CSE 140 Midterm Review

This serves as the review doc for CSE140's midterm 1. It includes notes summarized from lecture slides, discussion and zyBook. Created by M., feel free to collaborate.

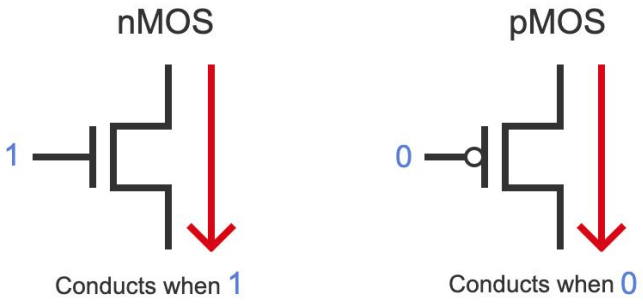
Logistics

- Topics:
  - CMOS: gate
  - Boolean Algebra: definition, duality, De Morgan's, Consensus, Shannon's Expansion
  - Circuit Specification: logic, truth table, full adder, voting machine, majority function.
  - K-map: prime, essential prime
  - Universal set: shannon's expression

Combinational Logic (ZyBook Chapter 1)

- Electrical Systems:
  - Voltage (volts), current (amps) and resistance (ohms)
  - Ohms' Law: V = IR
  - Voltage on a wire with no resistance is the same everywhere, but drops across a resistor as V = IR.
  - Switches:
    - Electronically-controlled switch has another input terminal with control input.
    - Transistor is a small switch with no mechanical parts

- **CMOS** transistors

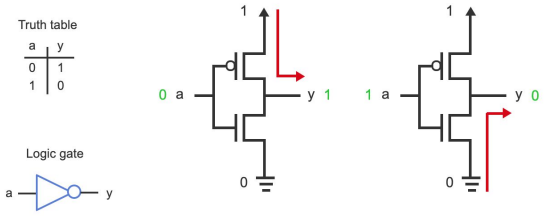


  - *nMOS:* conducts when its control input is 1.
  - *pMOS: conducts* when its control input is 0. *The graph for pMOS has a little circle.*

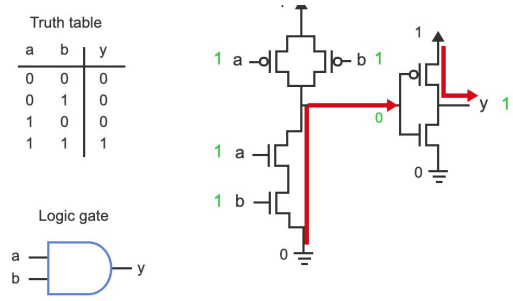
- Digital Systems

- Digital circuit: voltages that are treated as either high or low, built with connection of switches.
- Analog system: voltages that are treated as having infinite values.

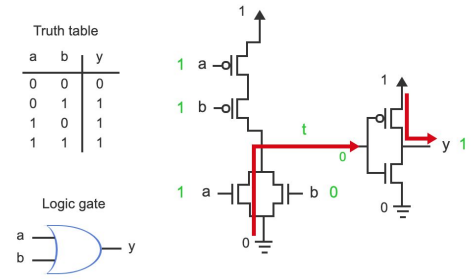
- **Logic Gates**
  - NOT gate: inverter. pMOS on the top and nMOS on the bottom.



  - AND gate



  - OR gate



- **Boolean Algebra**
  - Logical operators: AND, OR, NOT
  - Boolean equations: define the left variables (boolean expressions) based on the right variables's values.
  - Expression, Equation, Functions.

Item	Notation	Notes
Expression	ab	An expression lacks an equal sign, and involves input variables.
Equation	y = ab	An equation has an =, with expressions of input variables on the right, and an output variable on the left. (In general math, both sides of an equation can be expressions, but in this material, the left side is usually just an output variable.)
Function	Various	A relation of input values to output values. Can be represented in various ways: equation, table, circuit, etc. A function may have more than one input, but has only one output.

    - Function defines exactly one output value for unique input values, and must include all input possibilities.

