

Lecture 16

Combinatorial Digital Logic

Back to the first Lecture:

-----APPLICATION PROGRAMMER-----

Level 6: **Application - Problem-oriented language – High Level Language**

Level 5: **Assembly Language**

-----SYSTEM PROGRAMMER-----

Level 4: **Operating System machine**

Level 3: **Instruction Set Architecture (ISA) – Machine Language**

Level 2: **Micro architecture Level (Registers, ALU → Data Path)**

Level 1: Digital Logic Level (AND, OR's, Registers, Gate Level Logic)

Level 0: **Device Level (Transistors)**

- Digital Logic is how all instructions and calculations are implemented.
 - Underlying representation of storage, memory, calculations, *etc.*
 - Study to understand how control logic, registers, actually function. Limitations and challenges are better understood.

• Basic Logic Gates



AND gate

| A | B | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |



NAND gate

| A | B | F |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



OR gate

| A | B | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |



NOR gate

| A | B | F |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |



NOT gate

| A | F |
|---|---|
| 0 | 1 |
| 1 | 0 |

Truth table



XOR gate

| A | B | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

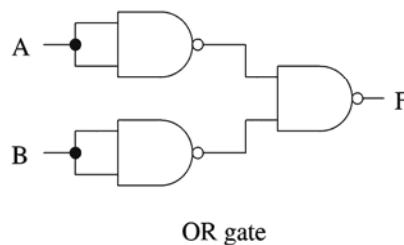
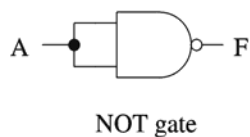
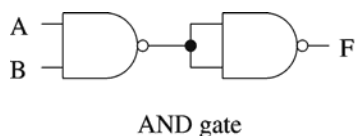
Logic symbol

Truth table

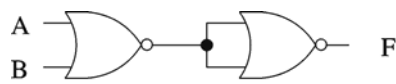
- *NAND* and *NOR* gates require only 2 transistors.
 - *AND* and *OR* need 3 transistors!
 - *NAND* and *NOR* are preferred for size. Smaller gates means more can be fitted on a single silicon die. More efficient in terms of area, which means cost.
- Boolean Logic Precedence:
 - $NOT > AND > OR$
 - $F = \bar{A}B + A\bar{B} = ((\bar{A})B) + (A(\bar{B}))$

- **Universal (Complete) Sets**

- A set of gates is *complete/universal* if you can implement **any** logical function using only the types of gates in the set.
 - You can use as many gates as you want to implement the function.
- *Minimal complete set* is a complete set with no redundant elements.
- Some examples of complete sets:
 - {*AND*, *OR*, *NOT*} – this is a complete set, but not a *minimal* complete set.
 - {*AND*, *NOT*}
 - {*OR*, *NOT*}
 - {*NAND*}
 - {*NOR*}
- Ex: Proof that *NAND* is universal by building gates that form a universal set.
 - It is sufficient to build a *NOT* gate and either an *OR* or an *AND* gate.



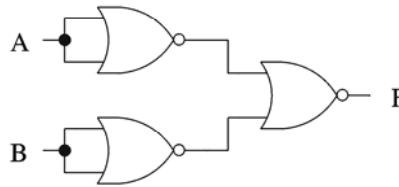
- Ex: Proof that *NOR* is universal by building gates that form a universal set.
 - It is sufficient to build a *NOT* gate and either an *OR* or an *AND* gate.



OR gate



NOT gate



AND gate

- How to prove any gate is or is not a universal set.

- First, examine the truth tables for the gates given.
- Write the Boolean expression for the gate.
- Find an input combination from which you can create a *NOT* gate. If you do not find a case, then it is not a universal set.
- Build the *NOT* gate, by setting the proper values on the inputs.
- Find a combination of inputs to build either an *AND* or an *OR* gate. If you do not find a way then it is not a universal set.

Ex: Show that the operation (gate) @ described by the following truth table is universal. Use, if needed, either constant 0 or constant 1.

| x | y | x@y |
|---|---|-----|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Solution:

Boolean expression $x @ y = x + \bar{y}$

a) To make a *NOT* gate using the given @ gate: let $x = 0$, $y = a$; output is $0 @ a = 0 + \bar{a} = \bar{a}$.

b) To an make *OR* gate using given @ gate, you need two @ gates:

The first @ gate is the one saw above: let $x = 0$, $y = c$; output would be $0 @ c = 0 + \bar{c} = \bar{c}$.

