EECS151/251A Introduction to Digital Design and ICs

Lecture 5: Combinational Logic

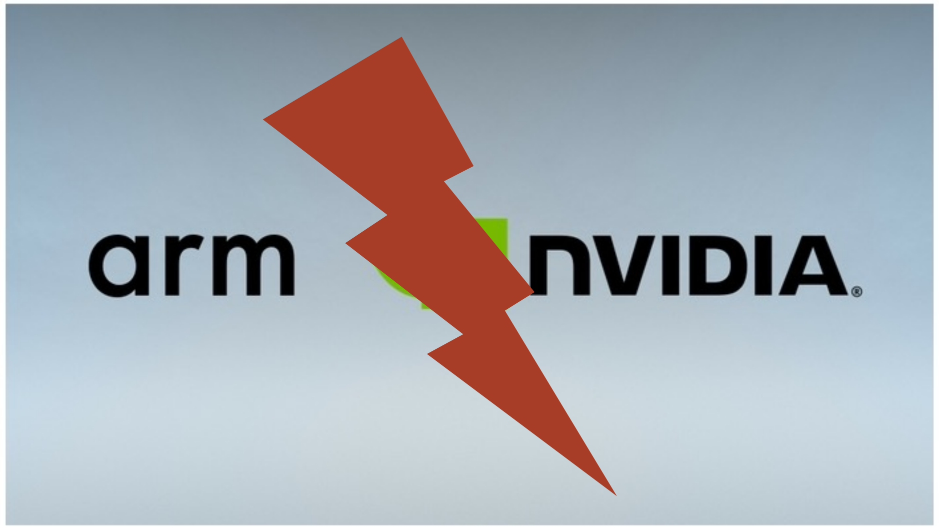
Sophia Shao

NVIDIA to Acquire Arm for $40 Billion (Fa20)

NVIDIA and SoftBank Group Announce Termination of NVIDIA's Acquisition of Arm (Fa22)

https://nvidianews.nvidia.com/news/nvidia-to-acquire-arm-for-40-billion-creating-worlds-premier-computing-company-for-the-age-of-ai

https://nvidianews.nvidia.com/news/nvidia-and-softbank-group-announce-termination-of-nvidias-acquisition-of-arm-limited



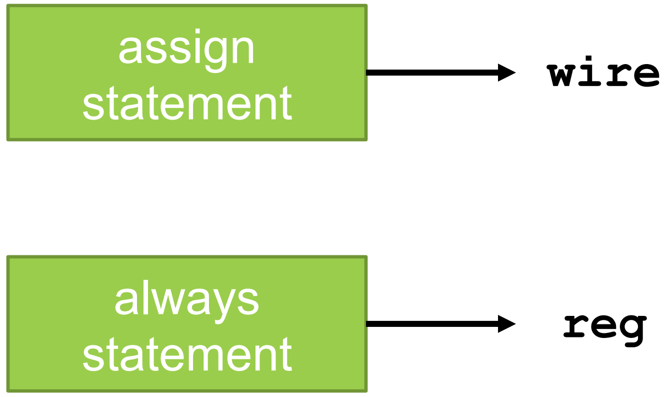
Simplified Verilog Guidelines

- Combinational logic:
  - Continuous Assignment:

```
assign a = b & c;
```

  - Always block with @(*)

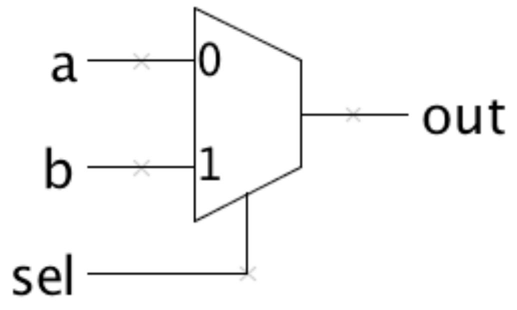
```
always @(*) begin
        a = b & c;  // blocking statement
end
```



