Implicants: The prime implicants for which each of its minterm is covered by some essential prime implicant.
- Selective Prime Implicants: The prime implicants for which are neither essential nor redundant aka non-essential prime implicants.

Karnaugh Maps and the Gray Code

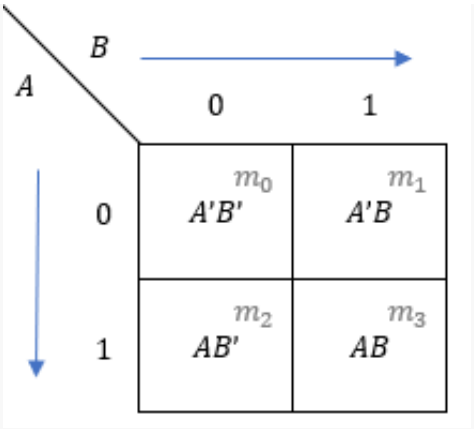
Attached Files: Generating Gray code.docx (26.149 KB)

Using Boolean algebra to simplify Boolean expressions can be difficult and may lead to solutions which, though they appear minimal, are not. The Karnaugh Map or K-Map provides a simple and straight-forward method of minimizing Boolean expressions that represent combinational logic circuits. A Karnaugh Map is a pictorial method of grouping together expressions with common factors and then eliminating unwanted variables.

A K-Map presents the same information found in a truth table but reformats it.

Consider the following two variable truth table and its associated K-Map in SOP form.

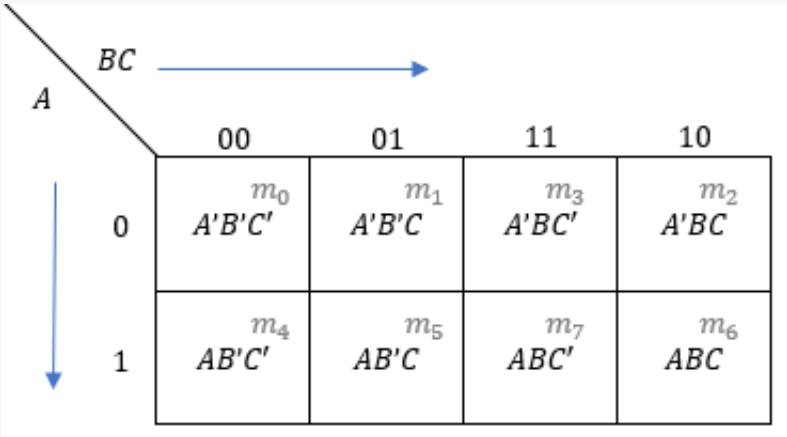
		Sum of Products		Product of Sums	
A	B	Term	Designation	Term	Designation
0	0	$A'B'$	m_0	$A + B$	M_0
0	1	$A'B$	m_1	$A + B'$	M_1
1	0	AB'	m_2	$A' + B$	M_2
1	1	AB	m_3	$A' + B'$	M_3



Notice the information contained in the truth table is contained in the K-Map

Consider the following three variable truth table and its associated K-Map in SOP form.

			Sum of Products		Product of Sums	
A	B	C	Term	Designation	Term	Designation
0	0	0	$A'B'C'$	m_0	$A+B+C$	M_0
0	0	1	$A'B'C$	m_1	$A+B+C'$	M_1
0	1	0	$AB'C$	m_2	$A+B'+C$	M_2
0	1	1	$A'BC$	m_3	$A'+B'+C'$	M_3
1	0	0	$AB'C'$	m_4	$A'+B+C$	M_4
1	0	1	$A'BC'$	m_5	$A'+B+C'$	M_5
1	1	0	ABC'	m_6	$A'+B'+C$	M_6
1	1	1	ABC	m_7	$A'+B'+C'$	M_7



The Gray Code (Reflection)

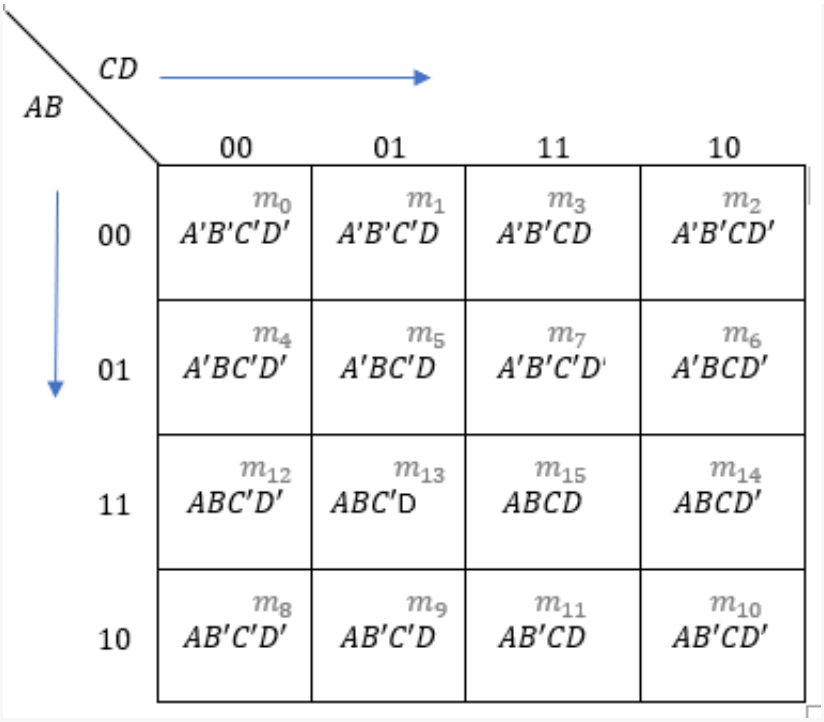
Notice the binary encoding along the top and side of the K-Map matrices. This encoding scheme is known as the Gray code after Frank Gray a noted physicist and researcher at Bell Labs. The Gray code is also known as reflected binary code and is an ordering system in which every successive value differs only by one bit. It should be noted that the binary grouping does not represent a binary, base-2 number; the Gray code is baseless.