The second @ gate: let $x = b$, $y = \bar{c}$ (the output of the first @ gate); output would be $b @ \bar{c} = b + \bar{c} = b + c$.

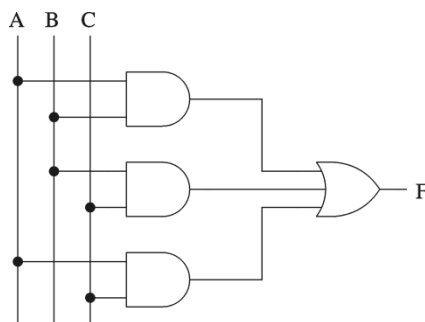
Since $\{NOT, OR\}$ is a universal set and we can make them using @ gates, the @ gate is by itself a universal set.

- Any type of gates can be used in combination to create a logical function.
 - Some gates are more efficient than others. (i.e., *NAND* and *NOR* are smaller in area than *AND* and *OR*).
- Logical functions can be expressed in several ways (all of which are functionally equivalent):
 - Truth tables.
 - Presents the output function in terms of all combinations of the input variables.
 - Logical expressions, aka *switching statement/expression*, aka *Boolean expressions*.
 - Presents the output function in terms of complemented and uncomplemented variables combined into terms with '+' (*OR*), '•' (*AND*), '¯' (*NOT*).
 - Gate diagrams.
 - Presents the function in terms of logical operations and wires.
 - Graphical form (*Karnaugh maps*).
 - Presents the function in terms of a grid with the output function in corresponding grid squares.
- Ex: Majority function
 - Input: Three 1-bit inputs $\{A, B, C\}$
 - Output: F , is 1 whenever the majority of inputs is 1.
 - Logical expression: $F = A \cdot B + B \cdot C + A \cdot C$

Truth Table

| A | B | C | F |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

Gate Diagram



- **Definitions:**

- *Switching statement/expression*: truth table written in Boolean algebra form.
- *Complement*: opposite of variable value; $0 \rightarrow 1, 1 \rightarrow 0$ (*NOT* operation); typically written with overbar ' $\bar{\quad}$ ', or ' $'$ ' (e.g., x').
- *Literal*: each occurrence of a variable in switching statement.
 - Ex: $x + y$ has 2 literals.
 - Ex: $x' y + z$ has 3 literals.
 - Ex: $x y + x' z$ has 4 literals.

- **Simplifying Boolean Expressions**

- There is no formula for simplifying Boolean expressions. The key is to learn the Identities.

- Double Complement: $(A')' = A$
- Idempotence: $A \cdot A = A$ $A + A = A$
- Inverse: $A \cdot A' = 0$ $A + A' = 1$
- Null (or Dominance): $A \cdot 0 = 0$ $A + 1 = 1$
- Identity: $A + 0 = A$ $A \cdot 1 = A$
- Distributive: $A \cdot (B + C) = A \cdot B + A \cdot C$ $A + (B \cdot C) = (A + B) \cdot (A + C)$
- Commutative: $A \cdot B = B \cdot A$ $A + B = B + A$
- Associative: $(A \cdot B) \cdot C = A \cdot (B \cdot C)$ $(A + B) + C = A + (B + C)$
- Absorption Law: $A + (A \cdot B) = A$
 (proof: $A + (A \cdot B) = (A + A) \cdot (A + B) = (A) \cdot (A + B) = A + (0 \cdot B) = A + 0 = A$)

 $A \cdot (A + B) = A$
 (proof: $A \cdot (A + B) = (A \cdot A) + (A \cdot B) = (A) + (A \cdot B) = A \cdot (1 + B) = A \cdot 1 = A$)
 Note that in these expressions, either A or B may stand for any complex Boolean expression.
- de Morgan's theorem: $(A + B)' = A' \cdot B'$ $(A \cdot B)' = A' + B'$

This theorem is widely used in Boolean logic design. Stated in words it is:

To 'invert' (negate) an expression, you replace the AND operator with the OR operator (and vice versa) and invert the individual terms.