- Digital Circuit Simulator
- Timing Diagrams
- Equations and Circuits
    - Design: convert behavior to a circuit → converting each operation to a gate
    - Analysis: convert a circuit to behavior (like an equation).
    - Combinational circuit: a circuit whose output value is determined solely by the **present** combination of input values.
    - Sequential circuit: a circuit whose output values may depend on the **past sequence** of input values, not just the present values.

- **Properties of Boolean Algebra**
    - Distributive: a(b + c) = ab + ac
    - Complement: a + a’ = 1
    - Identity: a * 1 = a
    - Commutative (AND): ab = ba
    - Commutative (OR): a + b = b + a
    - Null elements: a + 1 = 1
    - Idempotent: a + a = a; aa = a.
    - A complete summary of the Boolean Algebra

Property	Name	Description
a(b + c) = ab + ac a + (bc) = (a + b)(a + c)	Distributive (AND) Distributive (OR)	(AND) Same as multiplication in regular algebra (OR) Not at all like regular algebra
ab = ba a + b = b + a	Commutative	Variable order does not matter. Good practice is to sort variables alphabetically.
(ab)c = a(bc) (a + b) + c = a + (b + c)	Associative	Same as regular algebra
aa' = 0 a + a' = 1	Complement (AND) Complement (OR)	(AND) Clearly one of a, a' must be 0 1 · 0 = 0 · 1 = 0 (OR) Clearly one of a, a' must be 1 1 + 0 = 0 + 1 = 1
a · 1 = a a + 0 = a	Identity (AND) Identity (OR)	(AND) Result of a · 1 is always a's value *0* · 1 = *0* *1* · 1 (OR) Result of a + 0 is always a's value *0* + 0 = *0* *1* + (
a · 0 = 0 a + 1 = 1	Null elements	Result doesn't depend on the value of a.
a · a = a a + a = a	Idempotent	Duplicate values can be removed.
(a')' = a	Involution	(0')' = (1)' = 0 (1')' = (0)' = 1
(ab)' = a' + b' (a + b)' = a'b'	DeMorgan's Law	*Discussed in another section*

- Sum-of-products
    - The product term is ANDing variables, or a term

    - The sum of product consists solely of an ORing of product terms.
    - Interrupt: various devices surrounding a processor may request the processor to execute a sub-program on behalf of that device. Devices may be in two categories, low and high priority, and processor may disable either or both.

- Product of sums
    - The sum term is ORing variables
    - The product is ANDing all the sum terms
    -

- Binary and Counting
    - Decimal to binary, binary to decimal conversion

- Sum-of-minterms
    - Ensure same function have same equations with standard equation forms

Canonical form	Boolean equation
Sum of minterms	Canonical form of a boolean equation where right side expression is a sum-of-product with each product a **unique** min-term.
Minterm	Product term having exactly one literal for every function variable
Literal	Variable appearance, in true or complement form, in an expression

    - Transform: initially multiplying out to sum-of-products, transform each product term to a minterm, remove redundant minterms.

Given variables a, b, c. Convert y = a(b + bc') to sum-of-minterms.

y = a(b + bc')

= ab + abc'

= ab (1) + abc'

= ab (c + c') + abc'

= abc + abc' + ~~abc'~~

= abc + abc'

Duplicate terms should be removed. So y = abc + abc' + abc' becomes y = abc + abc'.
The equation is now in sum-of-minterms form.

    - Compact function notation: represents each minterm by a number. Ab'c → a/b/c (1, 0, 1) → m_5.

- Truth Table
    - For a given function, a truth table lists on the right for which rows (variable value combinations) that the output should be 1. Other rows get 0's.
        - *1 for the minterm in the functions, otherwise 0.*
    - Convert to equation:
        - Transformed to a sum of minterms equation by summing the minterms in rows **having 1.**
        - Transformed to product of sums equation by multiplying the maxterms in rows **having 0.**

- Top-down Design
    - Capture: precisely describing a circuit's desired behavior
    - Convert: the task of translating captured behavior into a circuit, possibly involving simplification.
    - E.g. Triple modular redundancy: prevent errors. → Odd number of inputs.

- Why Study Digital Design
    - Understand how the computers work under the hood
    - Embedded system: computing system embedded within another device.

