Logic Circuits and Boolean Algebra

Chapter 2

Outline

- Boolean Functions and Truth Tables
- AND, OR, NOT Gates
- Boolean Algebra and Logic Simplification
- Canonical Forms and Standard Forms
- Logic Diagrams
- Verilog
  - Synthesis
  - Simulation

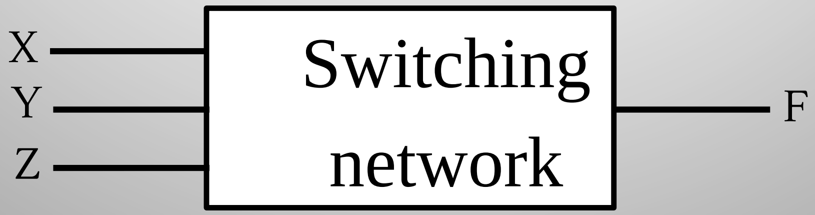
Basic Operations (1/4)

- Binary logic consists of binary variables and a set of logical operations. The variables are designated by letters of the alphabet, such as A, B, C, x, y, z, etc, with each variable having two and only two distinct possible values: 1 and 0, There are three basic logical operations: AND, OR, and NOT.

Boolean Variables: X, Y, Z … takes “0”, “1”

“L”, “H”

“F”, “T”

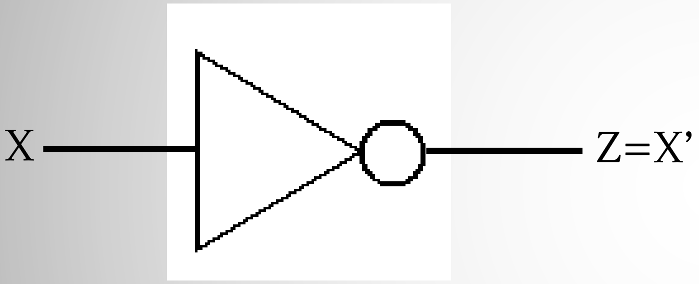


Basic Operations (2/4)

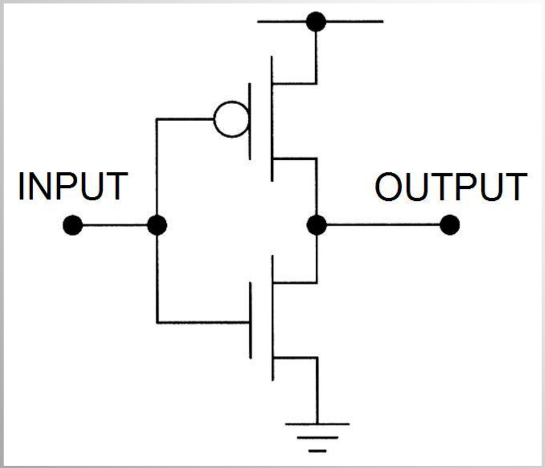
Complement (NOT or Inverter):

$0 \leftrightarrow 1 \quad x \leftrightarrow x' \; (\text{or } x \leftrightarrow \bar{x})$

Truth Table

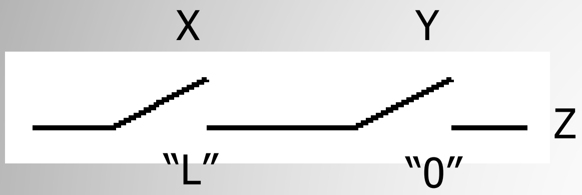


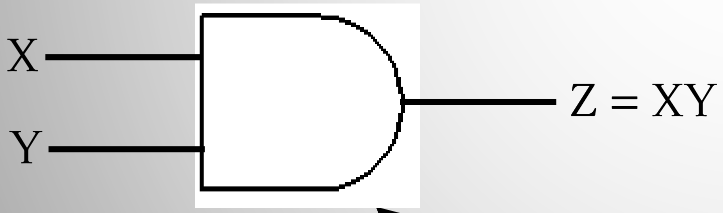
X	Z= X' or $(\overline{X})$
0	1
1	0



Basic Operations (3/4)

$$\boxed{AND \quad 0 \cdot 0 = 0 \quad 0 \cdot 1 = 0 \quad 1 \cdot 0 = 0 \quad 1 \cdot 1 = 1}$$





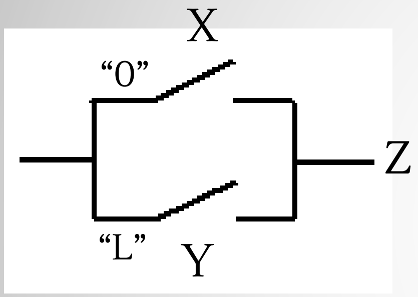
AND Gate

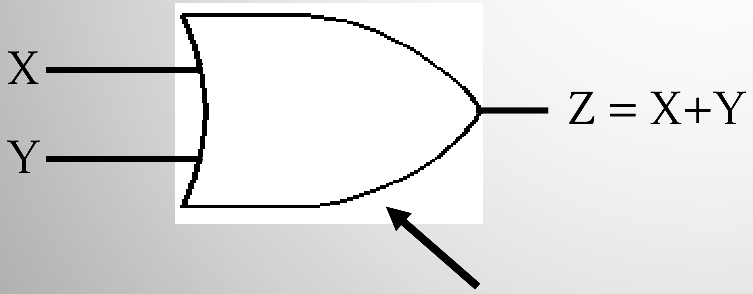
Truth Table

X	Y	$Z = X \cdot Y$
0	0	0
0	1	0
1	0	0
1	1	1

Basic Operations (4/4)

$$\boxed{OR \quad 0+0=0 \quad 0+1=1 \quad 1+0=1 \quad 1+1=1}$$





OR gate

Truth Table

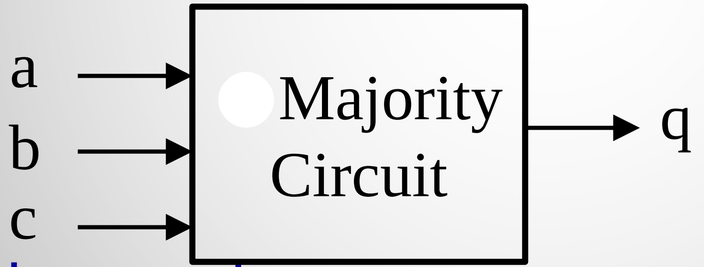
X	Y	$Z = X + Y$
0	0	0
0	1	1
1	0	1
1	1	1

Ways to Describe A Logic Function

- Example – 3-Input Majority Circuit

1) English language description (specification)
   - Outputs 1 if more inputs are 1 than are 0

1) Block Diagram



3) Truth table

a b c	q
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

4) Logic equation (Boolean equation)
   - q = (a · b) **+** (a · c) **+** (b · c)

Notes on Notation

- For AND logic function:
  - Alternative symbol: •or ∧ or &
- For OR logic function:
  - Alternative symbol: +or ∨ or |
- For NOT logic function:
  - Alternative symbol: ¬ or ’ or $\bar{a}$ or ~ or !

Truth Table

- A *truth table* of a Boolean function lists the output values for every possible input value combination in tabular form.