The theorem holds for any number of terms, so:

$$(A + B + C)' = ((A + B) + C)' = ((A + B)') \cdot C' = A' \cdot B' \cdot C';$$

and similarly:

$$(A \cdot B \cdot C \cdot \dots \cdot X)' = A' + B' + C' + \dots + X'.$$

You may have noticed by now that rules are often given in pairs. It makes sense that in a binary system there is some kind of symmetry between the two operators *AND* and *OR*. For Boolean algebra this symmetry is called *Duality*. Every equation has its dual which one can generate by replacing the *AND* operators with *ORs* (and vice versa), and the constants 0 with 1 (and vice versa). For example, the dual equation of the important simplifying rule $A + (A \cdot B) = A$ is $A \cdot (A + B) = A$.

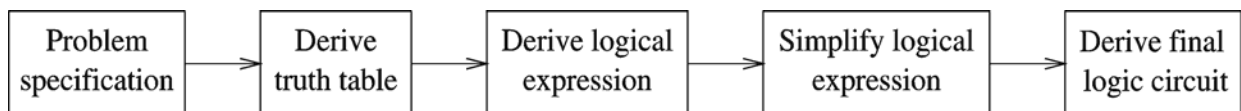
Do not mix up or get confused between a dual expression which is generated by the above rules and the complement (or inverted) expression which is generated by applying the *NOT* operator. The rules are similar, but they mean very different things.

○ To simplify a Boolean expression using logic:

- First look for terms that you can group together and factor out elements that evaluate to 1.
 - Ex: $F(A,B,C) = ABC + A'BC = (A + A')BC = BC$ since $(A + A') = 1$.
 - Ex: $F(A,B,C) = A'C' + A'BC' = A'C'(1 + B) = A'C'$ since $(1 + B) = 1$.
- Second, if you cannot find any terms to group together and simplify, then multiply elements by identities equivalent to 1 (but only using variables which do not appear in the term).
 - Ex: Expanding $F(A,B,C) = AB + A'C + BC = AB + A'C + BC(A + A')$
 $= AB + A'C + ABC + A'BC = AB(1 + C) + A'C(1 + B) = AB + A'C$.
- When all else fails, the best bet is to expand out all the terms into *minterms* (terms where all input variables appear together) and then regroup them.

• **The design of any logical process involves the following steps:**

- Problem specification (inputs and outputs of function / black box are given).
- Truth table derivation (write out all combinations of inputs, and the corresponding output values).
- Derivation of logical expression (Boolean algebra, Karnaugh maps).
- Simplification of logical expression (Boolean algebra, Karnaugh maps).
- Implementation (draw out circuit using logic gates).



- Derivation of logical expressions from truth tables:

- sum-of-products (SOP) form:

- Write an *AND* term for each input combination that produces a 1 output.
 - Write the input variable if its value is 1; write its complement otherwise.
- *OR* the *AND* terms to get the final expression.
- Ex: Boolean expression for majority function $F = A'BC + AB'C + ABC' + ABC$.

Truth Table

| A | B | C | F |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

- Four product terms, because there are 4 rows with a 1 output.
- These can be simplified to $F = AB + BC + AC$:

Since, by the Idempotence Law,

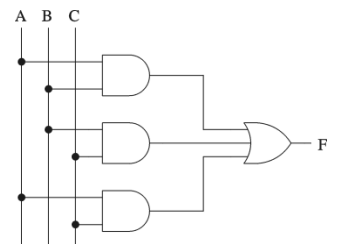
$([ABC + ABC] + ABC) = (ABC + ABC) = ABC$, we can write

$$F = A'BC + AB'C + ABC' + \mathbf{ABC} = A'BC + AB'C + ABC' + ([ABC + ABC] + ABC).$$

Grouping terms,

$$\begin{aligned} F &= (ABC + ABC') + (ABC + AB'C) + (ABC + A'BC) \\ &= AB(C + C') + A(B + B')C + (A + A')BC = AB + BC + AC. \end{aligned}$$

- Each term written, where a 1 appears in the output, is called a *minterm*. For example, $A'BC$ above is a minterm.
- SOP form means that the gate network is a 2-level *AND-OR* gate network.
- SOP can also be written in the shorthand form of $F = \Sigma m(\dots)$, where the contained values are the decimal values of the inputs.
 - Ex: for the majority function $F = \Sigma m(3,5,6,7)$



