Boolean Algebra and Logic Gates

CS207 Lecture 2

James YU

Feb. 26, 2020

南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Binary logic

- Binary logic deals with variables that take on two discrete values and with logical operations.
- Three basic logical operations:
  - **AND**: $x \cdot y = z$ or $xy = z$.
  - **OR**: $x + y = z$.
  - **NOT**: $x' = z$

AND			OR			NOT	
x	y	$x \cdot y$	x	y	$x + y$	x	x'
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
1	0	0	1	0	1		
1	1	1	1	1	1		

Boolean Algebra

- The previous binary logic is *two-valued Boolean algebra*.
  - On a set of two elements: `0` and `1`.
  - With rules for the three binary operators: `+`, `·` and `'`.
- Common properties:
  - $A + 0 = A$ and $A \cdot 1 = A$.
  - $A + 1 = 1$ and $A \cdot 0 = 0$.
  - $A + A' = 1$ and $A \cdot A' = 0$.
  - $A + A = A$ and $A \cdot A = A$.
  - $(A')' = A$.

Postulates

- **Closure**: A set S is closed with respect to a binary operator if, for every pair of elements of S, the binary operator specifies a rule for obtaining a unique element of S.
- **Associative law**: $A+(B+C)=(A+B)+C$ and $A(BC)=(AB)C$.
- **Commutative law**: $A+B=B+A$ and $AB=BA$.
- **Identity element**: A set S is to have an identity element with respect to a binary operation $*$ on S, if there exists an element $E \in S$ with the property $E*A=A*E=A$.
  - Element `0` is an identity element of `+`, and `1` is an identity element of `·`.
- **Distributive law**: $A(B+C)=AB+AC$ and $A+BC=(A+B)(A+C)$.
- **DeMorgan**: $(A+B)'=A'B'$ and $(AB)'=A'+B'$.
- **Absorption**: $A+AB=A$ and $A(A+B)=A$.

Duality property

- Every algebraic expression deducible from the postulates of Boolean algebra remains valid if the operators and identity elements are interchanged.
- Change `+` to `·` and vice versa.
- Change `0` to `1` and vice versa.
  - $A + A' = 1 \rightarrow A \cdot A' = 0$.
  - $A + B = B + A \rightarrow AB = BA$.
  - $A(B + C) = AB + AC \rightarrow A + BC = (A + B)(A + C)$.
  - $(A + B)' = A'B' \rightarrow (AB)' = A' + B'$.

Boolean function

- Binary variables have two values, either `0` or `1`.
- A Boolean function is an expression formed with *binary variable*s, the two *binary operator*s **AND** and **OR**, one *unary operator* **NOT**, *parentheses* and *equal sign*.
- The value of a function may be `0` or `1`, depending on the values of variables present in the Boolean function or expression.
- Example: $F = AB'C$.
  - $F = 1$ when $A = C = 1$ and $B = 0$,
  - otherwise $F = 0$.

Boolean function

- Boolean functions can also be represented by truth tables.
    - Tabular form of the values of a Boolean function according to the all possible values of its variables.
- n number of variables $\rightarrow 2^n$ combinations of 1's and 0's
- One column representing function values according to the different combinations.
- Example: $F = AB + C$.

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Boolean function simplification

- A Boolean function from an algebraic expression can be realized to a logic diagram composed of logic gates.
- Minimization of the number of literals and the number of terms leads to less complex circuits as well as less number of gates.
  - We first try use postulates and theorems of Boolean algebra to simplify.

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

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

$$\begin{aligned} F &= XYZ + XY'Z + XYZ' \\ &= XZ(Y + Y') + XY(Z + Z') \\ &= XZ + XY = X(Y + Z) \end{aligned}$$

- Each Boolean function has one representation in truth table, but a variety of ways in algebraic form.

Algebraic manipulation

- Reduce the total number of terms and literals.
- Usually not possible by hand for complex functions, use computer minimization program.
- More advanced techniques in the next lectures.

Boolean function complement

- Complement a Boolean function from F to F'.
  - Change `0`'s to `1`'s and vice versa in the truth table.
  - Use Use DeMorgan's theorem for multiple variables.
- Example: $F = x'yz' + x'y'z$.