The following is a comparison between decimal (base-10), binary (base-2), and binary reflected (Gray code) encoding schemes.

Decimal, Binary, Binary-Reflected Comparison		
Decimal	Binary	Binary-Reflected
Base 10	Base 2	No base
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111

In the next example of a four variable truth table and its associated K-Map notice the Gray code or reflective encoding along the top and side of the K-Map. Notice, also, the location of the min-terms within the K-Map.

				Sum of Products		Product of Sums	
A	B	C	D	Term	Designation	Term	Designation
0	0	0	0	$A'B'C'D'$	m_0	$A+B+C+D$	M_0
0	0	0	1	$A'B'C'D$	m_1	$A+B+C+D'$	M_1
0	0	1	0	$A'B'CD'$	m_2	$A+B+C'+D$	M_2
0	0	1	1	$A'B'CD$	m_3	$A+B+C'+D'$	M_3
0	1	0	0	$A'BC'D'$	m_4	$A+B'+C+D$	M_4
0	1	0	1	$A'BC'D$	m_5	$A+B'+C+D'$	M_5
0	1	1	0	$A'BCD'$	m_6	$A+B'+C'+D$	M_6
0	1	1	1	$A'BCD$	m_7	$A'+B+C+D$	M_7
1	0	0	0	$AB'C'D'$	m_8	$A'+B+C+D$	M_8
1	0	0	1	$AB'C'D$	m_9	$A'+B+C+D'$	M_9
1	0	1	0	$AB'CD'$	m_{10}	$A'+B+C'+D$	M_{10}
1	0	1	1	$AB'CD$	m_{11}	$A'+B+C'+D'$	M_{11}
1	1	0	0	$ABC'D'$	m_{12}	$A'+B'+C+D$	M_{12}
1	1	0	1	$ABC'D$	m_{13}	$A'+B'+C+D'$	M_{13}
1	1	1	0	$ABCD'$	m_{14}	$A'+B'+C'+D$	M_{14}
1	1	1	1	$ABCD$	m_{15}	$A'+B'+C'+D'$	M_{15}



Verilog Hardware Definition Language

Attached Files: The Complete Verilog Book (Vivek Sagdeo).pdf (6.015 MB)

Section 3.9 of the Digital Design Text

Manual methods for designing logic circuits are feasible only when the circuit is small. Practical circuits require the use of computer based design tools that reduce the cost of design and minimize the risk of creating a flawed design. A *hardware description language* (HDL) is a computer based language that describes the logic circuit in textual form. An HDL is a *modeling* language rather than a *computational* language such as C++. An HDL resembles an ordinary computer language but is specifically oriented to describing circuit structures and their behavior. HDLs can be used to represent logic diagrams, truth tables, Boolean expressions, and complex abstractions of the behavior of a digital system.

HDLs can be seen as describing a relationship between signals that are the inputs to a circuit and the signals that are the outputs of the circuit. For example, an HDL description of an AND gate describes how the logic value of the gate's output is determined by the logic values of its inputs.

Characteristics of an HDL:

- As a *documentation language*, an HDL is used to represent and document digital systems in a form that can be read by both humans and computers and is suitable as an exchange language between designers.
- *Design entry* creates an HDL based description of the functionality that is to be implemented in hardware. Depending on the HDL, the description can be in a variety of forms: Boolean logic equations, truth tables, a net list of interconnected gates, or an abstract behavioral model.
- *Logic simulation* displays the behavior of a digital system through the use of a

computer. A simulator interprets the HDL description and either produces readable output, such as a time ordered sequence of input and output signal values, or displays waveforms of signals. The simulation of a circuit shows how the hardware will behave before it is actually fabricated. Simulation detects functional errors in a design without having to physically create and operate the circuit. Errors that are detected during a simulation can be corrected by modifying the appropriate HDL statements. (Similar to using errors generated by a compiler to correct a computer program.) The stimulus that tests the functionality of th design is called a *test bench*. To simulate a digital system, the design is first described in an HD and then verified by simulating the design and checking it with a *test bench*, which is also written in the HDL. A test best simulator may be found by following this link Edit code - EDA Playground

- *Logic synthesis* derives an optimized list of physical components and their interconnections (called a *netlist*) from the model of a digital system described in an HDl. The netlist can be used to fabricate an integrated circuit or to lay out a printed circuit board with the hardware counterparts of the gates in the list. Logic synthesis produces a database describing the elements and structure of a circuit.
- *Timing verification* confirms that a synthesized and fabricated integrated circuit will operate at a specified speed. Because each logic gate in a circuit has a propagation delay, a signal transition at the input of a circuit cannot immediately cause a change in the logic value of the output of a circuit. Timing verification checks each signal path to verify that it is not compromised by propagation delay.
- In very large scale integrated (VLSI) circuit design, *fault simulation* compares the behavior of an ideal circuit with the behavior of a circuit that contains a process induced flaw such a circuit fabricated with a flaw.

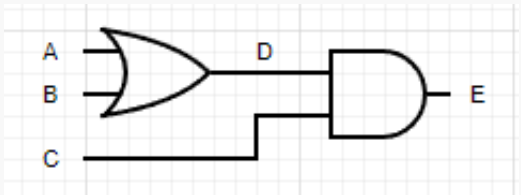
Verilog HDL

A Verilog model is composed of text using keywords which are predefined lowercase identifiers that define the language constructs. A full description of Verilog is attached to this item. In text, keywords are presented in lowercase bold face type. Keywords include, but not limited to: **module**, **endmodule**, **input**, **output**, **wire**, **and**, **or**, and **not**. Lines of text terminate with semicolons (**;**); text entered after to slashes (*//*) are comments. Verilog is case sensitive.

The term module refers to the text enclosed by the keyword pair module ... endmodule. A *module* is a fundamental descriptive (design) unit in Verilog that is declared with the keyword **module** that must always be terminated by the keyword **endmodule**.