○ product-of-sums (POS) form:

- Dual of the SOP form.
- Write an *OR* term for each input combination that produces a 0 output.
 - Write the input variable if its value is 0; write its complement otherwise.
- Ex: $F = (A + B + C) (A + B + C') (A + B' + C) (A' + B + C)$

Truth Table

| A | B | C | F |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

- Four sum terms: Because there are 4 rows with a 0 output.
- These can be simplified to $F = (A + B) (B + C) (A + C)$:

Since, by the Idempotence Law,

$$[(A + B + C) (A + B + C)] (A + B + C) = [(A + B + C)] (A + B + C) = (A + B + C),$$

we can write,

$$\begin{aligned} F &= (A + B + C) (A + B + C') (A + B' + C) (A' + B + C) \\ &= [(A + B + C) (A + B + C)] (A + B + C) (A + B + C') (A + B' + C) (A' + B + C) . \end{aligned}$$

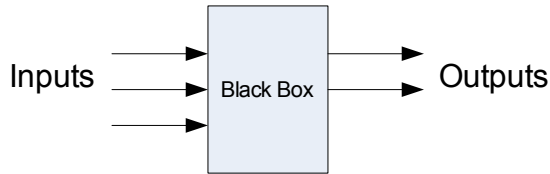
Grouping terms,

$$\begin{aligned} F &= [(A + B + C) (A + B + C')] [(A + B + C) (A' + B + C)] [(A + B + C) (A + B' + C)] \\ &= [(A + B) + (C \cdot C')] [(B + C) + (A \cdot A')] [(A + C) + (B \cdot B')] \\ &= [(A + B) + 0] [(B + C) + 0] [(A + C) + 0] = (A + B) (B + C) (A + C) . \end{aligned}$$

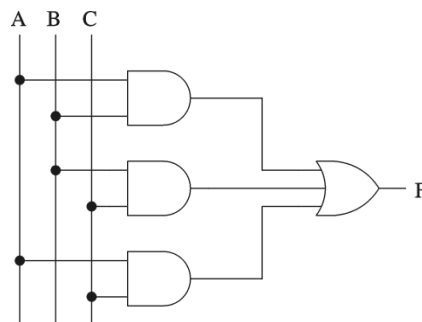
- Each term in the expression is called a *maxterm*. For example, $(A + B + C)$ above is a maxterm.
- POS form means that the gate network is a 2-level *OR-AND* gate network.
- POS can also be written in the form of $F = \Pi M (\dots)$, where the contained values are the decimal values of the inputs.
 - Ex: for the majority function $F = \Pi M (0,1,2,4)$.

2-level Gate Networks

- Basic gate networks are implemented in 2-levels.
 - Imagine a black box, with the input variables entering the box, and the output variables exiting the box.



- Inside the black box, the logic gates and the implementation of the function is designed with logic gates.
- For SOP expressions of minterms, a 2-level *AND-OR* gate network is created inside the box.
 - The input variables enter into the first level of gates, which are *AND* gates. Each term of the Boolean expression is created. Notice that the Boolean expression is written in this form:
 - $F = A'BC + AB'C + ABC' + ABC$ (SUM OF THE PRODUCTS)
 - The output of all the *AND* gates enter into an *OR* gate to create the output value.
 - Ex: Majority function (after simplifying). Inputs to the black box are *A*, *B* and *C*; the output is *F*.



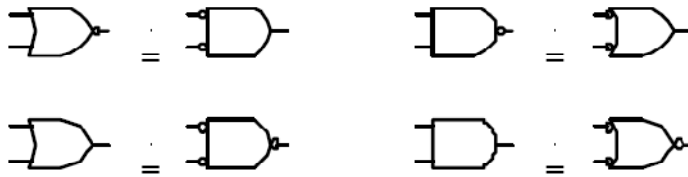
- For POS expressions of maxterms, a 2-level *OR-AND* gate network is created inside the box.
 - The input variables enter into the first level of gates, which are *OR* gates. Each term of the Boolean expression is created. Notice that the Boolean expression is written in this form:
 - $F = (A + B + C) (A + B + C') (A + B' + C) (A' + B + C)$ (PRODUCT OF THE SUMS)
 - The output of all the *OR* gates enter into an *AND* gate to create the output value.