- Multiple Outputs
    - Many combinational circuits have multiple outputs for the same inputs. Each output can be treated as a unique function.

Combinatorial Logic II *ZyBook Chapter 2*

- Two-level combinational logic simplification
    - Logic simplification: simplify a boolean expression before converting to a circuit to yield a smaller circuit.
    - Each AND or OR gate *input* requires two transistors
    - Simplifying can be hard to complete by hand

- **K-Map**

y = a'b' + ab'

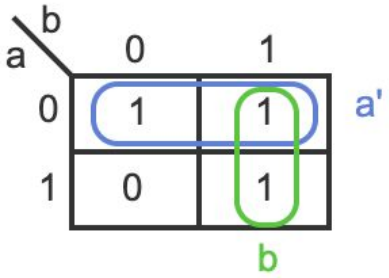
a \ b	0	1
0		a'b
1	ab'	ab

a	b	y
0	0	1
0	1	0
1	0	1
1	1	0

a \ b	0	1
0	1	0
1	1	0

- Graphical function representation that eases the simplification process for expressions involving a few variables. → Adjacently placing minterms that differ by exactly one variable.
- Simplification: i(j + j') = i(1) = i

y = ab + a'b + a'b'
y = a' + b



a'b' + a'b
a'(b' + b)
a'

a'b + ab
b(a' + a)
b

- **Rules**
  - Cover every 1 at least once using circles. Add circle's term to expression
  - Use fewest and largest circles possible

- **Three variables K-map**

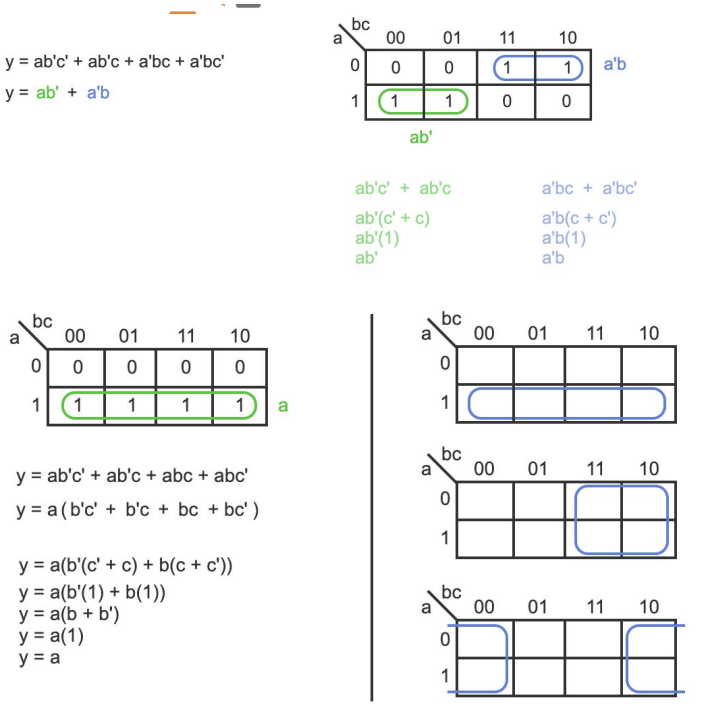
y = ab'c' + ab'c + a'bc + a'bc'

a \ bc	00	01	11	10
0		a'b'c	a'bc	
1	ab'c'		abc	abc'

a \ bc	00	01	11	10
0	0	0	1	1
1	1	1	0	0

a	b	c	y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

  - The columns don't count up in binary
  - Simplification



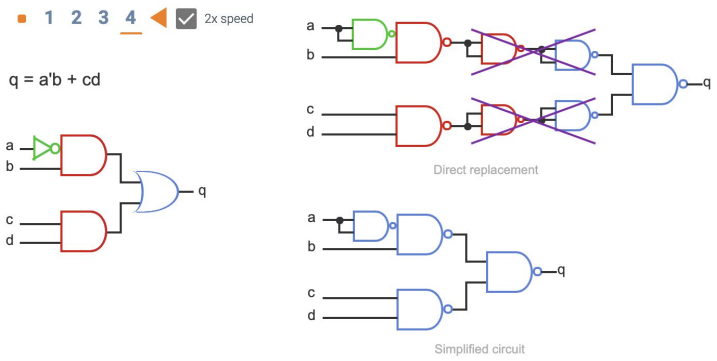
Allowed 4-cell circle shapes are a 1x4, or 2x2. An L shape does not have a corresponding algebraic simplification, so is not allowed.