Example:

Description: Description of a circuit where the output of an OR gate is one of two inputs to and AND gate.
Boolean Expression: E = (A + B)C
Logic Diagram:



Verilog Description:

```
module or_and (
  output    E,
  input     A, B, C
) ;

 wire D ;
```

```
  assign D = A || B ;  // '||' is the logical or operator
  assign E = C && D ;  // '&&' is the logical and operator

endmodule
```

Components:

- The keyword **module** starts the declaration of the description that is ended by the keyword **endmodule**.
- Identifiers: Identifiers (identified by green above) are meaningful names assigned by designers to modules, variables or signals, and other elements of the language. Identifiers are case sensitive and are composed of alphanumeric characters and the (_) underscore character.
- Boolean expressions describe the input-output logic of a digital circuit. In Verilog, they are composed as *continuous assignment* statements and placed with the code space defined by the **module** ... **endmodule** keywords. Continuous assignment statements have the appearance of an equation but it is essential to understand that ***a continuous assignment does not prescribe a computation***; it defines a relationship between signals in a circuit. A continuous assignment statement is continuous in the sense that it always governs the relationship between inputs and outputs.
- Operator symbols: **&&** (logical and), **||** (logical or), and **!** (not) represent logical operations.
- The port list of a module is the interface between the module and its environment. In the module or_and, the ports are the inputs **A**, **B**, and **C** and the output **E** of the circuit. The mode, or direction, of a port distinguished between **input**, **output**, and **inout** (bidirectional) ports. The input values to a circuit are determined by the environment; the output values are determined within the circuit and result from the action of the inputs on the circuit. The port list is enclosed within parenthesis; commas separate elements of the list; the statement is terminated by a (**;**) semicolon.
- Internal connections such a **D** are declared as wires.

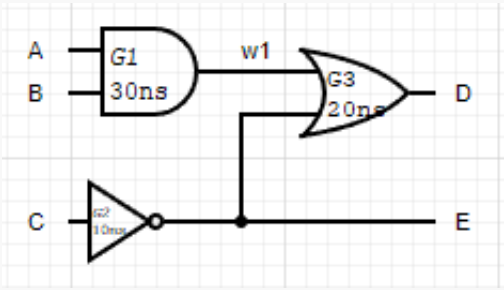
Additional Keywords are **and**, **nand**, **or**, **nor**, **xor**, **xnor**, **buf**, **not**, **bufif0**, **bufif1**, **notif0** and **notif1**.

Example:

Description: A structural model of a circuit featuring and, or, and propagation delay using descriptive keywords.
Boolean Expressions:

```
D = AB + C'
E = C'
```

Logic Diagram:



Verilog Description:

```
module and_or_prop_delay (E, D, C, B, A) ;
 output    D, E ;
 input     A, B, C ;
```

```
 wire w1;

 and G1 #30 (w1, A, B) ;     // output is listed first in port
list
 not G2 #10 (E, C) ;         // output is listed first in port
list
 or G3 #20 (D, w1, E) ;      // output is listed first in port
list

endmodule
```

Notes:

- Outputs appear in port lists first
- The gates may be listed in any order. They operate concurrently in a simulation. A signal can affect simultaneously all of the gates to which it is onnected.
- Declaration vs Instantiation. The *declaration* of a Verilog module specifies the input-output behavior of the hardware it represents. Predefined primitives such as **and**, and **or** are not declared because their definition is defined by the language. After a module has been declared it can be used or *instantiated* within another module in the design. This is similar to the definition of a C++ function and its subsequent use else where in a program.

Boolean Algebra

Attached Files: CSC215 Circuit Simplification 1.docx (21.615 KB)
CSC215 Table of Postulates and Theorems.docx (14.884 KB)

There are videos presented at the end of this item.

Circuit Design Websites - Use these websites to draw circuits then cut and paste them to Word documents (videos below).

- CircuitVerse -
- Circuit Diagram -
- CircuitLab -

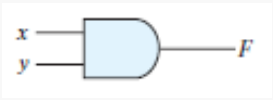
Topics Discussed:

- Definitions
  Read sections 2.2, 2.3 of the Digital Design text
  - Boolean algebra is a closed with respect to a binary operator (*, +). The result of every operation is a 0 or a 1
  - Truth Tables
    - A truth table is a table displaying all possible truth values of a Boolean expression via its individual terms. Truth tables are used in connection with Boolean algebra, boolean functions, and propositional calculus.
  - Three basic Boolean operations
    - AND Operation
      - A binary operation having two operands
      - Represented as multiplication using symbology similar to decimal Algebra
      - Will evaluate to true if all operands are true
      - Truth Table - A truth table is a table displaying all possible

truth values of a Boolean expression. Truth tables are used in connection with Boolean algebra, boolean functions, and propositional calculus.

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

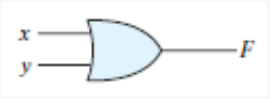
- Logic Gate - The AND operation is represented by the AND logic gate.



- OR Operation
  - A binary operation having two operands
  - Represented as addition using symbology similar to decimal Algebra
  - Will evaluate to true if any operand is true
  - Truth Table - A truth table is a table displaying all possible truth values of a Boolean expression. Truth tables are used in connection with Boolean algebra, boolean functions, and propositional calculus.

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

  - Logic Gate - The OR operation is represented by the OR logic gate.

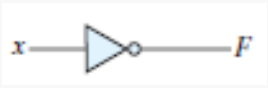


- NOT Operation
  - A unary operation having only one operand
  - Represented as an overbar, prime symbol, or logical not operator
  - Will return the complement of the input
  - Truth Table - A truth table is a table displaying all possible truth values of a Boolean expression. Truth tables are used in connection with Boolean algebra, boolean functions, and propositional calculus.

x	F
0	1
1	0

  - Logic Gate - The NOT operation is represented by the NOT

logic gate.