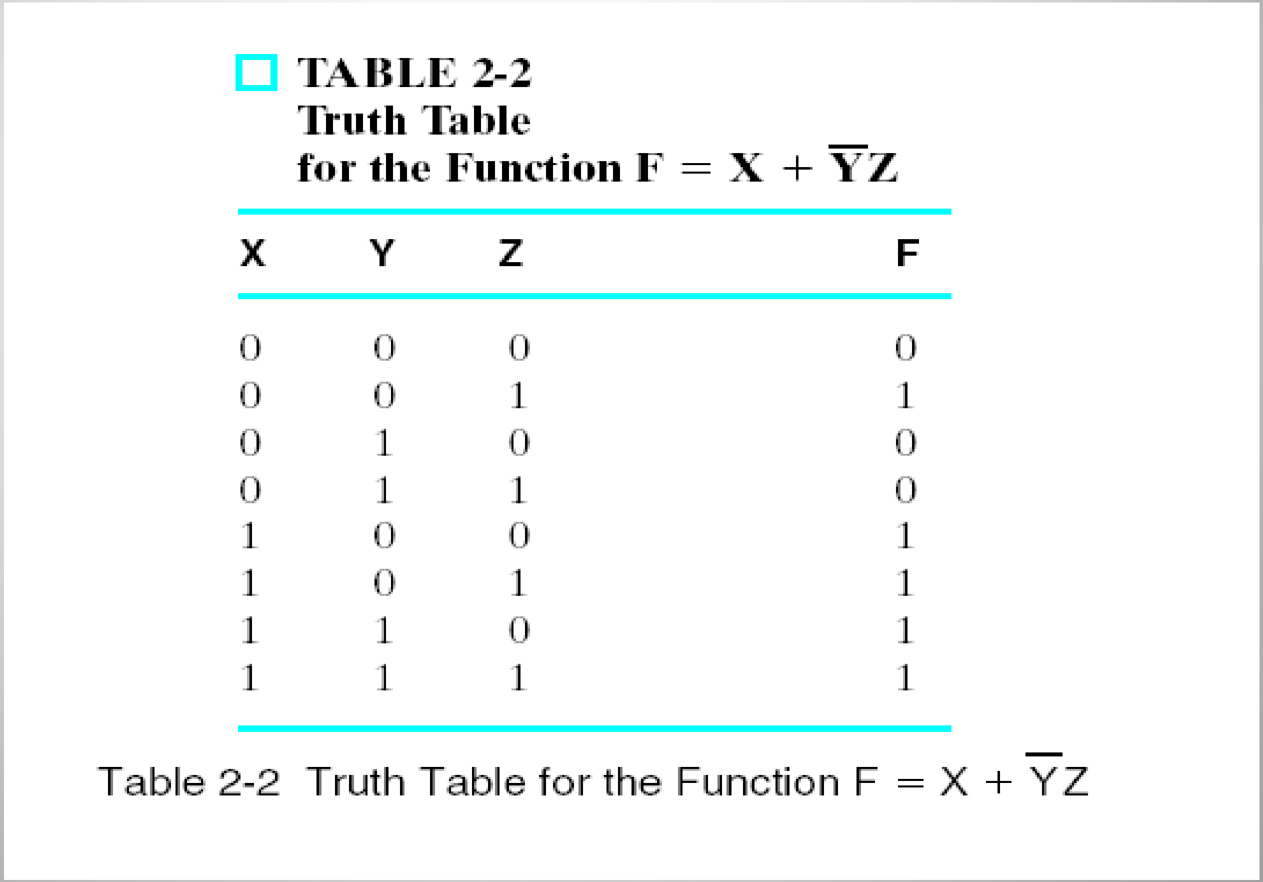


TABLE 2-2
Truth Table
for the Function $F = X + \overline{Y}Z$

X	Y	Z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Table 2-2 Truth Table for the Function $F = X + \overline{Y}Z$

Boolean Functions

- Examples
  - $F_1 = x\,y\,z'$
  - $F_2 = x + y'z$
  - $F_3 = x'\,y'\,z + x'\,y\,z + x\,y'$
  - $F_4 = x\,y' + x'\,z$

Boolean Functions

- Truth table of 2^n entries for *n* variables

x	y	z	F_1	F_2	F_3	F_4
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	1	0	0	0	0	0
0	1	1	0	0	1	1
1	0	0	0	1	1	1
1	0	1	0	1	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	0

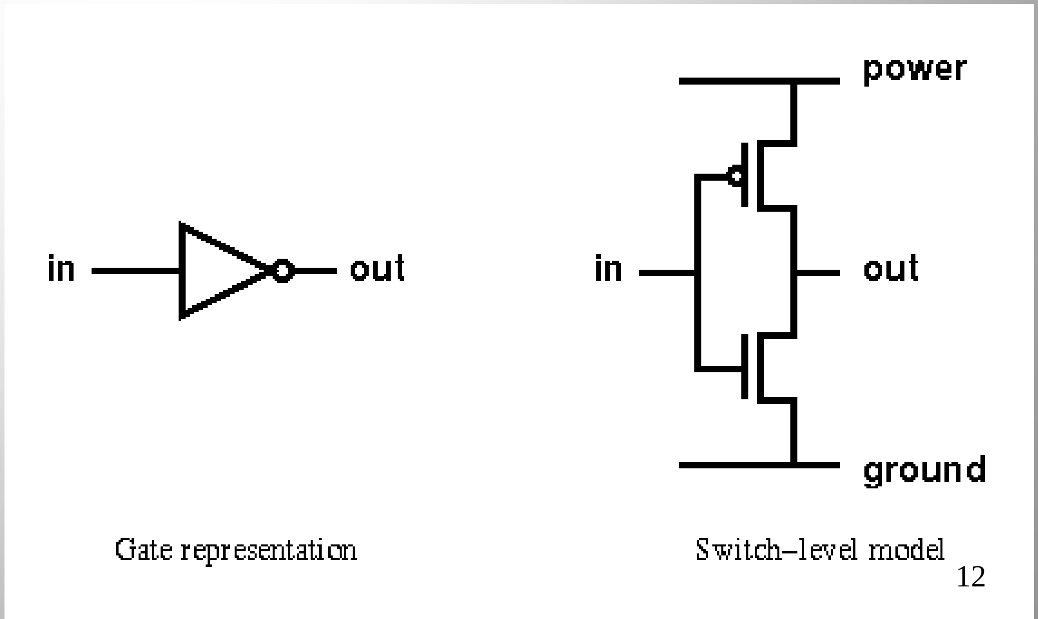
- Examples
  - $F_1 = x\,y\,z'$
  - $F_2 = x + y'z$
  - $F_3 = x'\,y'\,z + x'\,y\,z + x\,y'$
  - $F_4 = x\,y' + x'\,z$

- Two Boolean expressions may specify the same function
  - *e.g.* $F_3 = F_4$

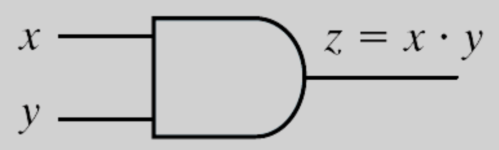
AND, OR, NOT Gates

- *Logic gates* are basic logic elements of a circuit
- They implement Boolean logic (use voltage to represent logic value)
- NOT gate, AND gate, OR gate implement the three basic logic operations

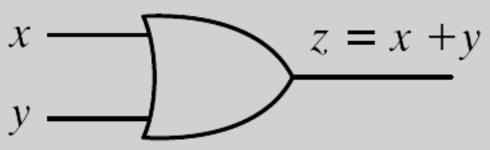
NOT gate:



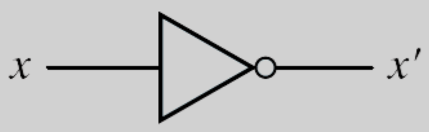
Graphic Symbols for Logic Gates



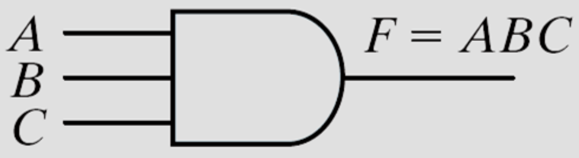
(a) Two-input AND gate



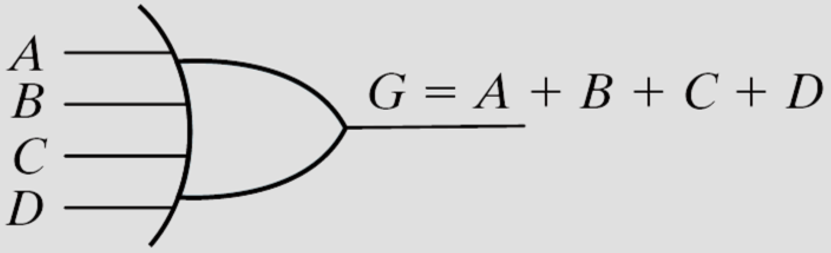
(b) Two-input OR gate



(c) NOT gate or inverter

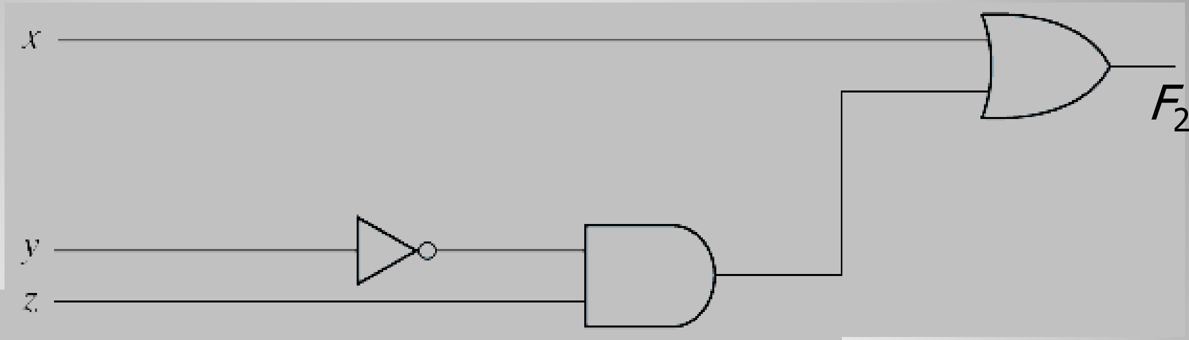


(a) Three-input AND gate

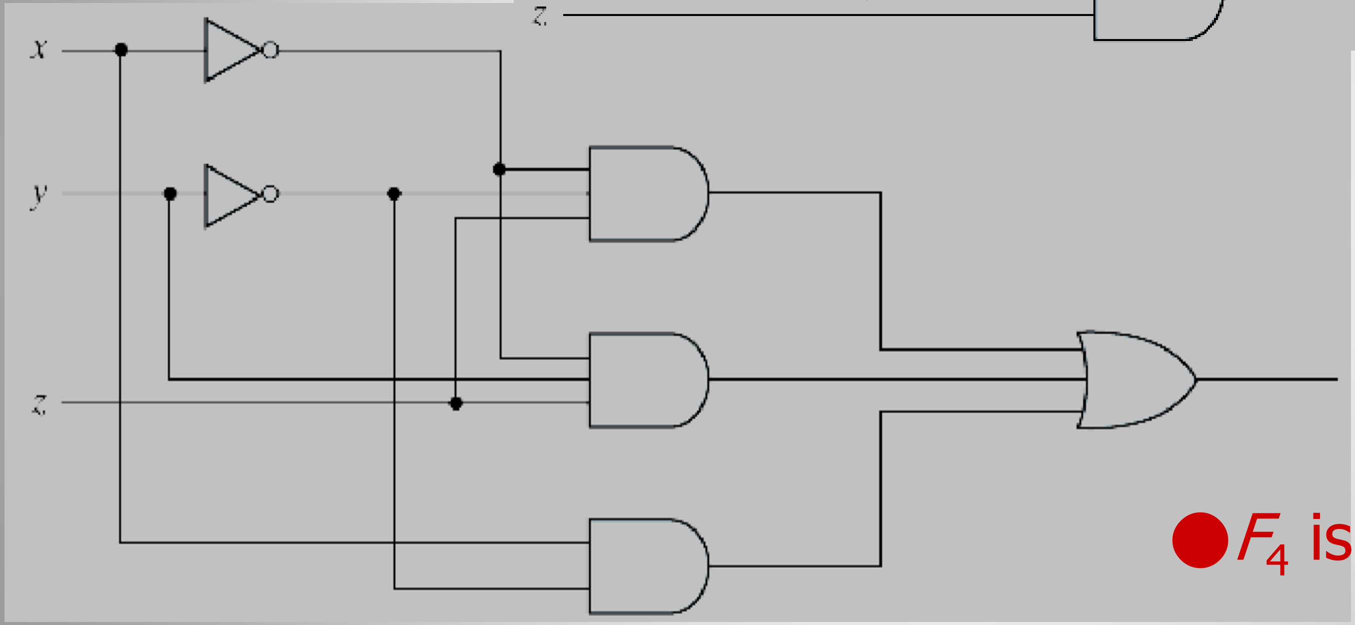


(b) Four-input OR gate

Implementation with Logic gates

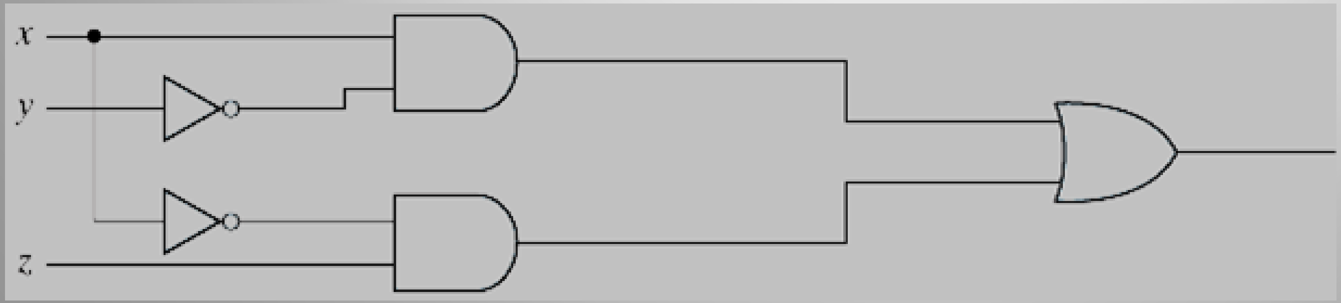


$F_2 = x + y'z$



$F_3 = x'y'z + x'yz + xy'$

- F_4 is more economical



$F_4 = xy' + x'z$

How to know if two circuits are equivalent?

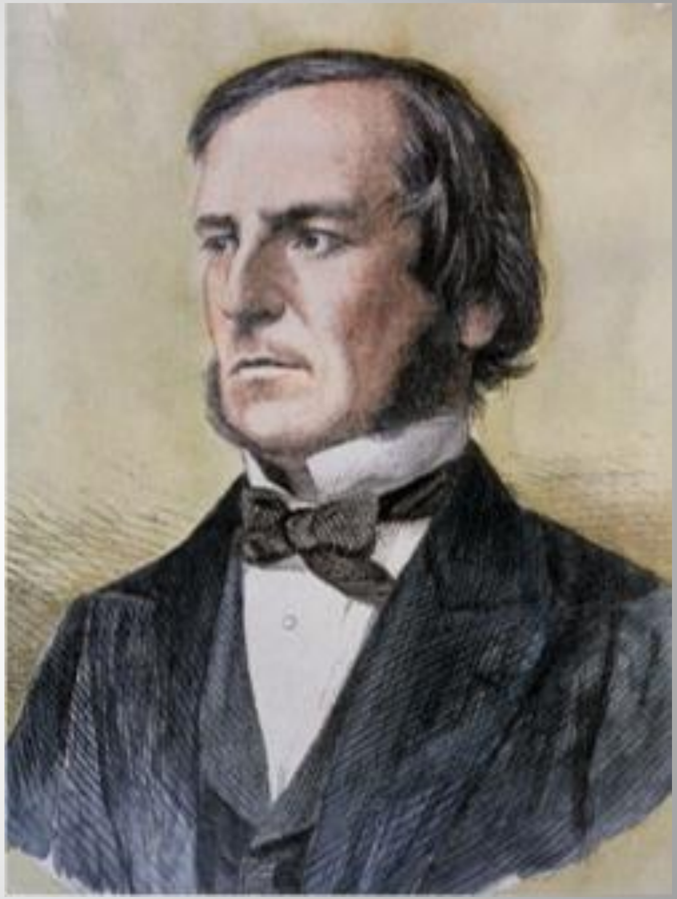
How to get a more economical implementation?

Boolean Algebra

- Boolean algebra has applications in set theory, mathematical logic, digital system design.
- Algebra over
  - two elements: 0 and 1
  - two binary operations: + and·
  - a complement operation: ′
- The operations are defined as follows:
  - $0' = 1$ and $1' = 0$
  - $0 \cdot 0 = 0$, $0 \cdot 1 = 0$, $1 \cdot 0 = 0$, $1 \cdot 1 = 1$
  - $0+0 = 0$, $0+1 = 1$, $1+0 = 1$, $1+1 = 1$

Boolean Logic

- George Boole (1815-1864)
  - An English mathematician, philosopher and logician
  - Mathematical Analysis of Logic (1847)
  - An Investigation of the Law of Thought (1854)
  - The inventor of Boolean logic, the basis of the modern digital computer


[Source: Wikipedia]

Operator Priority

- From high to low
  - Parentheses
  - NOT
  - AND
  - OR
- Examples
  - $x \cdot y' + z$
  - $(x \cdot y)' + z$

Axiomatic Approach of Boolean Algebra

- *Axiom*: a set of mathematical statement asserted to be true
- If we start with the following axioms (virtually stating the definitions of AND, OR, NOT with a set of math statements), then everything else in Boolean algebra can be derived from them
- Identity:
  - $1 \cdot x = x$
  - $0 + x = x$
- Annihilation:
  - $1 + x = 1$
  - $0 \cdot x = 0$
- Negation:
  - $0' = 1$
  - $1' = 0$

Theorems

- *Theorems* are logical consequences of the axioms
- Some useful theorems for logic simplification:

Commutative	$x \cdot y = y \cdot x$	$x + y = y + x$
Associative	$x \cdot (y \cdot z) = (x \cdot y) \cdot z$	$x + (y + z) = (x + y) + z$
Distributive	$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$	$x + (y \cdot z) = (x + y) \wedge (x \cdot z)$
Idempotence	$x \cdot x = x$	$x + x = x$
Complementation	$x \cdot x' = 0$	$x + x' = 1$
Absorption	$x \cdot (x + y) = x$	$x + (x \cdot y) = x$
Combining	$(x \cdot y) + (x \cdot y') = x$	$(x + y) \cdot (x + y') = x$

Theorems

DeMorgan's	$(x \cdot y)' = x' + y'$	$(x + y)' = x' \cdot y'$
Involution	$(x')' = x$	
Consensus	$(x \cdot y) + (x' \cdot z) + (y \cdot z)$ $= (x \cdot y) + (x' \cdot z)$	$(x + y) \cdot (x' + z) \cdot (y + z)$ $= (x + y) \cdot (x' + z)$