- **DeMorgan's Law**

Property	Name	Description
(a + b)' = a'b'	DeMorgan's Law (for OR)	Each literal complemented, ORs become ANDs.
(ab)' = a' + b'	DeMorgan's Law (for AND)	Each literal complemented, ANDs become ORs.

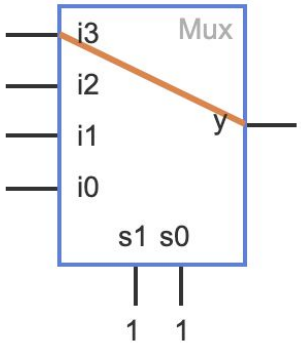
- Gates
  - XOR: outputs 1 if the input values differ
  - XNOR: output 1 if the input values are the same
  - (*universal*) NAND: opposite of AND gate. Output 0 if all are 1's. Else 1.
  - (*universal*) NOR: opposite of OR gate. Output 0 if any inputs are 1's, else 1.

- Universal Gate
  - Single gate type that can implement any combinational circuit. NAND is universal gate because it can implement NOT, AND, OR.



Finally, the designer simplifies the circuit by eliminating any sequence of NOT gate followed by a NOT gate, since such a double inversion yields the original signal (x" = x).

- **Muxes**
  - Multiplexor is a combinational circuit that passes one of multiple data input through a single output, selecting which one based on additional control inputs.



s1	s0	y
0	0	i0
0	1	i1
1	0	i2
1	1	i3

  - Mux sizes may be 2^n. Each mus requires $\log_2 N$ select inputs for N inputs.

- Decoder
  - Decoder is a combinational circuit that converts N inputs to a 1 on one of 2^N outputs.
  - Decoder m x n has n AND gates and always has 0 OR gate.
  - Enable input: 0 sets all outputs to 0s and 1 enables the decoder for normal behavior.

- **Incompletely Specified Functions**
  - Doesn't define output for every input combination.
  - All possible minterms of a function can be divided into
    - *On set:* output 1
    - *Off set*: output 0
    - *Don't Care*: output is not specified

- **On-sets and Implicants**

    - On-Set: set of minterms that define when the function is 1
    - Covers: a term covers a minterm if the term evaluates to 1 whenever the minterm does.
    - Implicant: an implicant of a function is a term that covers only minterms in the function's on-set

- Prime Implicant
    - An implicant that cannot have a literal removed without becoming a non-implicant

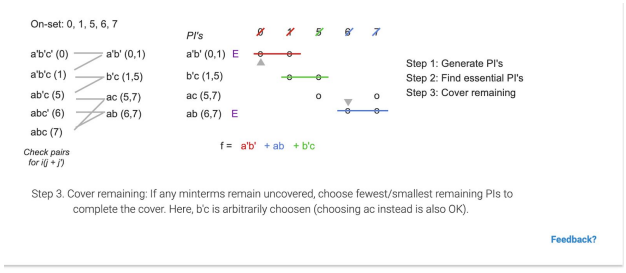
- Minimal Cover
    - Expression for the function, having the fewest terms, each with fewest literals

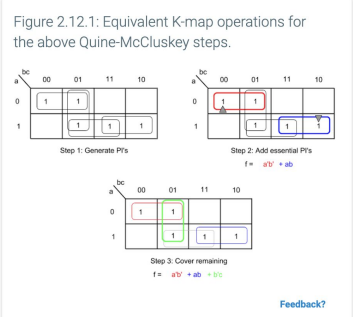
- Essential prime implicant
    - Only prime implicant to cover a particular minterm in a function's on-set. On K-map, an essential PI is a largest circle that is the only circle to cover a particular 1.