- According to De Morgan's law the following is true. The circles denote the complement of a variable (inverter).

DeMorgan's Law:

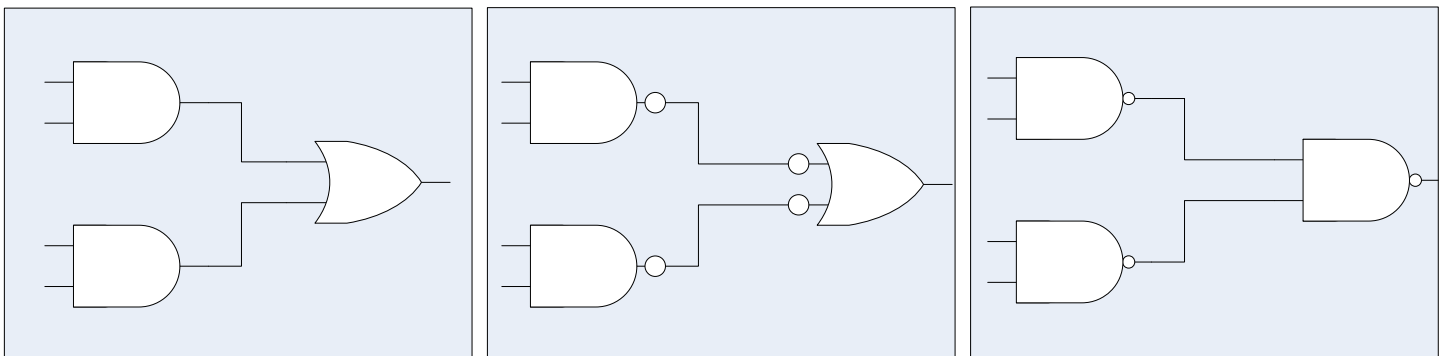
$$(a + b)' = a' b' \quad (a b)' = a' + b'$$

$$b + b = (a' b')' \quad (a b) = (a' + b')'$$



NAND-NAND

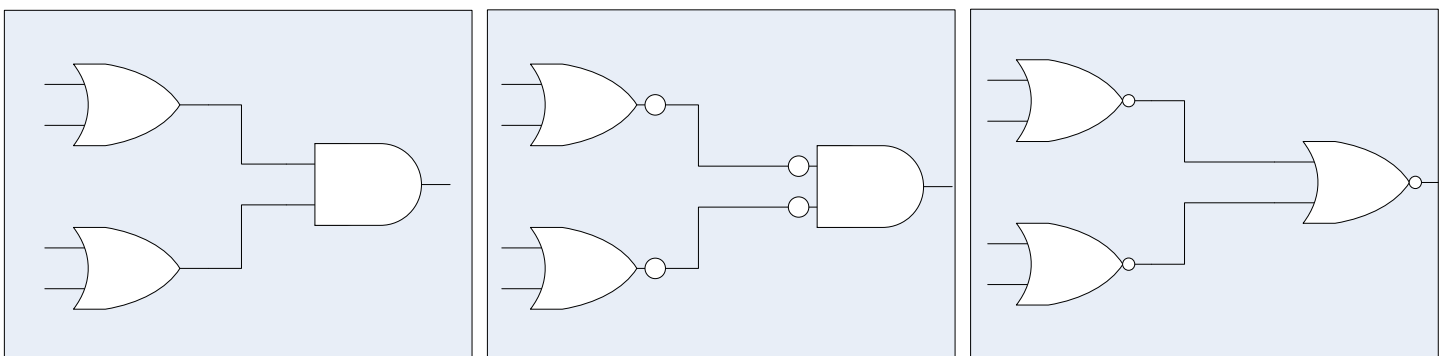
- Given the rules above, then if the output of the AND gates is inverted and then the values are again inverted at the input of the OR gate, the network becomes a NAND-NAND network.



- To create a NAND-NAND network you follow the same rules as for AND-OR (SOP) and then negate the outputs of the AND gates and the inputs of the OR gate as shown above.

NOR-NOR

- Given de Morgan's rules above, then if the output of the OR gates is inverted and then the values are again inverted at the input of the AND gate, the network becomes a NOR-NOR network.



- To create an NOR-NOR network you follow the same rules as for OR-AND (POS) and then negate the outputs of the OR gates and the inputs of the AND gate as shown above.