- Truth Tables
  - A truth table is a table displaying all possible truth values of a Boolean expression via its individual terms. Truth tables are used in connection with Boolean algebra, boolean functions, and propositional calculus.
  - The number of rows in a truth table is determined by the number of unique variables in the Boolean expression. There are 2 raised to the number of variables rows. A two variable expression would yield a table of four rows; a three variable expression would yield a table of eight rows; and, a four variable expression would yield a table of 16 rows.
  - To ensure that a truth table represents every possible truth value, it is organized according to a pattern. Consider the following Boolean expression and its truth table:

$$F = x + y'z$$

x	y	z	y'	$y'z$	$x + y'z$
0	0	0	1	0	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	1	0	1
1	0	1	1	1	1
1	1	0	0	0	1
1	1	1	0	0	1

Examining the Boolean expression, three variables are identified x, y, and z. Because there are three variables, the number of rows in the truth table is 2 ^ 3 = 8 rows. The first three columns are labeled with the variable names in ascending order. Notice the pattern of the true/false (1/0) values.

- the first column is populated first with one half the rows having a false value and the remaining with true values;
- the second column is populated with one quarter of the rows having false values followed by one quarter with true values - a pattern that is followed for the remaining rows;
- the third column is populated with one eighth of the rows have a false value followed by one eighth with true values - a pattern that is followed for the remaining rows

Subsequent columns contain the result of individual Boolean operations in an order starting from the inner most operation to the final operation representing the value of the expression.

- Postulates and Theorems
  Read sections 2.4 of the Digital Design text

  Boolean expressions may be represented as logic diagrams using the logic gates

described above. In fact, Boolean expressions represent the circuits of computer chips algebraically while logic gates represent the circuits as logic diagrams - two different representations of the same logic circuit. In computer systems, in the actual underlying hardware (the chips), it is important that logic circuits be as efficient as possible because efficient circuits generate less heat and perform more optimally than more complex circuits. Efficient circuits are comprised of fewer logic gates than more complex circuits. A complex circuit can be minimized using logical equivalences presented in the text as postulates and theorems (see attached handout). Postulates, or axioms, are statements assumed to be true while theorems are statements that can be shown to be true. Postulates can be used to demonstrate the truth of theorems (See section 2.4 of Digital Design text).

- Duality
  The Table of Postulates and Theorems are presented in the form of two columns a and b where the a column represents 'or' operations and the b column represents 'and' operations. For all but one postulate and theorem (involution or double negation), there is an 'or' and 'and' representation. Values in the a column may be derived from the values in the b column, and vice versa, by interchanging the binary operators and the identity elements. "This important property of Boolean algebra is call the *duality principle* and states that every algebraic expression deducible from the postulates of Boolean algebra remains valid if the operators and identity elements are interchanged. Applying the duality principle is as easy as interchanging 'and' and 'or' and changing the identity elements (change 1s to 0s and 0s to 1s).

  An identity element is a value that when paired with any variable in a Boolean operation, the value of the variable will be returned. In an 'or' operation, the identity element is 0 and in the 'and' operation, the identity element is 1.

- Operator Precedence
  Similar to any system have operators such as computer languages and decimal math and algebra, the operators of Boolen algebra respect and order of operations that ensures every expression will be evaluated properly at all times. The operator precedence, or order of operations, for Boolean Algebra is:

  1. Parentheses [( )]
  2. NOT [']
  3. AND [*]
  4. OR [+]

- Circuit Minimization using Boolean Algebra
  Circuit minimization is a form of logic optimization used to reduce complex logic in integrated circuits. Using postulates and theorems, it is possible to change an algebraic expression while retaining its truth value as demonstrated using a truth table. The goal of circuit minimization is to obtain the most efficient logic circuit with the fewest number of Boolean operators.

  Consider the Boolean expression: $(x' + z')(x + y' + z')$ [question 2.3{f) at the end of chapter 2] that can be minimized using the postulates and theorems as follows:

Step	Change in Expression	Reason

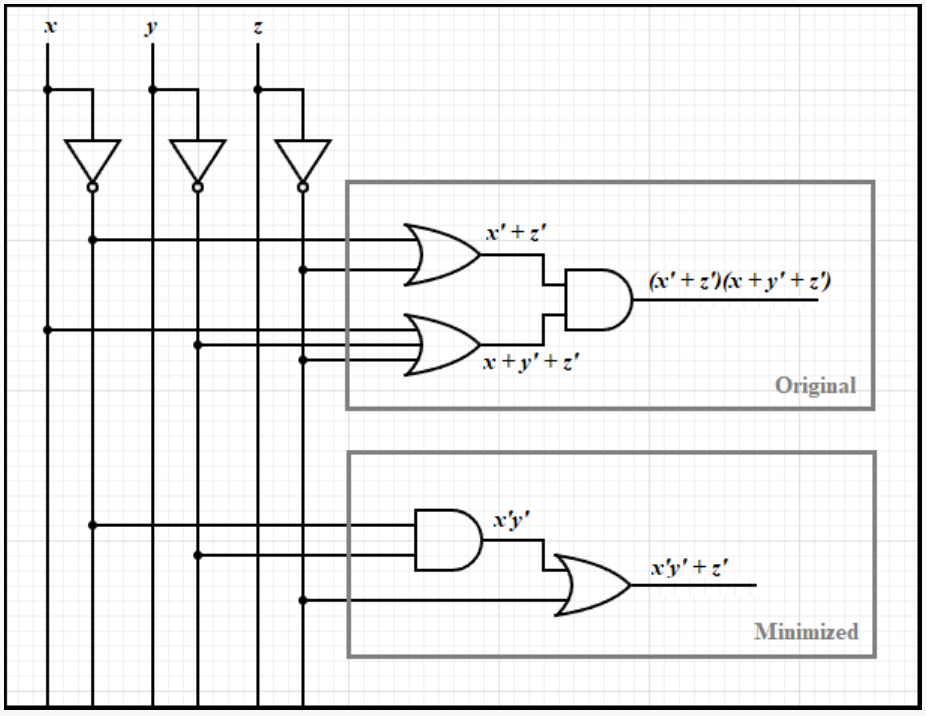
1	$(x' + z')(x + y' + z')$	Given	
2	$x'x + x'y' + x'z' + xz' + y'z' + z'z'$	Postulate 4(a) [Distributive Property]	
3	$0 + x'y' + x'z' + xz' + y'z' + z'z'$	Postulate 5(b) [$x'x$ is equivalent to 0]	
4	$x'y' + x'z' + xz' + y'z' + z'z'$	Postulate 2(a) [$0 + x'y$ is equivalent to $x'y$]	The first term has been eliminated. The expression is becoming smaller.
5	$x'y' + x'z' + xz' + y'z' + z'$	Theorem 1(b) [$z'z'$ is equivalent to z']	A term has been reduced.
6	$x'y' + z'(x' + x + y' + 1)$	Postulate 4(a) [Distributive Property]	
7	$x'y' + z'(x' + x + (y' + 1))$	Theorem 4(a) [Associative Property]	
8	$x'y' + z'(x' + x + 1)$	Theorem 2(a) [$y' + 1$ is equivalent to 1]	Another term has been elminated.
9	$x'y' + z'(x' + (x + 1))$	Theorem 4(a) [Associative Property]	
10	$x'y' + z'(x' + 1)$	Theorem 2(a) [$x+1$ is equivalent to 1]	Another term has been eliminated
11	$x'y' + z'(1)$	Theorem 2(a) [$x+1$ is equivalent to 1]	Another term has been eliminated
12	$x'y' + z'$	Postulate 2(b) [$z(1)$ is equivalent to z]	
13	$x'y' + z'$	Result	