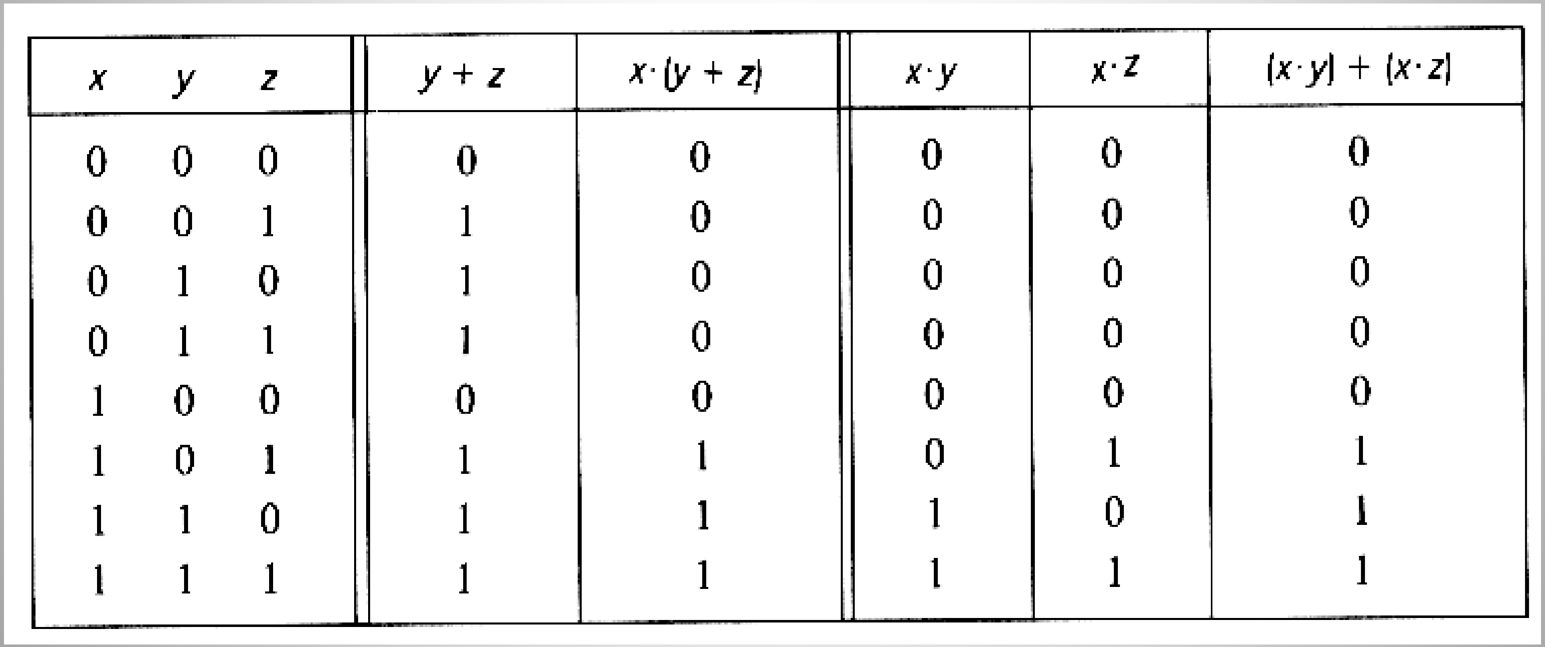
Example

- Prove $x \cdot y = y \cdot x$.   (commutative)

Example

- Prove $x \cdot (y + z) = x \cdot y + x \cdot z$. (distributive)

Proof. We can prove it by perfect induction.



x	y	z	$y + z$	$x \cdot (y + z)$	$x \cdot y$	$x \cdot z$	$(x \cdot y) + (x \cdot z)$
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

It can be seen that the statement is true for all possible values of *x*, *y*, and *z*, hence the statement is proved.

Duality

1. $X + 0 = X$	1D. $X \cdot 1 = X$
2. $X + 1 = 1$	2D. $X \cdot 0 = 0$
3. $X + X = X$	3D. $X \cdot X = X$
5. $X + X' = 1$	5D. $X \cdot X' = 0$
6. $X + Y = Y + X$	6D. $XY = YX$
7. $(X + Y) + Z = X + (Y + Z)$	7D. $(XY)Z = X(YZ)$
8. $X(Y + Z) = XY + XZ$	8D. $X + YZ = (X + Y)(X + Z)$
12. $(X + Y)' = X'Y'$	12D. $(XY)' = X' + Y'$

- *Duality principle*: the dual of any true statement is also a true statement
- The dual is obtained by interchanging OR and AND operations, interchanging 0 and 1
- Duality holds in the axioms, and all of the Boolean algebra derived from them

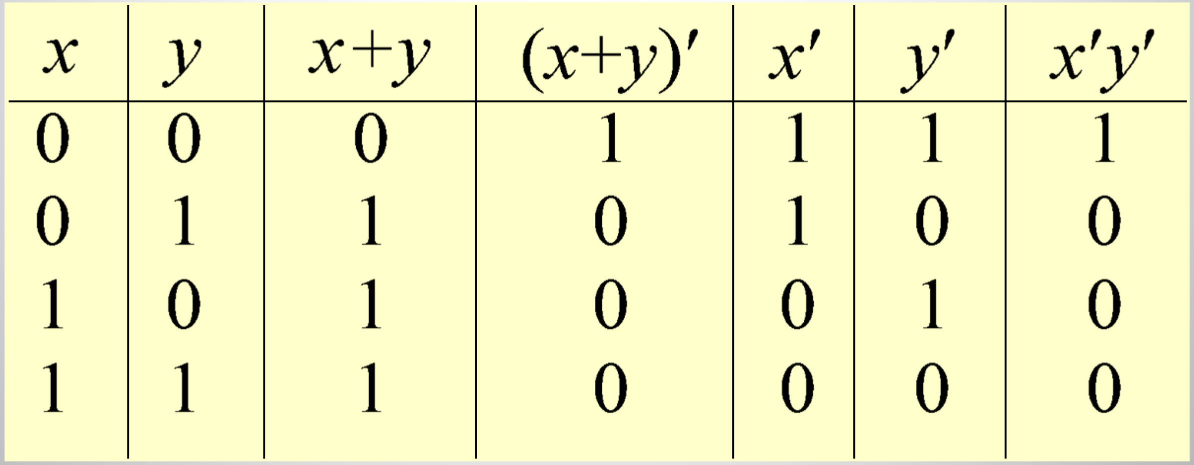
Remarks

- In general, Boolean properties can be proved
  1) By using axioms or known theorems/Boolean properties
  2) By checking their validity for all possible combinations of variables
     - Simply with truth table
     - Mathematicians call this proof technique perfect induction

Example

- Prove the *DeMorgan's Theorems*
  - $(x + y)' = x' y'$
  - $(x y)' = x' + y'$

Proof. We can prove them by perfect induction.

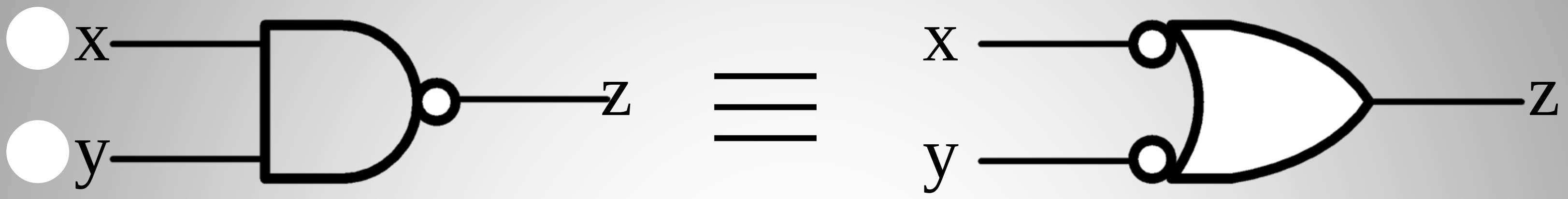


x	y	$x+y$	$(x+y)'$	x'	y'	$x'y'$
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

It can be seen that $(x+y)' = x' y'$ is true for all possible values of *x* and y, hence the statement is proved.

DeMorgan Graphically

- NAND: $z = (x\,y)' = x' + y'$



- NOR: $z = (x + y)' = x'\,y'$

Example: Simplify Logic Expression using Boolean Properties

$f(a,b,c) = (a \land c) \lor (a \land b \land c) \lor (\neg a \land b \land c) \lor (a \land b \land \neg c)$

$= \underline{(a \land c) \lor (a \land b \land c)} \lor$
$(\neg a \land b \land c) \lor (a \land b \land c) \lor$
$(a \land b \land \neg c) \lor (a \land b \land c)$

By Idempotence ($x = x \lor x$)

$= (a \land c) \lor$
$(\neg a \land b \land c) \lor (a \land b \land c) \lor$
$\underline{\overline{(a \land b \land \neg c) \lor (a \land b \land c)}}$

By Absorption Property
$x \lor (x \land y) = x$

$= (a \land c) \lor (b \land c) \lor (a \land b)$

By Combining Property

Majority function

Example: Simplify Logic Expression using Boolean Properties

f(a,b,c) = ac **+** abc **+** a’ bc + abc’

= ac + (abc+a’bc) + (abc +abc’) *By Idempotence (x = x ∨ x)*

= ac+bc+ab *By Combining Property*

Majority function

Example

- Simplify the following Boolean expression using known Boolean properties
- $f(x, y) = (x \wedge (y \vee \neg x)) \vee \neg(\neg x \wedge \neg y)$

$$= x(y+x')+(x'y')'$$

$$= xy+ x+y$$

$$= x+y$$

How to implement arbitrary functions using AND, OR, NOT gate?