Complement:

$$
\begin{aligned}
F' &= (x'yz' + x'y'z)' \\
&= (x'yz')'(x'y'z)' \\
&= (x + y' + z)(x + y + z')
\end{aligned}
$$

Dual:

$$F^* = (x' + y + z')(x' + y' + z)$$

Canonical forms

- Logical functions are generally expressed in terms of different combinations of logical variables with their true forms as well as the complement forms: x and x'.
- An arbitrary logic function can be expressed in the following forms, called *canonical forms*:
  - *Sum of products* (SOP), and
  - *Product of sums* (POS).
- What are the products and sums?

Canonical forms

- The logical product of several variables on which a function depends is considered to be a product term.
  - Called *minterm*s when all variables are involved: For x and y, xy, $x'y$, xy', and $x'y'$ are all the minterms.
- The logical sum of several variables on which a function depends is considered to be a sum term.
  - Called *maxterm*s when all variables are involved: For x and y, $x+y$, $x'+y$, $x+y'$, and $x'+y'$ are all the maxterms.
- **SOP**: The logical sum of two or more logical product terms is referred to as a sum of products expression.
- **POS**: The logical product of two or more logical sum terms is referred to as a product of sums expression.

Minterms

- In the minterm, a variable will possess the value `1` if it is in true or uncomplemented form, whereas, it contains the value `0` if it is in complemented form.

A	B	C	Minterm
`0`	`0`	`0`	$A'B'C'$
`0`	`0`	`1`	$A'B'C$
`0`	`1`	`0`	$A'BC'$
`0`	`1`	`1`	$A'BC$
`1`	`0`	`0`	$AB'C'$
`1`	`0`	`1`	$AB'C$
`1`	`1`	`0`	ABC'
`1`	`1`	`1`	ABC

- It possesses the value of `1` for only one combination of n input variables
  - The rest of the $2^n - 1$ combinations have the logic value of `0`.

Minterms

- *Canonical SOP* expression, or *sum of minterms*: A Boolean function expressed as the logical sum of all the minterms from the rows of a truth table with value 1.

A	B	C	F	Minterms
0	0	0	0	$A'B'C'$
0	0	1	1	$A'B'C$
0	1	0	0	$A'BC'$
0	1	1	1	$A'BC$
1	0	0	0	$AB'C'$
1	0	1	1	$AB'C$
1	1	0	1	ABC'
1	1	1	1	ABC

- $F = AB + C = A'B'C + A'BC + AB'C + ABC' + ABC = \sum(1, 3, 5, 6, 7)$.
  - A compact form by listing the corresponding decimal-equivalent codes of the minterms.

Minterms

- The canonical sum of products form of a logic function can be obtained by using the following procedure.
  1. Check each term in the given logic function. Retain if it is a minterm, continue to examine the next term in the same manner.
  2. Examine for the variables that are missing in each product which is not a minterm.
  3. If the missing variable in the minterm is X, multiply that minterm with $(X + X')$.
     - Example: $A + B \rightarrow A(B + B') + B(A + A')$
  4. Multiply all the products and discard the redundant terms.

Minterms

- Example: $F(A,B,C,D) = AB + ACD$.

$$
\begin{aligned}
F(A,B,C,D) &= AB + ACD \\
&= AB(C + C')(D + D') + ACD(B + B') \\
&= (ABC + ABC')(D + D') + ABCD + AB'CD \\
&= ABCD + ABCD' + ABC'D + ABC'D' + ABCD + AB'CD \\
&= ABCD + ABCD' + ABC'D + ABC'D' + AB'CD
\end{aligned}
$$

Maxterms

- In the maxterm, a variable will possess the value `0`, if it is in true or uncomplemented form, whereas, it contains the value `1`, if it is in complemented form.

A	B	C	Maxterm
`0`	`0`	`0`	$A + B + C$
`0`	`0`	`1`	$A + B + C'$
`0`	`1`	`0`	$A + B' + C$
`0`	`1`	`1`	$A + B' + C'$
`1`	`0`	`0`	$A' + B + C$
`1`	`0`	`1`	$A' + B + C'$
`1`	`1`	`0`	$A' + B' + C$
`1`	`1`	`1`	$A' + B' + C'$

- It possesses the value of `0` for only one combination of n input variables
  - The rest of the $2^n - 1$ combinations have the logic value of `1`.