- **Quine-McCluskey**

1. *Generate PI's:* Create a table of minterms, then pairwise check minterms for i(j + j') opportunities, combining into new terms in a new column, repeating with new terms until no more combinations can be made. Each term that wasn't combined with another (minterms or new terms) is a prime implicant (PI).
2. *Find essentials:* Draw a table with PI's as rows and minterms as columns, putting a mark to indicate a PI covers a minterm. For any column with only one mark, the PI for that row is essential so is added to the cover. All minterms covered by that PI are also checked off as covered.
3. *Cover remaining:* Select minimal unadded prime implicants to cover remaining minterms.

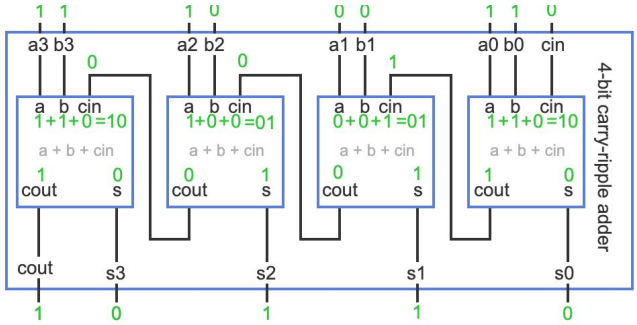
    - Generate PI;s
    - Find essentials
    - Cover remaining

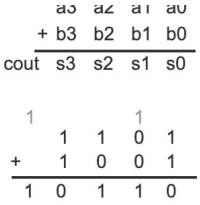


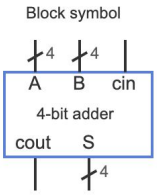


Datapath Components *ZyBook Chapter 4*

- Adders: computes A + B where A and B are N-bit numbers.
  - Carry-Ripple Adder mimics adding by hand" adds a digit's pair of bits and carry in bit, generate sum and carry out.



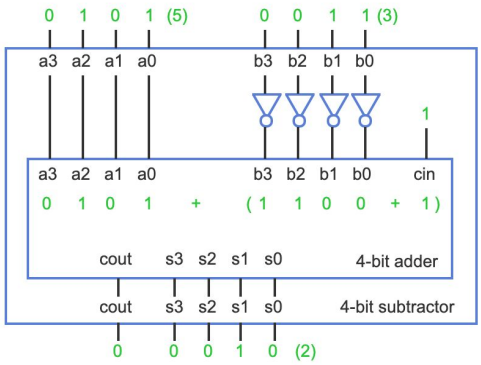




  - Full Adders: adds three bits and generates a sum and carry-out. It can be designed starting from a truth table. *Above uses four adders*.
  - Half Adders: adds two bits and generates a sum and carry-out bit. Sufficient for an incrementer.

- Signed numbers in binary
  - Signed-magnitude representation: left bit for the sign: 0 positive, 1 negative.
  - Complement: an N-digit number that also yields a sum of 1000 (N 0's) → Represent the negative.
  - Two's complement: inverts every bit and adds 1.
    - Left bit indicates the sign
    - Complement of 0000 is also 0000
    - Represents one more negative number than positive

- Subtractors
  - Computes A - B. Can be built with adders if number are two's complement representation.



A - B is A + (-B)

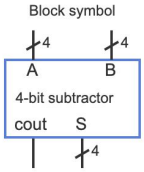
a3a2a1a0 - b3b2b1b0

5 - 3 = 5 + (-3)

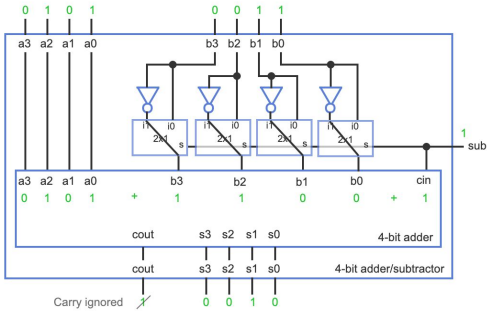
= 0101 + ((0011)' + 1)

= 0101 + (1100 + 1)

= 0010



- Single adder circuit can perform either addition or subtraction



sub = 0: addition

0011 (3) + 0100 (4)

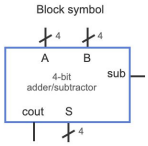
0111 (7)

sub = 1: subtraction

0101 (5) - 0011 (3)

0101(5) + 1101 (-3)

0010 (2)



An adder/subtractor is commonly represented as a block symbol.