Minterms and Maxterms

- *Minterms*
  - minterm: an AND term consisting of all literals in their normal form or in their complement form
  - For two binary vars *x* and *y*, the minterms are
    - *xy, xy', x'y, x'y'*
  - *n* variables -> 2^n minterms
  - Also called *standard products*
- *Maxterms*
  - maxterm: an OR term consisting of all literals in their normal form or in their complement form
  - Also called *standard sums*

Minterms and Maxterms

Maxterm M_j is the complement of minterm m_j, and vice versa

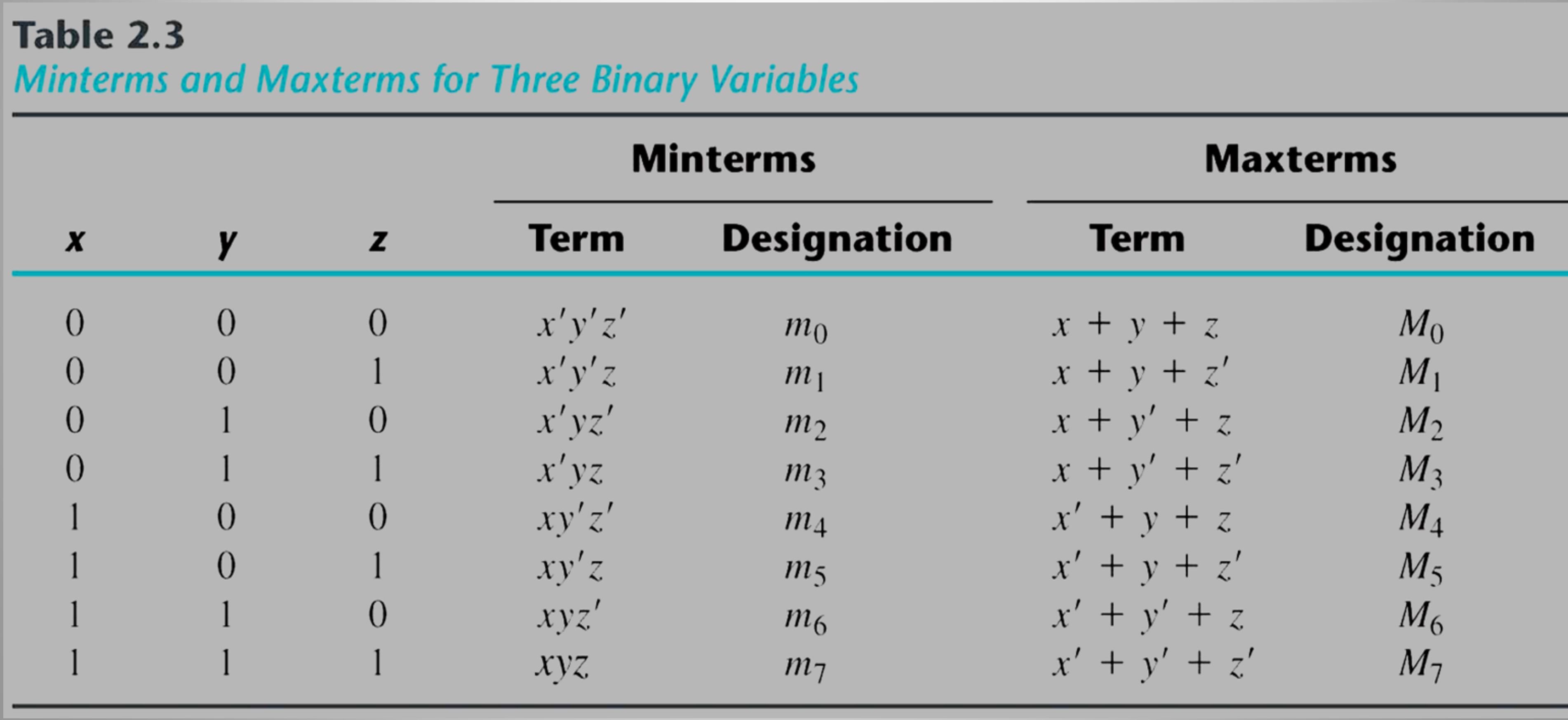


Table 2.3
Minterms and Maxterms for Three Binary Variables

			Minterms		Maxterms	
x	***y***	***z***	**Term**	**Designation**	**Term**	**Designation**
0	0	0	$x'y'z'$	m_0	$x + y + z$	M_0
0	0	1	$x'y'z$	m_1	$x + y + z'$	M_1
0	1	0	$x'yz'$	m_2	$x + y' + z$	M_2
0	1	1	$x'yz$	m_3	$x + y' + z'$	M_3
1	0	0	$xy'z'$	m_4	$x' + y + z$	M_4
1	0	1	$xy'z$	m_5	$x' + y + z'$	M_5
1	1	0	xyz'	m_6	$x' + y' + z$	M_6
1	1	1	xyz	m_7	$x' + y' + z'$	M_7

Sum of Minterms

- A Boolean function can be expressed by
  - a truth table, or
  - sum of minterms that produces output 1
- e.g. $f_1 = x'y'z + xy'z' + xyz = m_1 + m_4 + m_7$

  $f_2 = x'yz + xy'z + xyz' + xyz = m_3 + m_5 + m_6 + m_7$

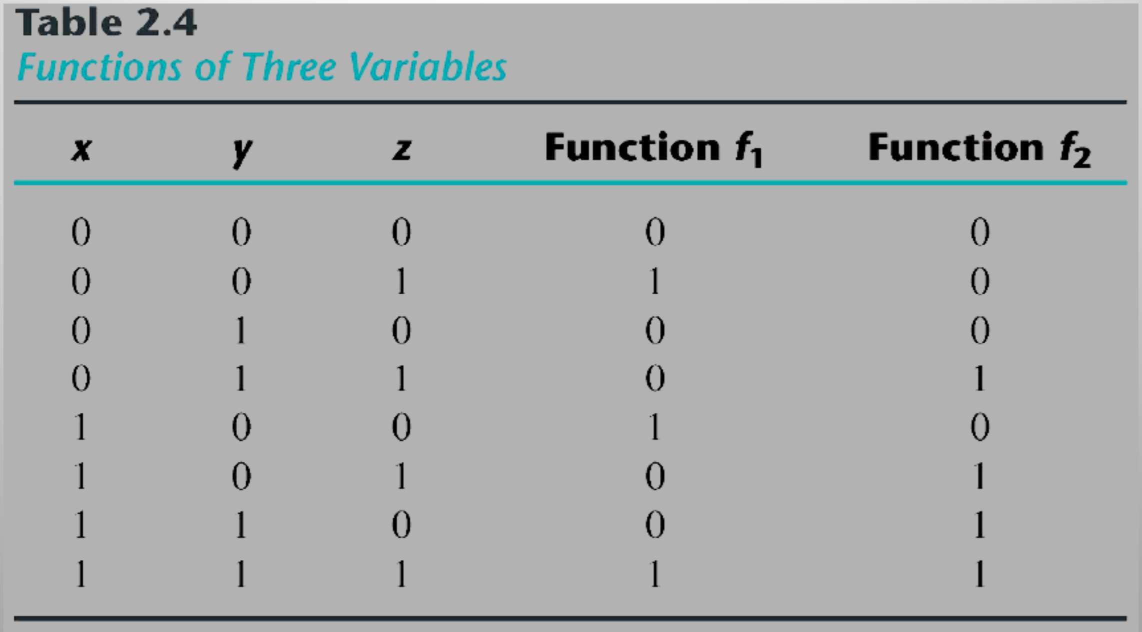


Table 2.4
Functions of Three Variables

x	y	z	Function f_1	Function f_2
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Product of Maxterms

- The complement of a function can be expressed by sum of minterms that produce output 0 for the original function
- e.g. If $f = m_1 + m_4 + m_7$, then

$$f' = m_0 + m_2 + m_3 + m_5 + m_6$$

- Notice that

$f = (f')'$

$= m'_0 m'_2 m'_3 m'_5 m'_6$

$= M_0 M_2 M_3 M_5 M_6$

- Any Boolean function can be expressed as
  - a sum of minterms, or a product of maxterms
  - both forms are *Canonical forms*

Conversion between Canonical Forms

- Example
  - $F(A,B,C) = \sum m(1,4,5,6,7)$

    $\Rightarrow F'(A,B,C) = \sum m(0,2,3)$

    $\Rightarrow$ By DeMorgan's theorem

    $F(A,B,C) = \prod M(0,2,3)$
- Interchange the symbols $\sum$ and $\prod$ and list those numbers missing from the original form

Example

- $F = xy + x'z$
- $F(x, y, z) = \sum m\,(1, 3, 6, 7)$
- $F(x, y, z) = \prod M\,(0, 2, 4, 5)$