Maxterms

- *Canonical POS* expression, or *product of maxterms*: A Boolean function expressed as the logical product of all the maxterms from the rows of a truth table with value `0`.
- $F = (A + B + C)(A + B' + C)(A' + B + C') = \prod(0, 2, 5)$.
  - A compact form by listing the corresponding decimal-equivalent codes of the maxterms.

Maxterms

- Example: $F(A,B,C,D) = A + B'C$.

$$
\begin{aligned}
F(A,B,C,D) &= A + B'C \\
&= (A+B')(A+C) \\
&= (A+B'+CC')(A+C+BB') \\
&= (A+B'+C)(A+B'+C')(A+B+C)(A+B'+C) \\
&\qquad\qquad \text{using the distributive property: } X+YZ=(X+Y)(X+Z) \\
&= (A+B'+C)(A+B'+C')(A+B+C)
\end{aligned}
$$

Derive from a truth table

A	B	C	F	Minterm	Maxterm
0	0	0	0		$A+B+C$
0	0	1	0		$A+B+C'$
0	1	0	1	$A'BC'$	
0	1	1	0		$A+B'+C'$
1	0	0	1	$AB'C'$	
1	0	1	1	$AB'C$	
1	1	0	1	ABC'	
1	1	1	0		$A'+B'+C'$

- The final **canonical SOP** for the output F is derived by summing or performing an **OR** operation of the four product terms as shown below:
  - $F = A'BC' + AB'C' + AB'C + ABC' = \sum(2,4,5,6)$.
- The final **canonical POS** for the output F is derived by summing or performing an **AND** operation of the four sum terms as shown below:
  - $F = (A+B+C)(A+B+C')(A+B'+C')(A'+B'+C') = \prod(0,1,3,7)$.

Conversion between minterms and maxterms

- Minterms are the complement of corresponding maxterms: $m_i = M_i'$.
  - Example: $A' + B' + C' = (ABC)'$.

$$
\begin{aligned}
F(A,B,C) &= \sum(2,4,5,6) = m_2 + m_4 + m_5 + m_6 \\
&= A'BC' + AB'C' + AB'C + ABC' \\
F'(A,B,C) &= \sum(0,1,3,7) = m_0 + m_1 + m_3 + m_7 \\
F(A,B,C) &= \left(F(A,B,C)\right)' = (m_0 + m_1 + m_3 + m_7)' \\
&= m_0' m_1' m_3' m_7' \\
&= M_0 M_1 M_3 M_7 \\
&= \prod(0,1,3,7) \\
&= (A+B+C)(A+B+C')(A+B'+C')(A'+B'+C').
\end{aligned}
$$

Other logic operators

- When the binary operators AND and OR are applied on two variables A and B, they form two Boolean functions AB and $A + B$ respectively.

Other logic operators

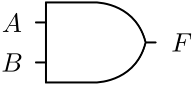
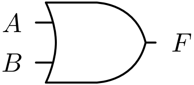
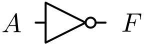
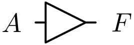
- When the three operators AND, OR, and NOT are applied on two variables A and B, they form 16 Boolean functions:

Boolean Functions	Operator Symbol	Name	Comments
$F_0 = 0$		Null	Binary constant 0
$F_1 = xy$	$x \cdot y$	AND	x and y
$F_2 = xy'$	x/y	Inhibition	x, but not y
$F_3 = x$		Transfer	x
$F_4 = x'y$	y/x	Inhibition	y, but not x
$F_5 = y$		Transfer	y
$F_6 = xy' + x'y$	$x \oplus y$	Exclusive-OR	x or y, but not both
$F_7 = x + y$	$x + y$	OR	x or y
$F_8 = (x + y)'$	$x \downarrow y$	NOR	Not-OR
$F_9 = xy + x'y'$	$(x \oplus y)'$	Equivalence	x equals y
$F_{10} = y'$	y'	Complement	Not y
$F_{11} = x + y'$	$x \subset y$	Implication	If y, then x
$F_{12} = x'$	x'	Complement	Not x
$F_{13} = x' + y$	$x \supset y$	Implication	If x, then y
$F_{14} = (xy)'$	$x \uparrow y$	NAND	Not-AND
$F_{15} = 1$		Identity	Binary constant 1