It is important after an expression has been minimized, that the result be checked to ensure it is logically equivalent to the original expression. A truth table can be used as follows:

x	y	z	x'	y'	z'	$x' + y'$	$x + y'$	$(x' + z')(x +$	x'	$x'y'$

			$'$	$'$	$'$	z'	$+ z'$	$y' + z')$	y'	$+ z'$
0	0	0	1	1	1	1	1	1	1	1
0	0	1	1	1	0	1	1	1	1	1
0	1	0	1	0	1	1	1	1	0	1
0	1	1	1	0	0	1	0	0	0	0
1	0	0	0	1	1	1	1	1	0	1
1	0	1	0	1	0	0	1	0	0	0
1	1	0	0	0	1	1	1	1	0	1
1	1	1	0	0	0	0	1	0	0	0
								Truth value of original expression		Truth value of resultant expression

The truth table demonstrates the original and resultant expression are logically equivalent. The resultant expression has been minimized and is comprised of fewer logica gates as demonstrated below



- Boolean Functions
  Read section 2.5 of the Digital Design text

  A Boolean function is described by an algebraic expression consisting of binary variables, the constants 0 and 1 and the logic operation symbols AND, OR, and NOT. For a given set of binary variables, the function will

evaluate to either a 0 or 1. Boolean functions are a more formal representation of Boolean expressions as presented above. An example of a Boolean function is:

$$F_1 = x + y'z$$

The function F_1 will evaluate to 1 if x is equal to 1 or both y' and **z** are equal to 1.

A Boolean function expresses the logical relationship between binary variables and is evaluated by determining the binary value of the expression of all possible values of the variables.

As with the Boolean expressions presented above, Boolean functions can be represented in a truth table, can be algebraically manipulated using the postulates and theorems of Boolean Algebra, and can be represented in a digial logic diagram using logic gates. When drawing a logic diagram of a Boolean functions, each term requires a gate and variable within a term is referred to as a literal.

- Complement of a Function
  Read section 2.5 of the Digital Design text

  The complement of a function is F is F' and is obtained by interchanging 0s for 1s and 1s for 0s and may be derived using DeMorgan's theorems. Consider the following Boolean Function:

  $$F = (A + B + C)'$$

  The postulates and theorems are presented in terms of two variables but they can be extended to three variables. The following example demonstrates how DeMorgan's Theorem can be used to find the Boolean Function's complement.

	Change	Reason
1	$(A + B + C)' = (A + x)'$	let $B + C = x$
2	$= A'x'$	Theorem 5(a) [DeMorgan's Theorem]
3	$= A'(B+C)'$	expand x
4	$= A'(B'C')$	Theorem 5(a) [DeMorgan's Theorem]
		Theorem 4(b)

5	$= A'B'C'$	[Associative Property]

The generalized form of DeMorgan's theorems states that the complement of a function is obtained by interchanging AND and OR operators and complementing each literal.

- Minterms and Maxterms
  Read section 2.6 of the Digital Design text

  Canonical Form
  A binary variable may appear in either its normal form or complement form where the normal form has a value of 1 and the complement form a value of 0. This is demonstrated in truth tables where a variable x is represented in both forms.

  In Boolean algebra, a product term (AND operation) in which each variable appears once in either its normal or complemented form is called a minterm. A Boolean function can be expressed, canonically, as a sum (OR operation) of minterms where each minterm corresponds to a column of a function's truth table where the value is 1. A Boolean function in this form is in Sum of Product or SOP form.

			Minterm	
x	y	z	Term	Destination
0	0	0	$x'y'z'$	m_0
0	0	1	$x'y'z$	m_1
0	1	0	$x'yz'$	m_2
0	1	1	$x'yz$	m_3
1	0	0	$xy'z'$	m_4
1	0	1	$xy'z$	m_5
1	1	0	xy	m_6

			z'	
1	1	1	$x\ y\ z$	m_7

In Boolean algebra, a sum term (OR operation) in which each variable appears once in either its normal or complemented form is called a maxterm. A Boolean function can be expressed, canonically, as a product (AND operation) of maxterms where each maxterm corresponds to a column of a function's truth table where the value is 0. A Boolean function in this form is in Product of Sum or POS form.