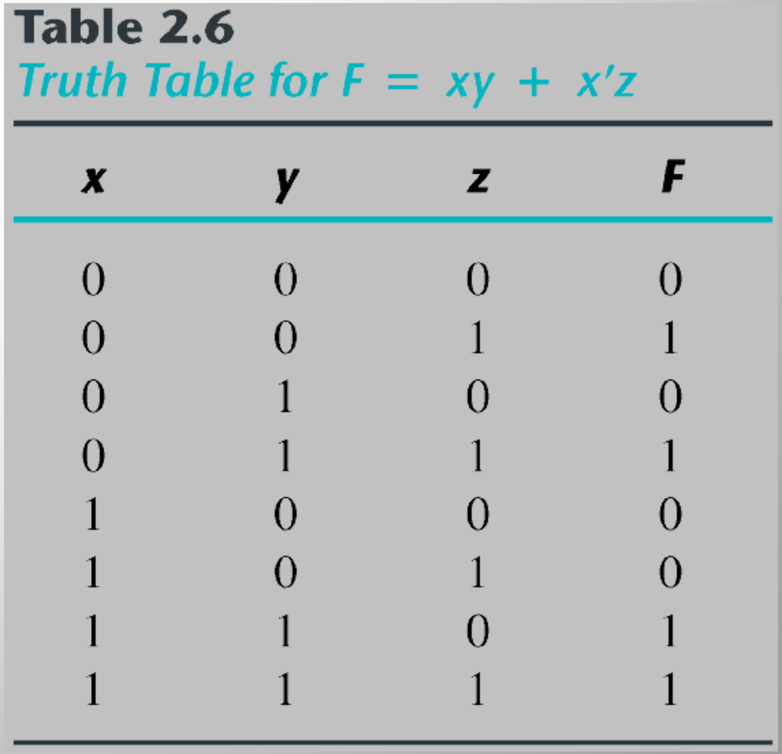


Table 2.6
Truth Table for $F = xy + x'z$

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

Standard Forms

- Two kinds of *Standard forms*
  - Sum of products (SOP) form
    - e.g. f = xy + x'y + y'z
  - Product of sums (POS) form
    - e.g. g = x(y'+z)(x'+y+z'+w)
  - SOP/POS form of a function is not unique e.g. f can also be expressed as y + y'z, or y + z, etc.

Sum-of-Product & Product-of-Sum Forms

- Sum-of-product (SOP) form
  - AB + A'BC' + B'D + C
  - AB'C + A'C
- Product-of-sum (POS) form
  - (A + B)(A' + D + E)
  - (A + C)(B + D)(B' + C' + D')CE'

Transforming to Canonical Form

- Any logic expression can be transformed to Canonical form by factoring about each input variable using the identity:

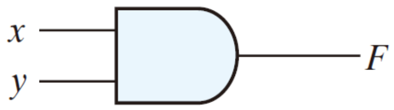
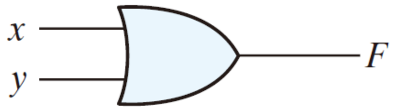
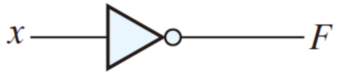
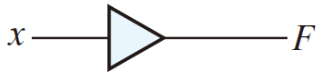
$$f(x_1,\ldots,x_i,\ldots,x_n)$$
$$= (x_i \land f(x_1,\ldots,1,\ldots,x_n)) \lor (\overline{x_i} \land f(x_1,\ldots,0,\ldots,x_n))$$

- E.g., f(a,b,c) = (a ∧ c) ∨ (b ∧ c) ∨ (a ∧ b)

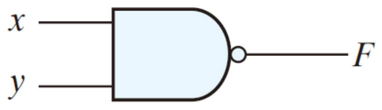
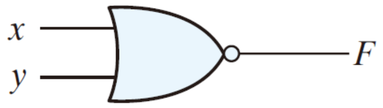
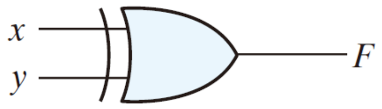
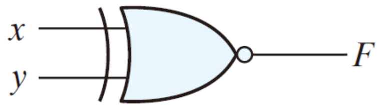
  f(a, b, c) = (a ∧ f(1, b, c)) ∨ (a' ∧ f(0, b, c))
  = (a ∧ (c ∨ (b ∧ c) ∨ b)) ∨ (a' ∧ (b ∧ c))
  = (a ∧ c) ∨ (a ∧ b ∧ c) ∨ (a ∧ b) ∨ (a' ∧ b ∧ c)

- Repeat with b and c (exercise)

Digital Logic Gates (1/2)

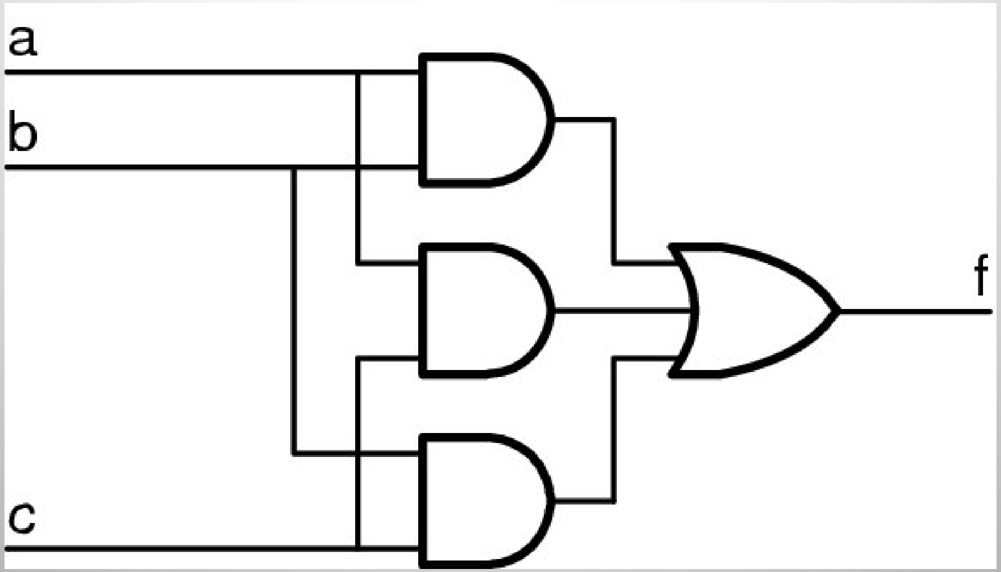
Name	Graphic symbol	Algebraic function	Truth table
AND		$F = x \cdot y$	x y | F 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR		$F = x + y$	x y | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
Inverter		$F = x'$	x | F 0 | 1 1 | 0
Buffer		$F = x$	x | F 0 | 0 1 | 1

Digital Logic Gates (2/2)

			x	y	F
NAND		$F = (xy)'$	0	0	1
			0	1	1
			1	0	1
			1	1	0
			x	y	F
NOR		$F = (x + y)'$	0	0	1
			0	1	0
			1	0	0
			1	1	0
			x	y	F
Exclusive-OR (XOR)		$F = xy' + x'y$ $= x \oplus y$	0	0	0
			0	1	1
			1	0	1
			1	1	0
			x	y	F
Exclusive-NOR or equivalence		$F = xy + x'y'$ $= (x \oplus y)'$	0	0	1
			0	1	0
			1	0	0
			1	1	1

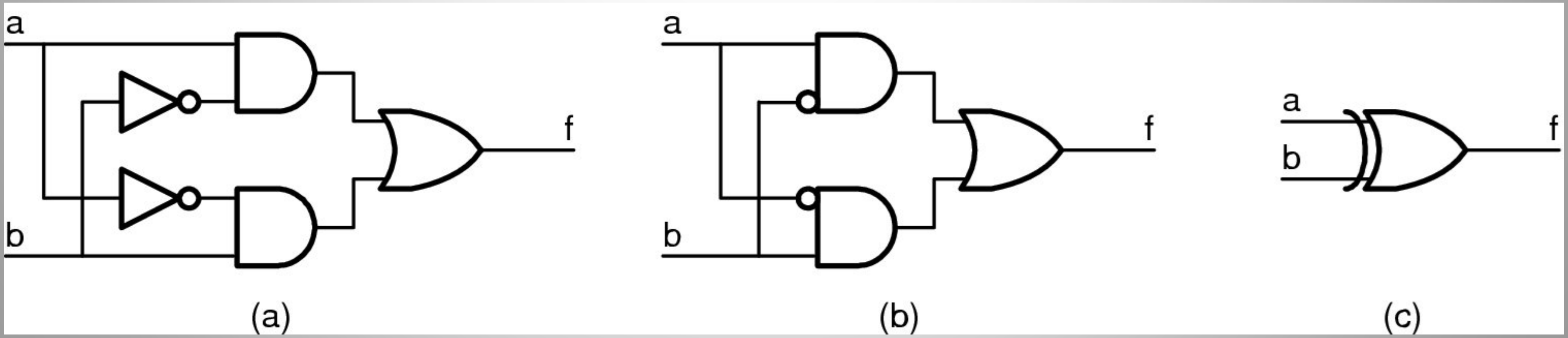
From Equations to Gates

- Logical functions can be represented by a (gate-level) logic diagram
  - A (gate-level) schematic drawing of gate symbols connected by lines
- E.g., $f = a \cdot c + b \cdot c + a \cdot b$