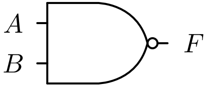
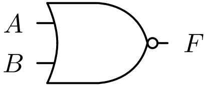
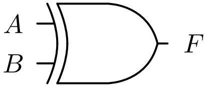
Digital logic gates

- As Boolean functions are expressed in terms of AND, OR, and NOT operations, it is easier to implement the Boolean functions with these basic types of gates.
  - It is possible to construct other types of logic gates.
- The following factors are to be considered for construction of other types of gates.
  - The **feasibility** and economy of producing the gate with physical parameters.
  - The possibility of **extending** to more than two inputs.
  - The basic properties of the binary operator such as **commutability** and **associability**.
  - The ability of the gate to **implement Boolean functions** alone or in conjunction with other gates.

Digital logic gates

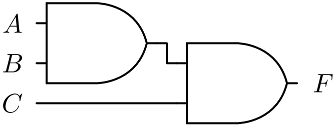
			A	B	F
AND		$F = AB$	0	0	0
			0	1	0
			1	0	0
			1	1	1
OR		$F = A + B$	0	0	0
			0	1	1
			1	0	1
			1	1	1
NOT		$F = A'$	0	-	1
			1	-	0
Buffer		$F = A$	0	-	0
			1	-	1

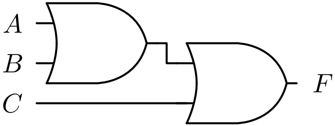
Digital logic gates

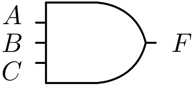
			A	B	F
NAND		$F = (AB)'$	0	0	1
			0	1	1
			1	0	1
			1	1	0
NOR		$F = (A + B)'$	0	0	1
			0	1	0
			1	0	0
			1	1	0
XOR		$F = AB' + A'B$ $= A \oplus B$	0	0	0
			0	1	1
			1	0	1
			1	1	0

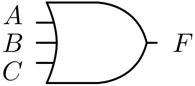
Multiple input logic gates

- A gate can be extended to have multiple inputs if its binary operation is commutative and associative.
- AND and OR gates are both commutative and associative.
  - $F = ABC = (AB)C$.
  - $F = A + B + C = (A + B) + C$.



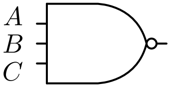




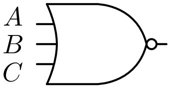


Multiple input logic gates

- The NAND and NOR functions are the complements of AND and OR functions respectively.
  - They are commutative, but **not associative**.
  - $((AB)'C)' \neq (A(BC)')'$: does not follow associativity.
  - $((A+B)'+C)' \neq (A+(B+C)')'$: does not follow associativity.
- We modify the definition of multi-input NAND and NOR:



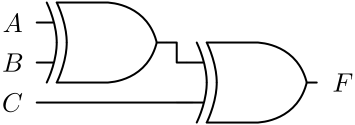
$$F = (ABC)' = A' + B' + C'$$

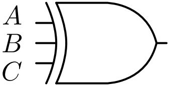


$$F = (A+B+C)' = A'B'C'$$

Multiple input logic gates

- The XOR gates and equivalence gates both possess commutative and associative properties.
  - Gate output is low when even numbers of `1`'s are applied to the inputs, and when the number of `1`'s is odd the output is logic `0`.
  - Multiple-input exclusive-OR and equivalence gates are uncommon in practice.

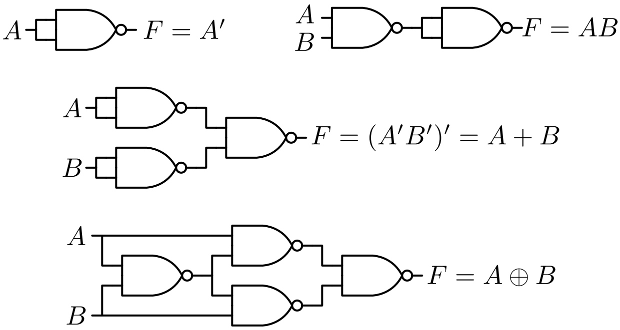




$$F = A \oplus B \oplus C$$

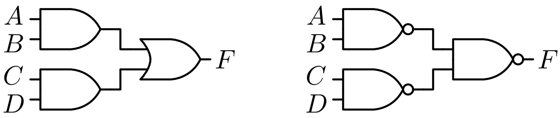
Universal gates

- NAND gates and NOR gates are called *universal gates* or *universal building blocks*.
  - Any type of gates or logic functions can be implemented by these gates.