			Maxterm	
x	y	z	Term	Designation
0	0	0	$x+y+z$	M_0
0	0	1	$x+y+z'$	M_1
0	1	0	$x+y'+z$	M_2
0	1	1	$x+y'+z'$	M_3
1	0	0	$x'+y+z$	M_4
1	0	1	$x'+y+z'$	M_5

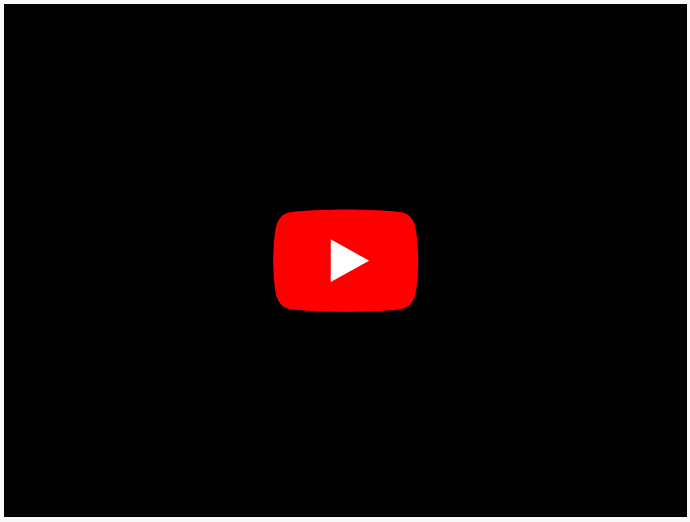
1	1	0	$x' + y' + z$	M_6
1	1	1	$x' + y' + z'$	M_7

A Boolean function that is expressed as a sum of minterms or as a product of maxterms is said to be in its canonical form.

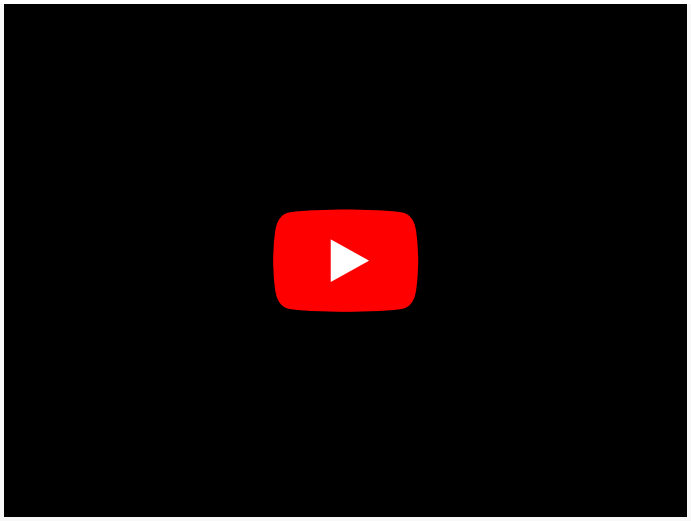
- Algebraic Manipulation

Videos:

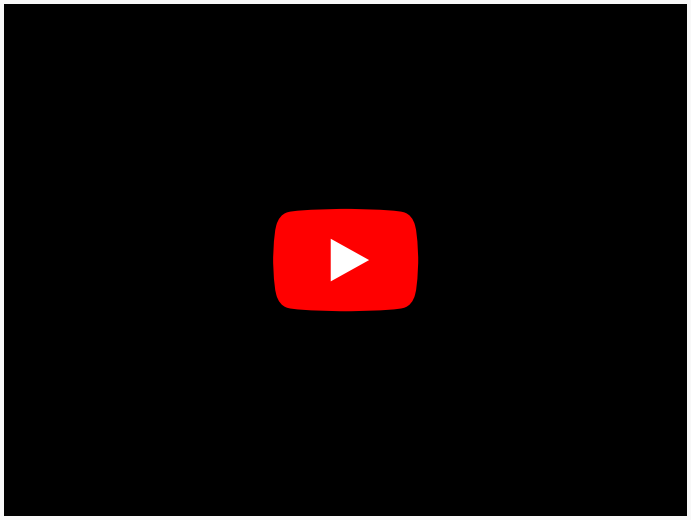
- Snipping Tool Short Cut (Windows)



- Windows Snipping Tool



- Cut and Paste (Mac)



Binary Logic

Attached Files: CSC215 Table of Basic Binary Logic.docx (90.986 KB)

Read sections 1.9 of the Digital Design text

- Operators and Operands
  - Operands may have a value of 0 (false, low-voltage) or 1 (true, higher-voltage)
  - Unary operators
    An operator with one operands
  - Binary operators
    An operator with two operands
  - Ternary operators
    An operator with three operands
- Basic Boolean Operations and Operators
  - AND Operation [Multiplication Operator (*)]
    - A * B = C or AB=C
    - The *rule*: All operands must be true for the result to be true
  - OR Operation [Addition Operator (+)]
    - A + B = C

        - The *rule*: At least one operand must be true for the result to be true
    - NOT Operation [Prime Operator (')]
        - A' = B
        - The *rule*: The result is the inverse or negation of the operand
- Truth Table for one operand
    - NOT

Input	Output
A	B
0	1
1	0

- Truth Tables for two operands
    - AND

Input		Output
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

    - OR

Input		Output
A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

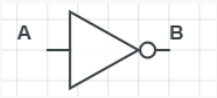
- Truth Table for three operands
    - AND

Input			Output
A	B	C	D
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

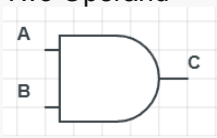
    - OR

Input			Output
A	B	C	D
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

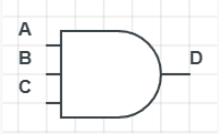
- Logic Gates
  - NOT



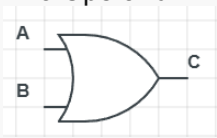
  - AND
    - Two Operand



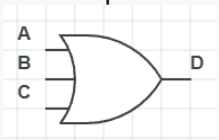
    - Three Operand



  - OR
    - Two Operand



    - Three Operand



Arithmetic Operations

Attached Files: CSC215 Twos Complement.pdf (2.24 MB)

A video is presented at the end of this item.

Arithmetic operations such as addition and subtraction are possible because of the way data can be represented within a computer.

- Signed Binary Integers
  In ordinary arithmetic, negative numbers are indicated by a minus sign and positive numbers are indicated by a plus sign. In a binary number, it is customary to represent the sign with a bit placed in the left most, or most significant, position of the number where a 0-bit value represents a positive number, and a 1-bit represents a negative number.