The Sequential always Block

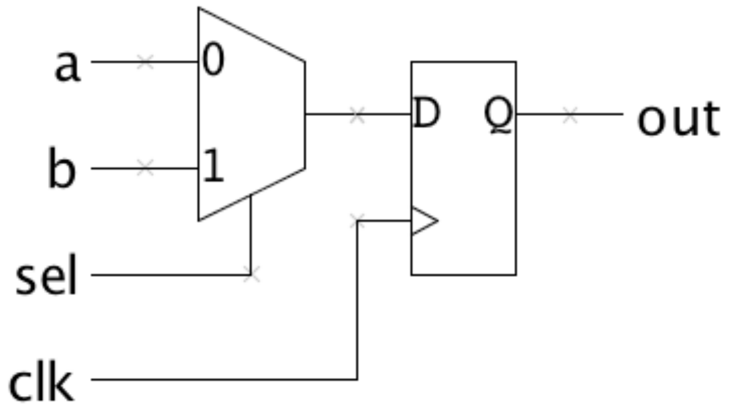
Combinational

```
module comb(input a, b, sel,
     output reg out);
  always @(*) begin
    if (sel) out = b;
    else out = a;
  end
endmodule
```



Sequential

```
module seq(input a, b, sel, clk,
                output reg out);
  always @(posedge clk) begin
    if (sel) out <= b;
    else out <= a;
  end
endmodule
```



Always Blocks

Always blocks give us some constructs that are impossible or awkward in continuous assignments.

case statement example:

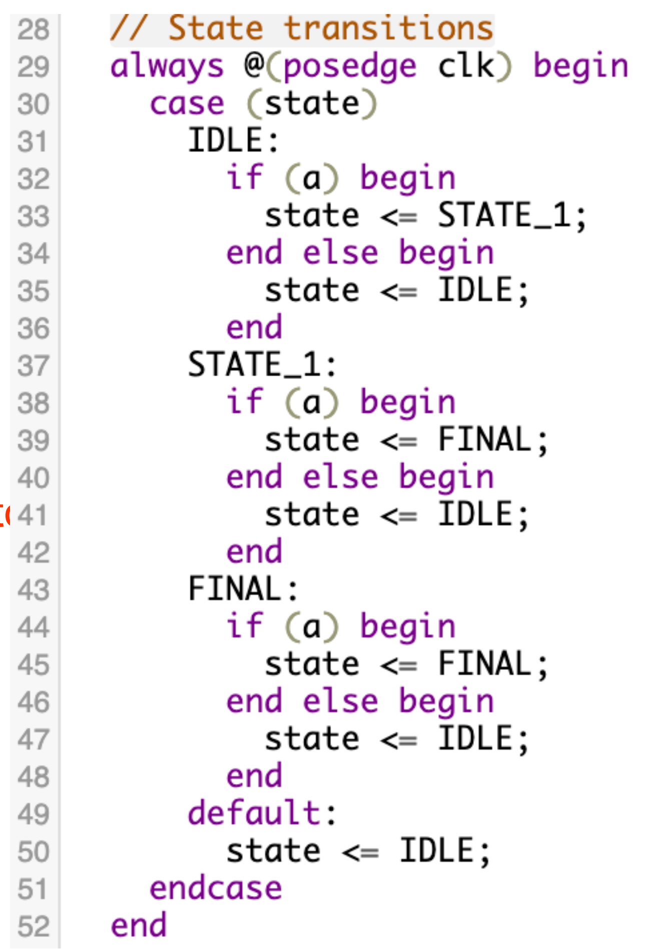
```
module mux4 (in0, in1, in2, in3, select, out);
   input in0,in1,in2,in3;
   input [1:0] select;
   output      out;
   reg         out;

  always @ (in0 in1 in2 in3 select)
  case (select)
      2'b00: out=in0;
      2'b01: out=in1;
      2'b10: out=in2;
      2'b11: out=in3;
   endcase
endmodule // mux4
```

keyword

The statement(s) corresponding t
whichever constant matches
"select" get applied.

https://www.edaplayground.com/



```
// State transitions
always @(posedge clk) begin
  case (state)
    IDLE:
      if (a) begin
        state <= STATE_1;
      end else begin
        state <= IDLE;
      end
    STATE_1:
      if (a) begin
        state <= FINAL;
      end else begin
        state <= IDLE;
      end
    FINAL:
      if (a) begin
        state <= FINAL;
      end else begin
        state <= IDLE;
      end
    default:
      state <= IDLE;
  endcase
end
```