Universal gates

- Universal gates are easier to fabricate with electronic components.
  - Also reduce the number of varieties of gates.
- Example: $F = AB + CD$ requires two AND and one OR gates.
  - Or three NAND gates.
  - $F = AB + CD = ((AB + CD)')' = ((AB)'(CD)')'$



Verilog

- Verilog is a hardware Description Language (HDL) that consists of digital logic. It is intended to be used for simulations, timing analysis, testing, and synthesis.
- The basic building block of Verilog is the module statement.

```
module <module_name>(<input_list>, <output_list>);
input <input_list>;
output <output_list>;

endmodule
```

Verilog modules

- Example: A module that takes in three inputs: two 5-bit operands called `a` and `b`, and an enable input called `en`.

```verilog
module comparator(a, b, en, a_gt_b);
input [4:0] a, b;
input en;
output a_gt_b;

endmodule
```

- In this state, the module just does nothing, for two reasons.
  - There is no code in the body of the module.
  - defining a module in and of itself does nothing (unless it is the top level module).
    - We need to create an instance of a module in our design to actually use it.

Instantiating modules

- We can include an instance of a module within another module using the following syntax:

```
<module_name> <instance_name>(<port_list>);
```

- Example: to instantiate a comparator module with the name `comp1`, input wires `in1`, `in2`, and `en`, and an output wire `gt`, we could write:

```
comparator comp1(in1, in2, en, gt);
```

  - This instantiation depends on the ordering of the ports in the comparator module.

Instantiating modules

- There is an alternate syntax for instantiating modules which does not depend on port ordering:

```
<module_name> <instance_name>(.<port_name>(ioname), ...);
```

- Continuing from the last example, we could instead write:

```
comparator comp1(.b(in2), .a(in1), .en(en), .a_gt_b(gt));
```

```
comparator comp1(in1, in2, en, gt);
```

Comments

- Comments in Verilog are exactly the same as in Java.

```
// This is a comment
/* Multi-line
   comment */
```

Numerical

- Many modules will contain numerical literals.
- In Verilog, numerical literals are unsigned 32-bit numbers by default.
- You should get into the habit of declaring the width of each numerical literal.

```
/* General syntax:
   <bits>'<base><number>
   where <base> is generally b, d, or h */

wire [2:0] a = 3'b111; // 3 bit binary
wire [4:0] b = 5'd31; // 5 bit decimal
wire [31:0] c = 32'hdeadbeef; // 32 bit hexadecimal
```

Constants

- We can use `` `define `` to define global constants in our code.
  - **Do not append a semicolon** to the `` `define `` statement.

```verilog
`define RED 2'b00 // DON'T add a semicolon to these statements
`define WHITE 2'b01
`define BLUE 2'b10

wire [1:0] color1 = `RED;
wire [1:0] color2 = `WHITE;
wire [1:0] color3 = `BLUE;
```

Wires

- To start with, we have two kinds of data types in our modules: *wire*s and *register*s.
- You can think of *wire*s as modeling physical wires.

```
wire        a_wire;
wire [1:0] two_bit_wire;
wire [4:0] five_bit_wire;
```

Wires

- We then use the `assign` statement to connect them to something else.
    - Assume that we are in a module that takes a two bit input named `two_bit_input`:

```
wire       a_wire;
wire [1:0] two_bit_wire;
wire [4:0] five_bit_wire;

assign two_bit_wire = two_bit_input;
// Connect a_wire to the lowest bit of two_bit_wire
assign a_wire = two_bit_wire[0];
/* {} is concatenation - 3 MSB will be 101, 2 LSB will be
connected to two_bit_wire */
assign five_bit_wire = {3'b101, two_bit_wire};
// This is an error! You cannot assign a wire twice!
// assign a_wire = 1'b1;
```

Wires

- There is a shortcut that is sometimes used to declare and assign a wire at the same time:

```
// Declares gnd, and assigns it to 0
wire gnd = 1'b0;
```