  The user must know when a binary number is signed or unsigned. The binary number 01001 will be understood as a 9 (unsigned binary) or +9 (signed binary). The number 11001 will be understood as 25 (unsigned binary) or as -9 (signed binary) because the most significant digit represents a minus sign.

- Representing Negative Numbers

Consider the binary representation of the number 9 represented in an 8-bit byte: 0000 1001. Note that the most significant or left most bit is 0 indicating that the number is either unsigned or possibly a positive number. The number -9 can be represented in different ways:

        - 1000 1001 : signed magnitude format, obtained by changing the most significant bit to 1
        - 1111 0110 : signed 1's complement format, obtained by complementing the bits of +9 including the sign bit
        - 1111 0111: signed 2's complement format, obtained by 2's complementing the bits of +9 including the sign bit

    - Binary Addition

Binary addition follows the rule of ordinary, decimal addition. When the signs are the same, the numbers are added, and the sum inherits the common sign. If two n-bit numbers are added and the sum is n + 1 bits and condition called an overflow occurs.

Unlike decimal addition where an overflow is not a concern, in a computer, an overflow is a significant concern because memory units are of finite size. Consider the following:

```
int a = 255 ;
int b = 255 ;
long c = 0 ;

   c = a + b ;  // the programmer must ensure the recipient
variable is large
                // enough to accept the sum
```

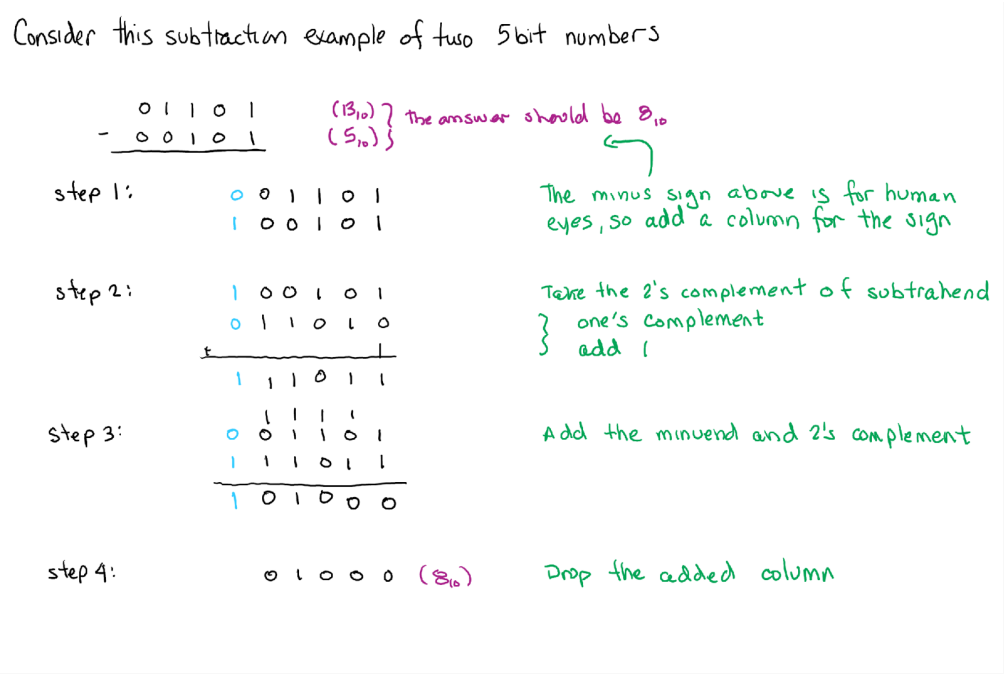
If the variable `c` was of the integer data type, there would not be enough memory to store the sum of a + b .

The rules of binary addition are the same as decimal addition but only two symbols, 0 and 1, are used. Keep the following in mind:

    - 0 + 0 = 0
    - 1 + 0 = 1
    - 0 + 1 = 1
    - 1 + 1 = 10
    - 1 + 1 + 1 = 11

- Binary Subtraction
  Binary subtraction is the result of the binary addition of the two's complement of an equation's subtrahend. To remind you, consider the following:

```
   5      (this is called the minuend)
 - 3      (this is called the subtrahend)
 ---
   2      (this is called the difference)
```



Videos:

- Binary Addition and 2's Complement

Data Representation

Attached Files: ASCII EBCDIC Chart.pdf (359.297 KB)

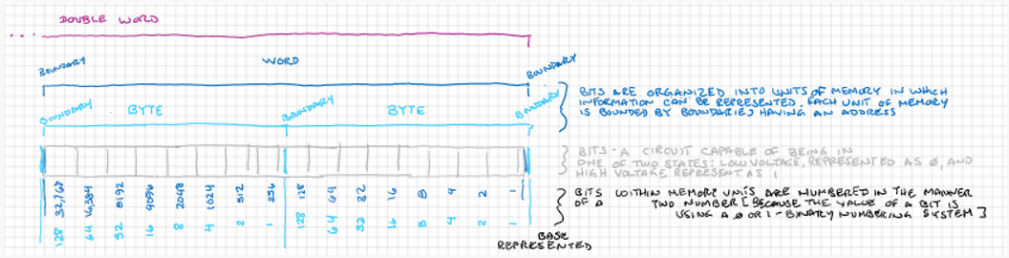
Topics in the topic Data Representation
Read sections 1.1, 1.2, 1.3, 1.4, 1.5, of the Digital Design text

Data Organization

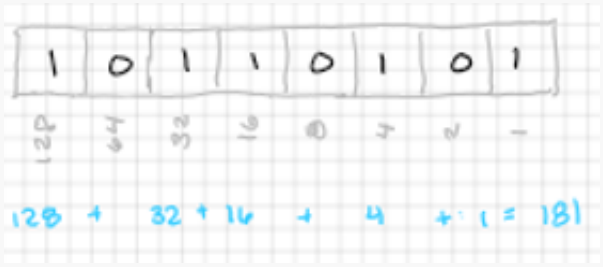
- Bit - A circuit capable holding a high or low voltage.
- Memory Units - Organizations of bits bounded by a specific boundary
    - BYTE - An organization of 8 bits bounded by a byte boundary
    - WORD - An organization of 16 bits bound by a word boundary
    - DOUBLE WORD (DWORD) - An organization of 32 bits bounded by a double word boundary
    - QUAD WORD (QWORD) - Any organization of 64 bits bounded by a quad word boundary