Blocking vs. Nonblocking Assignments

- Verilog supports two types of assignments within always blocks, with subtly different behaviors.
  - ❑ Blocking assignment (=): evaluation and assignment are immediate

```
always @(*) begin
  x = a | b;       // 1. evaluate a|b, assign result to x
  y = a ^ b ^ c;   // 2. evaluate a^b^c, assign result to y
  z = b & ~c;      // 3. evaluate b&(~c), assign result to z
end
```

  - ❑ Nonblocking assignment (<=): all assignments deferred to end of simulation time step after all right-hand sides have been evaluated (even those in other active always blocks)

```
always @(*) begin
  x <= a | b;      // 1. evaluate a|b, but defer assignment to x
  y <= a ^ b ^ c;  // 2. evaluate a^b^c, but defer assignment to y
  z <= b & ~c;     // 3. evaluate b&(~c), but defer assignment to z
  // 4. end of time step: assign new values to x, y and z
end
```

Assignment Styles for Sequential Logic

```
module blocking(
  input in, clk,
  output reg out
);
  reg q1, q2;
  always @(posedge clk) begin
    q1 = in;
    q2 = q1;
    out = q2;
  end

endmodule
```

```
module nonblocking(
  input in, clk,
  output reg out
);
  reg q1, q2;
  always @(posedge clk) begin
    q1 <= in;
    q2 <= q1;
    out <= q2;
  end

endmodule
```

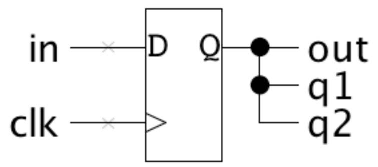
Use Nonblocking for Sequential Logic

```
always @(posedge clk) begin
    q1 = in;
    q2 = q1;    // uses new q1
    out = q2;   // uses new q2
  end
```

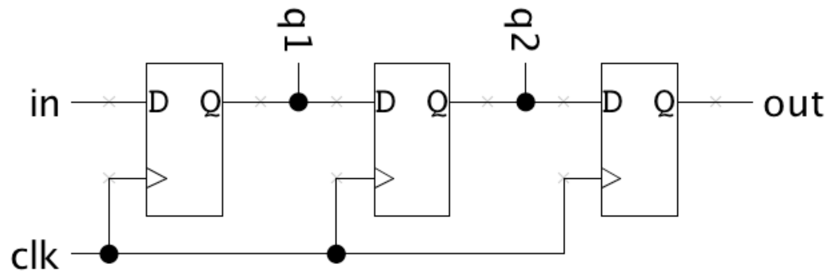
```
always @(posedge clk) begin
    q1 <= in;
    q2 <= q1;    // uses old q1
    out <= q2;   // uses old q2
  end
```

("old" means value before clock edge, "new" means the value after most recent assignment)

"At each rising clock edge, q1 = in.
After that, q2 = q1.
After that, out = q2.
Therefore out = in."



"At each rising clock edge, q1, q2, and out simultaneously receive the old values of in, q1, and q2."



- ❑ Blocking assignments **do not** reflect the intrinsic behavior of multi-stage sequential logic
- ❑ Guideline: use **nonblocking** assignments for sequential always blocks

Simplified Verilog Guidelines

- Combinational logic:
  - Continuous Assignment:

```verilog
assign a = b & c;
```

  - Always block with @(*)

```verilog
always @(*) begin
        a = b & c;  // blocking statement
end
```

- Sequential logic:
  - Always block with @(posedge clk)

```verilog
always @(posedge clk) begin
        a <= b & c;  // nonblocking statement
end
```

$$\begin{tikzcd}
\boxed{\text{assign statement}} \arrow[r] & \texttt{wire} \\
\boxed{\text{always statement}} \arrow[r] & \texttt{reg}
\end{tikzcd}$$

Verilog in EECS 151/251A

- We use behavioral modeling at the bottom of the hierarchy
- Use instantiation to:
  - 1) build hierarchy and,
  - 2) map to FPGA and ASIC resources not supported by synthesis.
- Use named ports.
- Verilog is a big language. This is only an introduction.
  - Harris & Harris book chapter 4 is a good source.
  - We will be introducing more useful constructs throughout the semester. Stay tuned!

Final Thoughts on Verilog Examples

A large part of digital design is knowing how to write Verilog that gets you the desired circuit.

First understand the circuit you want then figure out how to code it in Verilog.