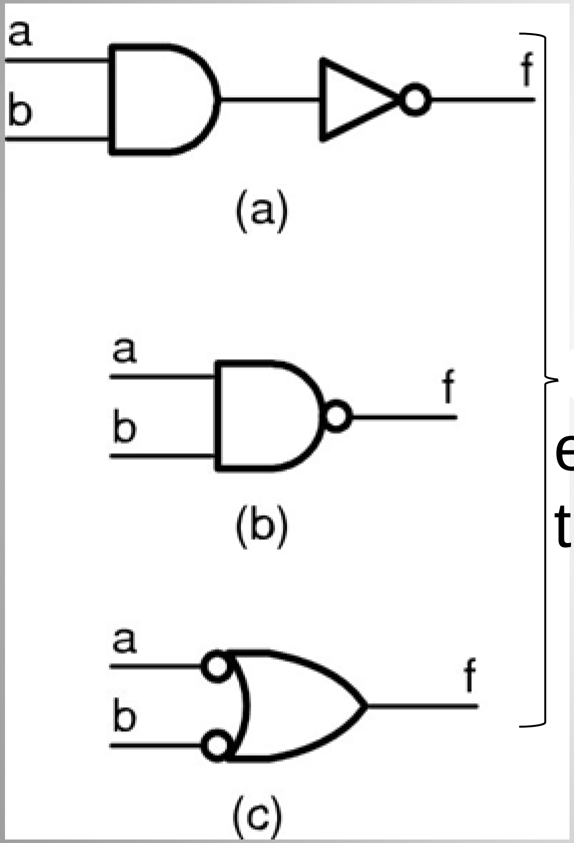


From Equations to Gates (cont)

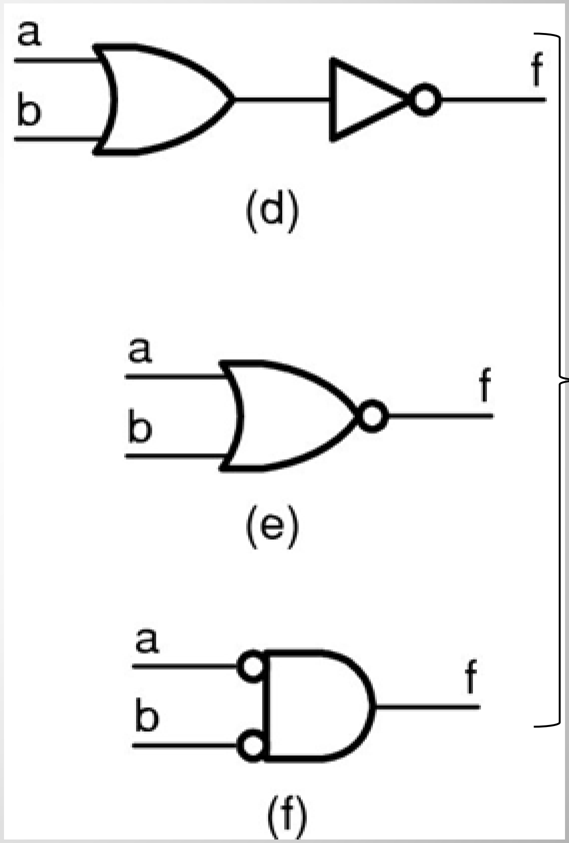
- Using AND/OR/INV gates, we can draw logic diagram (schematic) for any Boolean expression
- E.g., exclusive-or (XOR) function
  $f = a \cdot b' + a' \cdot b$



Moving Inversion Bubbles (Review of DeMorgan's Law)

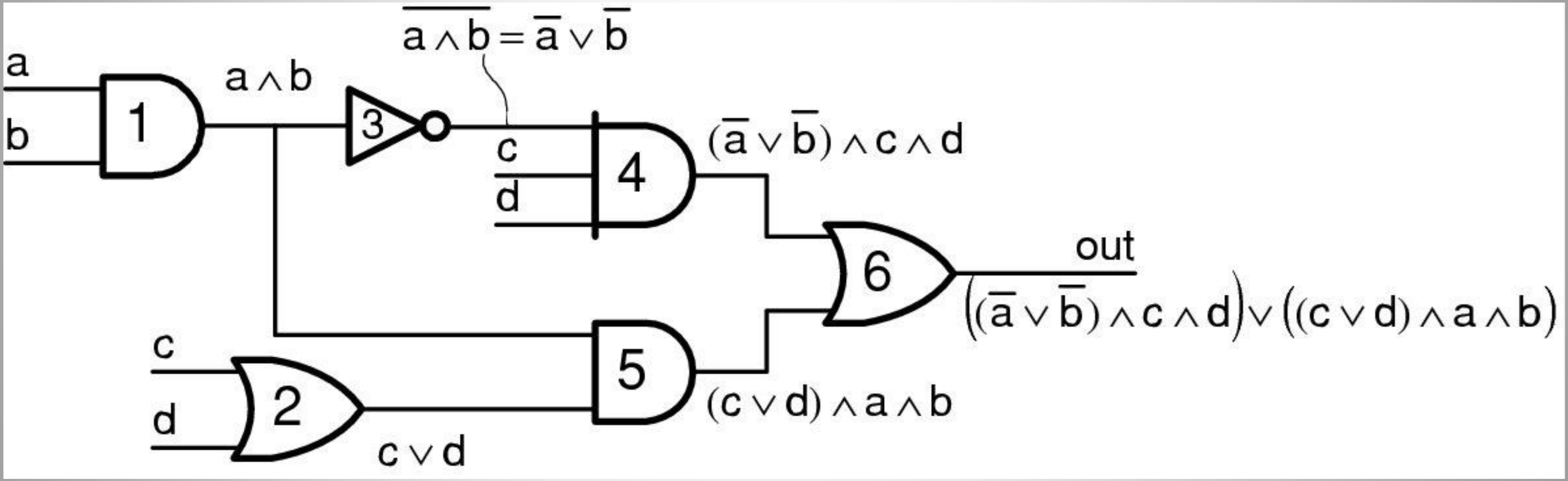


- all are equivalent to NAND



- all are equivalent to NOR

Converting from Logic Diagram to Equation

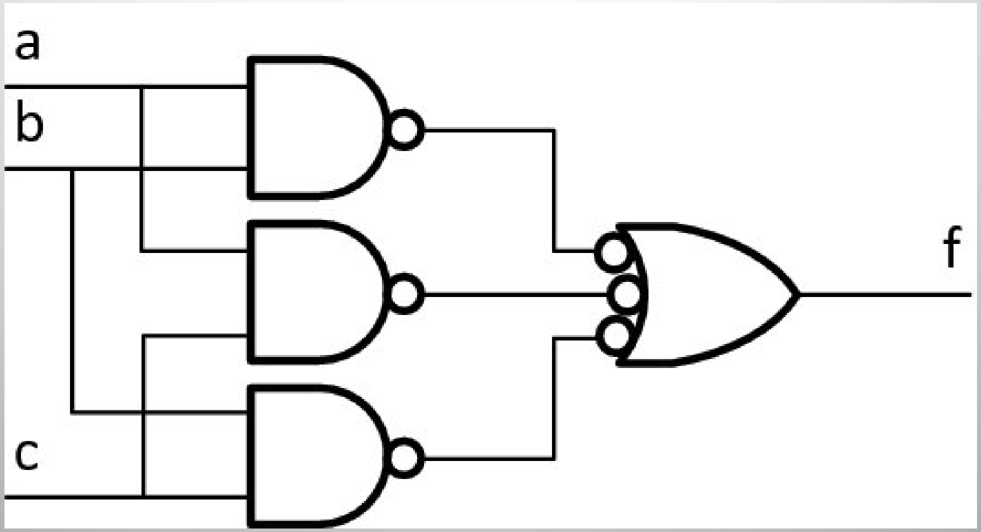


$\overline{a \wedge b} = \bar{a} \vee \bar{b}$
a
b
1
$a \wedge b$
3
c
d
4
$(\bar{a} \vee \bar{b}) \wedge c \wedge d$
6
out
$\left((\bar{a} \vee \bar{b}) \wedge c \wedge d\right) \vee \left((c \vee d) \wedge a \wedge b\right)$
c
d
2
5
$(c \vee d) \wedge a \wedge b$
$c \vee d$

Example: From Equation to Schematic

- Draw the schematic for a three-input majority function with NAND gates only

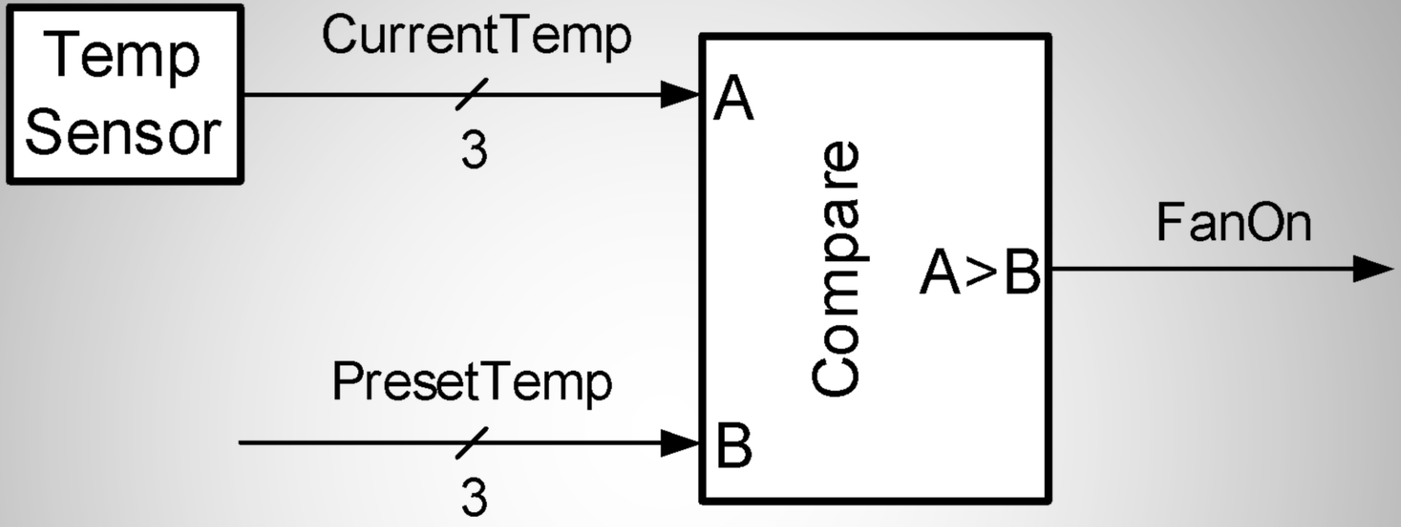
$$
\begin{aligned}
f(a,b,c) &= (a \wedge b) \vee (a \wedge c) \vee (b \wedge c) \\
&= \overline{\overline{(a \wedge b) \vee (a \wedge c) \vee (b \wedge c)}} \\
&= \overline{\overline{(a \wedge b)} \wedge \overline{(a \wedge c)} \wedge \overline{(b \wedge c)}}
\end{aligned}
$$