LECTURE NOTES

LEC 2 - Combinational Logic *DeMorgan's. Consensus Theorem, Shannon's Expansions*

- Levels: Boolean Algebra (Multiple-valued logic) > Switching Algebra (2 discrete values) > Two Level Logic (sum of products, products of sums).

- Basics: no memory. Inputs and outputs can only have two discrete values.

- Def:
  - Complement: variable with ' after it.
  - Literal: variable or its complement
  - Implicant: product of literals. E.g. ABC
  - Implicate: sum of literals. E.g. (A + B + C)
  - Minterm: implicant that includes all inputs; F(A,B): AB
  - Maxterm: implicate that includes all inputs. F(A, B): A + B

- Switching Expression:
  - Boolean Algebra ≠ Switching Algebra: boolean is multiple-valued logic.
  - Equation: # literals, #variables, #operators
  - Schematic Diagram:
    - #Nets = #Variables + #Operators. *1 net for each input, 1 net for each gate*.
    - #Pin= #Literal + 2 * #Operators - 1. *1 pin for each literal and 2 pins for each gate*. *Last gate adds only 1 pin*
    - #Gate = #Operator? Consistent in your drawing.

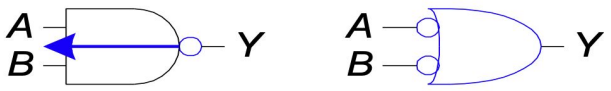
- **Fundamental Rules**

Associative laws	(a+b)+c=a+(b+c)	(a·b) ·c=a·(b·c)
Commutative laws	a+b=b+a	a·b=b·a
Distributive laws	a+(b·c)=(a+b)·(a+c)	a·(b+c)=a·b+a·c
Identity laws	a+0=a	a·1=a
Complement laws	a+a'=1	a·a'=0

  - Duality: swap all operators between (+ and *) and interchange all elements between (0, 1).

- **Demorgan's Theorem**

- (A + B)' = A'B', (AB)' = A' + B'
- Bubble Pushing (vice versa):



- **Consensus Theorem**
  - AB + AC + B'C = AB + B'C
    - Proof: LHS = AB + AC*1 + B'C = AB + AC*(B + B') + B'C
      = AB + ACB + ACB' + B'C = AB*(1+C) + (A+1)*B'C
      = AB + B'C.
  - (A + B)(A + C)(B' + C) = (A + B)(B' + C)
    - Proof: apply duality to the previous proof.
  - E.g. A + A'B = A + B
  - E.g. AB + A'CD + BCDE = AB + A'CD + BCDE + BCD = AB + A'CD + BCD = A'B + A'CD

- **Shannon's Expansion**
  - Assumes a switching algebra system. Divide a switching function into smaller functions. Pick a variable *x*, partition the switching function into two cases: x = 1 and x = 0.
    - f(x, y, z, ..) = x * f(x=1, y, z, …) + x' * f(x = 0, y, z, …)
    - f(x, y, z, ..) = (x + f(x = 0, y, z, ..))(x' + f(x = 1, y, z, ..))
  - Shannon's Expansion can decompose a switching function into **minterms**

$$f(x,y) = xf(1,y) + x'f(0,y)$$
$$= x\big(yf(1,1) + y'f(1,0)\big) + x'\big(y\big(f(0,1) + y'f(0,0)\big)\big)$$
$$= xyf(1,1) + xy'f(1,0) + x'yf(0,1) + x'y'f(0,0).$$

id	x	y	f(x,y)
0	0	0	f(0,0)
1	0	1	f(0,1)
2	1	0	f(1,0)
3	1	1	f(1,1)

    - Decompose into a truth table
  - Shannon's Expansion can decompose a switching function into **maxterms**

$$f(x,y) = (x' + f(1,y)) \cdot (x + f(0,y))$$
$$= (x' + (y' + f(1,1)) \cdot (y + f(1,0))) \cdot (x + \big(y' + f(0,1)\big) \cdot (y + f(0,0)))$$
$$= \big(x' + y' + f(1,1)\big)\big(x' + y + f(1,0)\big)\big(x + y' + f(0,1)\big)\big(x + y + f(0,0)\big)$$

- *Note: it can also be proven using laws/theorems in switching function*

- Truth Table
  - Minterms: product of all variables in the function. A minterm is equal to 1 on exactly one row of the truth table. Covers **True(1)** output.
  - Maxterm: sum of all variables in the function. A maxterm is equal to 0 on exactly one row of the truth table. Covers **False(0)** output.