If you try to write Verilog without a clear idea of the desired circuit, you will struggle.

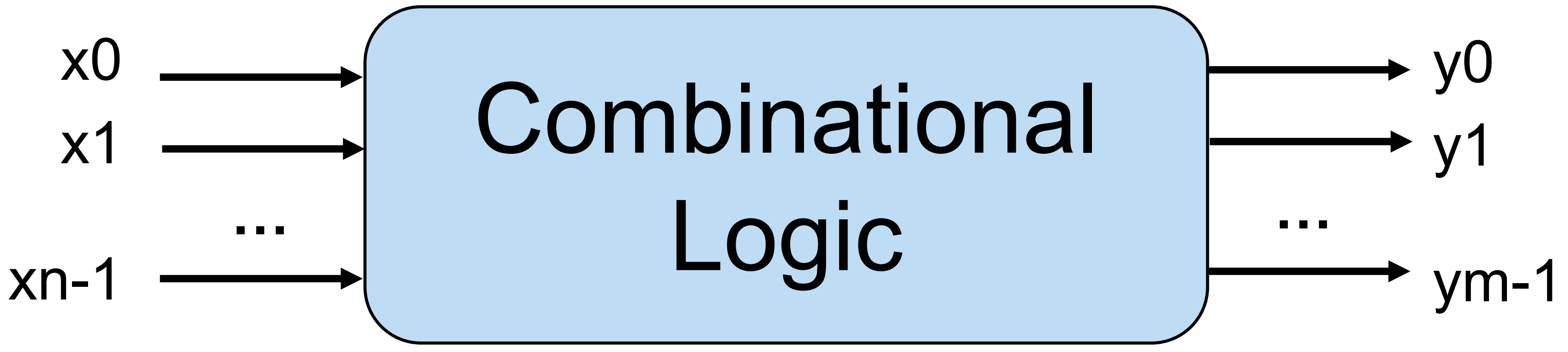
Administrivia

- Hope you enjoy Lab 2!
  - Don’t fall behind.
- Discussion 1 recording is posted.
- HW1 is due this Friday.
  - HW2 will be released this week.
- Help your TA to better support you!
  - Try to debug first: load the waveform.
  - Articulate the problem.
    - Talk to your fellow students.
- Apple-Berkeley event in Woz today.
  - Food and Boba!

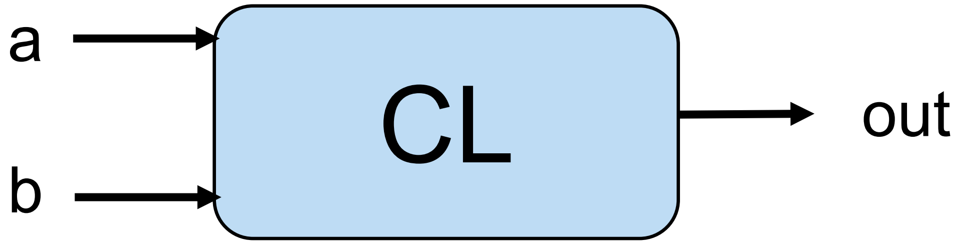
- **Combinational Logic**
  - **Introduction**
  - Boolean Algebra
    - DeMorgan's Law
    - Sum of Products
    - Product of Sums

Combinational Logic

- The outputs depend *only* on the current values of the inputs.
  - Memoryless: compute the output values using the current inputs.
- If we change X, Y will change immediately (well almost!)
  - There is an implementation dependent delay from X to Y.



Combinational Logic Example



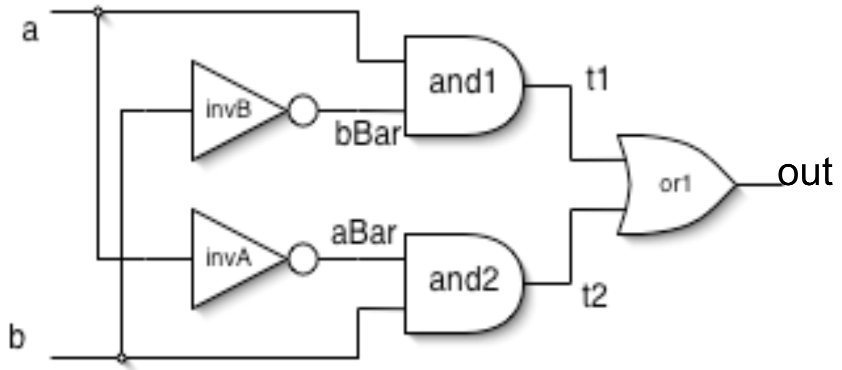
Boolean Equations:

$$y = a\bar{b} + \bar{a}b$$
$$= a\ XOR\ b$$

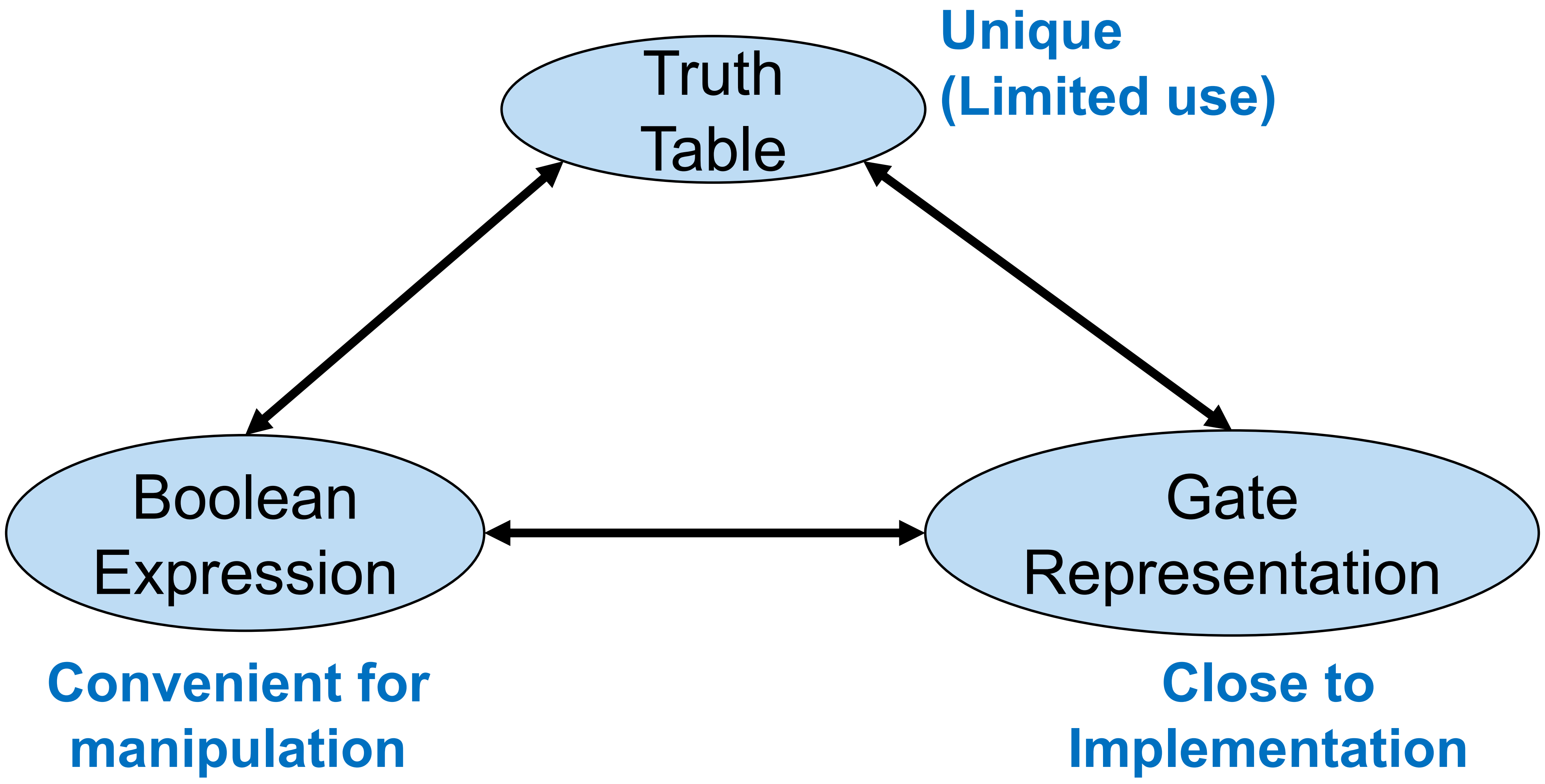
Truth Table Description:

a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

Gate Representations:



Relationship Among Representations

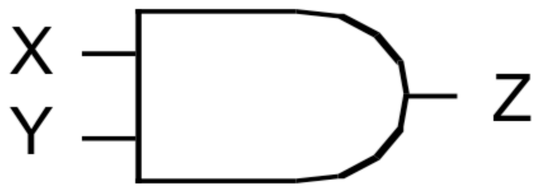


Boolean Algebra Background

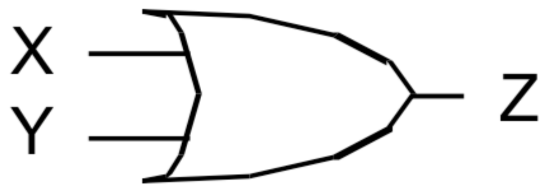
- Why are they called “Logic Circuits”?
  - Logic: The study of the principles of reasoning.
  - The 19th Century Mathematician, George Boole, developed a math. system (algebra) involving logic, Boolean Algebra.
    - His variables took on TRUE, FALSE.
  - Later Claude Shannon (father of information theory) showed (in his Master's thesis!) how to map Boolean Algebra to digital circuits in 1937.
  - Shannon’s work became the foundation of digital circuit design.

Boolean Algebra Fundamentals

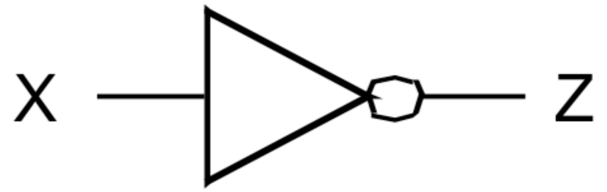
- Two elements {0, 1}
- Two binary operators: AND (·), OR (+)
- One unary operator: NOT (¯ , ´)



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



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



X	Z
0	1
1	0

Boolean Operations of 2 variables

- Given two variables (x, y), 16 logic functions

X	Y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_A	F_B	F_C	F_D	F_E	F_F
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

$F_0 = 0 \quad F_1 = X \bullet Y \quad F_3 = X \quad F_5 = Y \quad F_7 = X + Y$

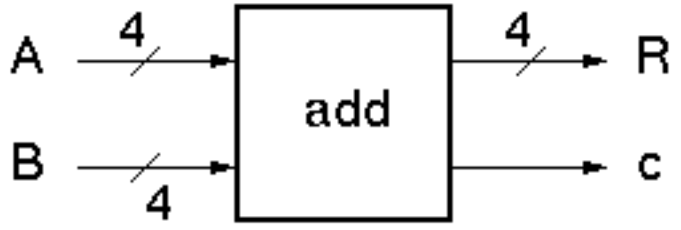
$F_A = \overline{Y} \quad F_F = 1 \quad F_C = \overline{X} \quad F_8 = \overline{X + Y} \quad F_E = \overline{X \bullet Y}$

$F_6 = X \oplus Y \quad F_9 = \overline{X \oplus Y}$

XOR *XNOR* *NOR* *NAND*

Decomposition in Digital Design

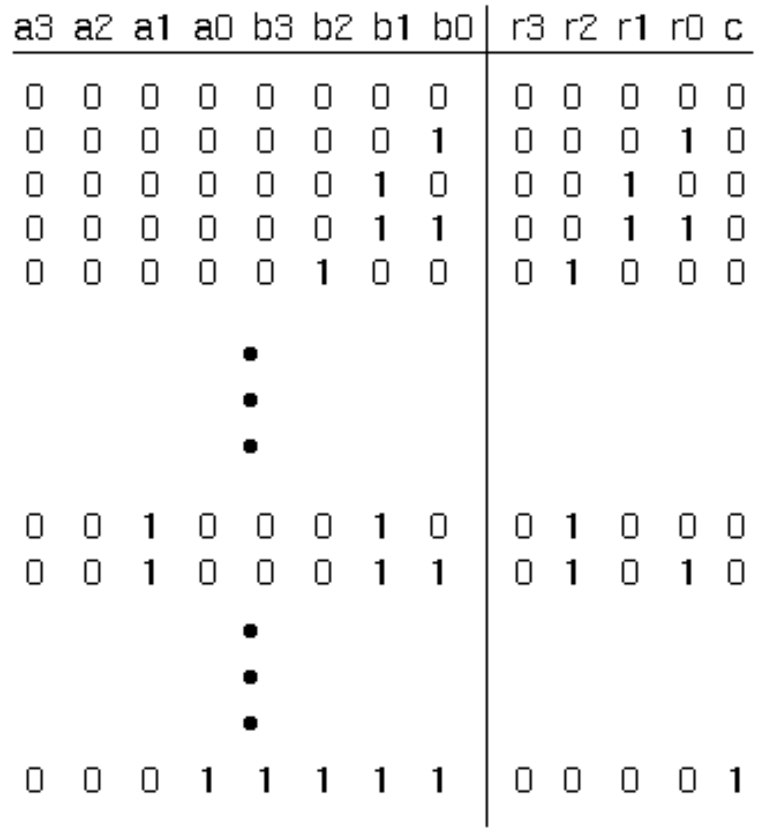
- For n inputs, need 2^n rows in truth table.