Basic Concepts of Verilog

- A *Hardware Description Language* *(HDL)*
- Used as design description for
  - Synthesis – implement hardware with a network of logic gates/cells in ASIC or FPGA
  - Simulation – see what your hardware will do before you build it
- Basic unit is a *module*
- Modules have
  - Module declaration
  - Input and output declarations
  - Internal signal declarations
  - Logic definition
    - `assign` statements
    - `case` statements
    - Submodule instantiations

Example: Verilog for Thermostat



```
module Thermostat(presetTemp, currentTemp, fanOn) ;
  input [2:0] presetTemp, currentTemp ; // temp inputs,3 bits
   each
  output fanOn ;                        // true when current > preset

  wire fanOn ;
  assign fanOn = (currentTemp > presetTemp) ;
endmodule
```

Example: Verilog for Thermostat

```
module Thermostat(presetTemp, currentTemp, fanOn) ;
  input [2:0] presetTemp, currentTemp ;    // temperature inputs, 3 bits each
  output fanOn ; // true when current > preset


  wire fanOn ;
  assign fanOn = (currentTemp > presetTemp) ;
endmodule
```

Example: Verilog for Thermostat

- Module declaration
- I/O list

```
module Thermostat(presetTemp, currentTemp, fanOn) ;
  input [2:0] presetTemp, currentTemp ;    // temperature inputs, 3 bits each
  output fanOn ; // true when current > preset
```

- Declare I/Os
- 3-bit wide signals

```
  wire fanOn ;
  assign fanOn = (currentTemp > presetTemp) ;
endmodule
```

- An assign statement defines a signal with an equation
- A wire is a signal set with an assign statement or connected to a module

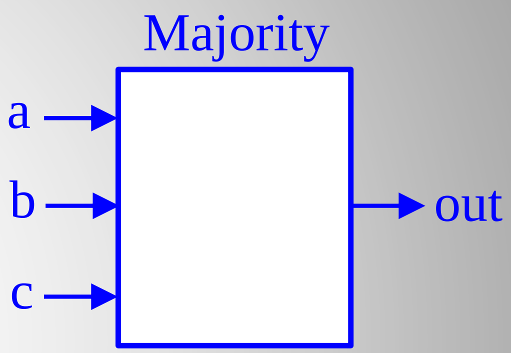
Verilog Description of Majority Function: Majority.v

Module name

I O

```
module Majority(a, b, c, out);
  input a, b, c;
  output out;
  wire out;
  assign out = (a & b) | (a & c) | (b & c);
endmodule
```

Block Diagram



Can be replaced with

```
wire out = (a & b) | (a & c) | (b & c);
```

Logic Function in Verilog

- Combinational logic in Verilog

Wire type (signal type)

```
wire out;
assign out = (a & b) | (a & c) | (b & c);
```

Signal IS NOT variable

$$f(a,b,c) = \overline{\overline{(a \wedge b)} \wedge \overline{(a \wedge c)} \wedge \overline{(b \wedge c)}}$$

- Alternatively,

```
assign out = ~(~(a & b) & ~(a & c) & ~(b & c));
```

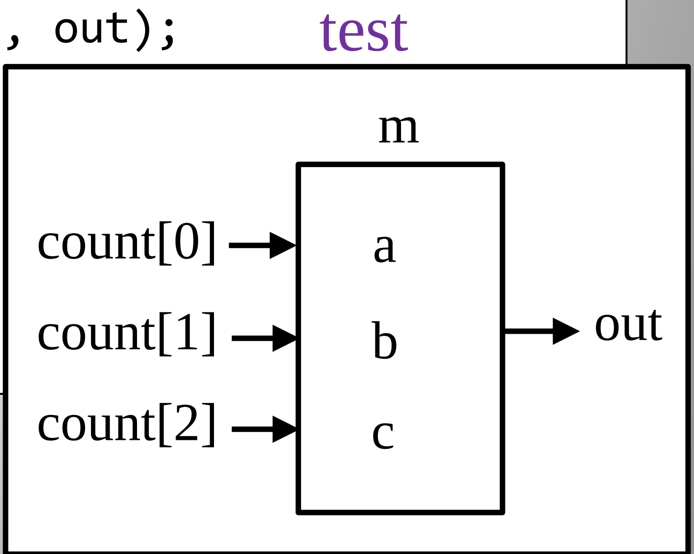
  - Less readable
  - Not necessary since synthesis tool performs optimization for you

Test Stimulus (Testbench): test.v

Register type (signal type), 3-bit vector

Delay

```
module test;
  reg [2:0] count;     // input - three bit counter
  wire out;            // output of majority
  // instantiate the block
  Majority m(count[0], count[1], count[2], out);
// generate all eight input patterns
  initial begin
    count = 3'b000;
    repeat (8) begin
      #100
      $display(“in = %b, out = %b”, count, out);
      count = count + 3'b001;
    end
  end
endmodule
```



Verilog Simulation Output

- There is 100-unit delay time between two outputs
- Simulation result

```
in = 000, out = 0
in = 001, out = 0
in = 010, out = 0
in = 011, out = 1
in = 100, out = 0
in = 101, out = 1
in = 110, out = 1
in = 111, out = 1
```

Example: Days in Month Function

```
module DaysInMonth(month, days) ;
  input [3:0] month ; // month of the year 1 = Jan, 12 = Dec
  output [4:0] days ; // number of days in month

  reg [4:0] days ;

  always @(month) begin // evaluate whenever month changes
    case(month)
      2: days = 5’d28 ;         // February has 28
      4,6,9,11: days = 5’d30 ; // 30 days have September ...
      default: days = 5’d31 ;  // all the rest have 31...
    endcase
  end
endmodule
```

Example: Days in Month Function

```
module DaysInMonth(month, days) ;
  input [3:0] m                        r 1 = Jan, 12 = Dec
  output [4:0]                         n month

  reg [4:0] days ;

  always @(month) begi                         changes
    case(month)
      2: days = 5'd28 ;
      4,6,9,11: days = 5'd30 ;
      default: days = 5'd31 ;  // all the rest have 31...
    endcase
  end
endmodule
```

- Reg defines a signal set in an *always* block. It does **NOT** define a register.
- Always block evaluates each time activity list changes. In this case each time *month changes*
- Case statement selects statement depending on value of argument. Like a truth table.
- Can have multiple values per statement
- Default covers values not listed.

Verilog design style – for Synthesizable Modules

1. Combinational modules use only
    1. Assign statements
    2. Case or Casex statements (with default)
    3. If statements – only if all signals have a default assignment
    4. Instantiations of other combinational modules
2. Sequential modules use only
    1. Combinational logic
    2. Explicitly declared registers (flip-flops)
3. Do not use
    1. Loops
    2. Always blocks except for case, casex, or if
4. Do use
    1. Signal concatenation, e.g., {a, b} = {c, d}
    2. Signal subranges, e.g., a[7:1] = b[6:0] ;
5. Logic is organized into small modules
    1. Leaf modules not more than 40 lines
    2. If it could be made two modules, it should be

Verilog design style – for Synthesizable Modules (cont)

6. Use lots of comments
   1. Comments themselves
   2. Meaningful signal names – tempHigh, not th
   3. Meaningful module names – DaysInMonth, not Mod3
7. Activation lists for case statements include **ALL** inputs (or use *)
8. Constants
   1. All constants are `defined if used more than once
   2. Width of all constants is specified 5’d31, not 31
9. Signals
   1. Buses (multi-bit signals) are numbered high to low
      - e.g., wire [31:0] bus
   2. All signals should be high-true (except primary inputs and outputs)
10. Visualize the logic your Verilog will generate.
    - If you can’t visualize it, the result will not be pretty

Summary

- Boolean algebra
  - Rules to manipulate logic equations
- Logic diagrams
  - Representation of logic equations
- Verilog
  - Describe hardware for simulation and synthesis