Data Representation

- Information is represented within memory units using combinations of high and low voltages in the bits comprising the memory units. A high voltage value is called '1' and a low voltage value is called '0'. The combinations of 1s and 0s within memory units represent information,
    - The term *binary* means "relating to, composed of, or involving two things." In the case of information representation, the two things are the high and low voltages a bit is capable of holding, or the 1 and 0. The representation of information in a binary form is called *binary notation*.
- A *binary number* is a number expressed in the base-2 numeral system or binary numeral system that uses only two symbols a 1 (one) and 0 (zero). A binary number is a very basic representation of information that can easily be stored within a memory unit where each bit in a memory unit corresponds to a column in a base-2 number. The diagram belows demonstrates the binary numbering system and shows how memory units are 'overlapping' in memory.



- Binary numbers can be easily interpreted by converting a binary number into a decimal number. A decimal number is a base-10 numeral system that uses ten symbols 0 thru 9. The diagram below demonstrates how a binary number, for example 10110101 (base-2) can be converted to decimal or base-10.

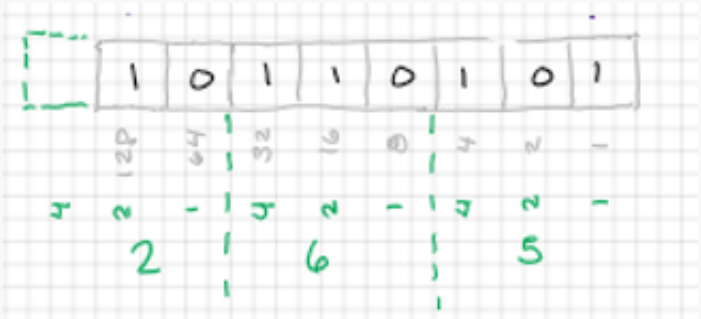


Notice, that the decimal values of each base-2 column are added together when the the binary value is 1.

The binary number 10110101 (base-2) is equal to the decimal number 181 (base-10)

- Binary numbers can be represented using different number systems to make understanding binary easier. Binary can be converted from base-2 to other base systems by seeing 'nybbles' within each memory unit that corresponds to the number of bits needed to represent one digit of the number system.
    - Octal - a base-8 numeral system using the symbols 0-7 that is historically associated with a 7 bit byte and the original ASCII character set. Because the number of bits needed to

represent 8 symbols is 3 (2^3), the nybble size is three. The diagram below demonstrates how a binary number, for example 10110101 (base-2) can be converted to octal or base-8.



Notice, the decimal values of each base-2 column within the nybble are added together when the binary value is 1.

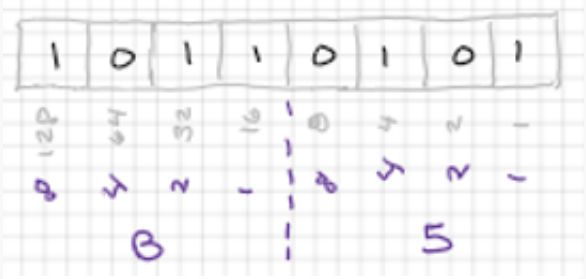
The binary number 10110101 (base-2) is equal to the octal number 265 (base-8).

- Binary Coded Decimal (BCD) - a base-2 representation of the individual digits of a decimal number. Because the number of bit needed to represent 10 symbols is 4 (2^3 is too little), the nybble size is 4. The diagram below demonstrates how a decimal (base-10) number, for example 35, is represented in binary coded decimal.

Notice, the decimal values of each base-2 column within the nybble are added together when the binary value is 1.

The decimal number 35 (base-10) is equal to the BCD number 00110101 (base-2).

- Hexadecimal - a base-16 numeral system using the symbols 0-F (0-9, A-F) that is historically associated with an 8 bit byte and the EBCDIC character set. Because the number of bits needed to represent 16 symbols is 4 (2^4), the nybble size is four. The diagram below demonstrates how a binary number, for example 10110101 (base-2) can be converted to hexadecimal or base-16.



Notice, the decimal values of each base-2 column within the nybble are added together when the binary value is 1

The binary number 10110101 (base-2) is equal to the hexadecimal number B5 (base-16)

- Character Representation
  A 'character set' is a mapping of characters to their identifying code values. Character sets may either be Single Byte Character Sets (SBCS), where each character is identified by a value one byte wide, or Multibyte character sets such as the double-byte character set, where each character set provides a means to represent character sets with many characters. Examples of character sets are:

  - ASCII - an acronym for American Standard Code for Information Interchange, the original ASCII is a 7 bit code for representing letters, numerals, and other symbols. Because a bytes are comprised of 8 bits rather than 7 bits, the extended ASCII character set includes an additional 128 characters. See attached document.

  - EBCDIC - an acronym for Extended Binary Coded Decimal Interchange Code, EBCDIC is an 8 bit code for representing letters, numerals, and other symbols used extensively in IBM mainframe computers. See attached document. See attached document.

  - Unicode is a worldwide character-encoding standard supporting numerous scripts used by languages around the world as well as a large number of technical symbols and special characters used in publishing. UTF-16 represents characters in 16 bits rather than 8 bits. The Unicode standard may be found here.

- Complements
  An alternative representation of numbers can be found in a numbers complement which is the distance a number is from its base. For example:

  - the complement of 9 (base-10) is 10 - 9 = 1 (1 is the complement of 9 in base-10)
  - the complement of 7 (base-10) is 10 - 7 = 3 (3 is the complement of 9 in base-10)
  - the complement of 1 (base-2) is 1 - 1 = 0 (0 is the complement of 1 in base-2)
  - the complement of 0 (base-2) is 1 - 0 = 1 (1 is the complement of 0 in base-2)

This alternative form of numeric representation facilitates the arithmetic operation of subtraction.