LEC3 - Incompletely Specified Function and K-Map *April 23th*

- Defs:
  - Product terms, Sum terms
  - Minterm, Maxterm: product/sum of n literals which every variable appears once

- Incompletely Specified Function
  - Output of a switching function can be either a 0 or 1 for a particular combination of inputs.

- Truth Table's Canonical Form
  - On-set: all input conditions for which the output is 1
  - Off-set: all input conditions for which the output is 0
  - Don't care set: all input conditions for which the output is a "Don't care"

- **K-Map**
  - Graphical representation of the boolean expression.
    - Two variable

ID	A	B	f(A,B)
0	0	0	0
1	0	1	1
2	1	0	1
3	1	1	1

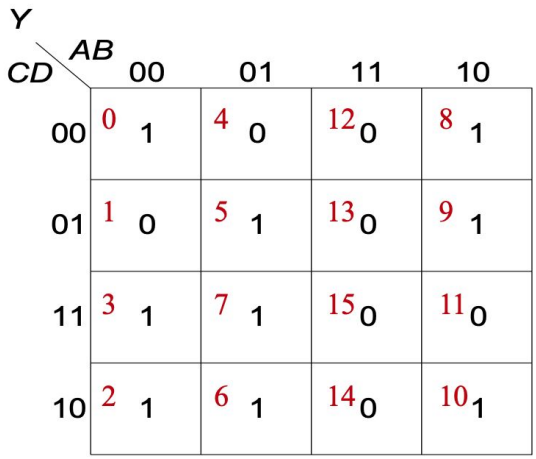
	B = 0	B = 1
A = 0	0: 0	1: 1
A = 1	2: 1	3: 1

    - Three variable

Id	a	b	c	f (a,b,c)
0	0	0	0	0
1	0	0	1	0
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	X
7	1	1	1	1

	(0,0)	(0,1)	(1,1)	(1,0)
c = 0	0: 0	2: 1	6: X	4: 1
c = 1	1: 0	3: 0	7: 1	5: 1

- Four variable



- Definitions
  - Minterms
    - Each 1 in the K-map is a minterms.
  - **Implicant:** product term that has non-empty intersection with *on-set* F and does not intersect with *off-set R*.
    - Each valid circle is an implicant (*not necessarily largest*)
  - **Prime Implicant**: an implicant that is *not covered by* any other implicant
    - Cannot have a literal removed without becoming a non-implicant
    - Each ***largest*** possible valid circle represents a prime
  - Minimal Cover: expression for the function with fewest terms, each with fewest literals.
    - Each term in a minimal cover must be a prime implicant.
    - Fewest, largest circle
  - **Essential Prime Implicant**: a prime implicant that has an element in *on-set F* but shi element is not covered by any other prime implicants.
    - Only prime implicant to cover a particular minterm in a function's on-set.

LEC4 - Universal Set and Gates *April 25th*

- Universal set: set of gates such that every switching function can be implemented with the gates in this set.
  - E.g. {AND, OR, NOT}, {AND, NOT}, {OR, NOT}.
  - E.g. {XOR} {XOR, NOT} is not universal, {XOR, AND} is universal
  - E.g. {NAND}, {NOR} gate is universal.
  - E.g. {f(x, y) = xy'}

- Other gates
  - XOR: x xor y = xy' + x'y. Parity function.

- Commutative: x XOR y = xy’ + x’y
- Associative: (x XOR y) XOR z = x XOR (y XOR z)
- 1 XOR x = x’, 0 XOR x = x
- x XOR x = 0, x XOR x’ = 1