R = A + B,

c is carry out

- Truth Table Representation:

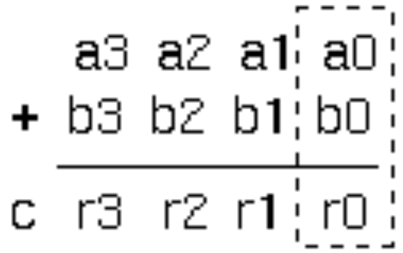


a3	a2	a1	a0	b3	b2	b1	b0	r3	r2	r1	r0	c
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	0	1	1	0	0	1	1	0
0	0	0	0	0	1	0	0	0	1	0	0	0
			⋮									
0	0	1	0	0	0	1	0	0	1	0	0	0
0	0	1	0	0	0	1	1	0	1	0	1	0
			⋮									
0	0	0	1	1	1	1	1	0	0	0	0	1

256 rows!

Decomposition in Digital Design

- Motivate the adder circuit design by hand addition:

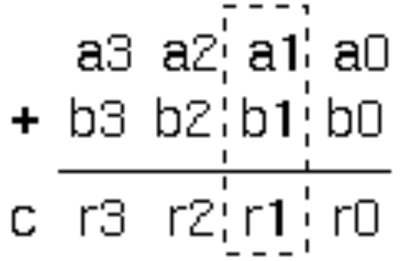


- Add a0 and b0 as follows:

a	b	r	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

carry to next stage

r = ab’ + a’b = a XOR b
c = a AND b = ab



- Add a1 and b1 as follows:

ci	a	b	r	co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	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

r = a xor b xor c_i
co = ab + (a + b)c_i

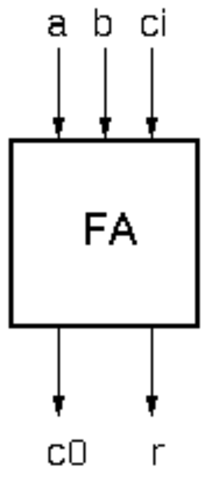
Decomposition in Digital Design

- In general:

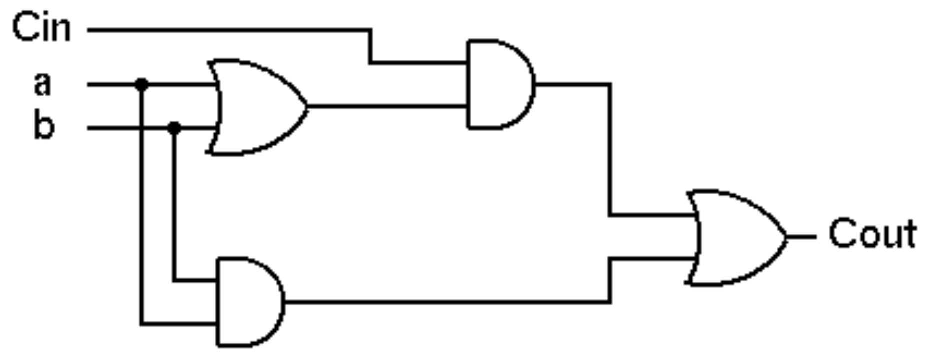
  $r_i = a_i \oplus b_i \oplus c_{in}$

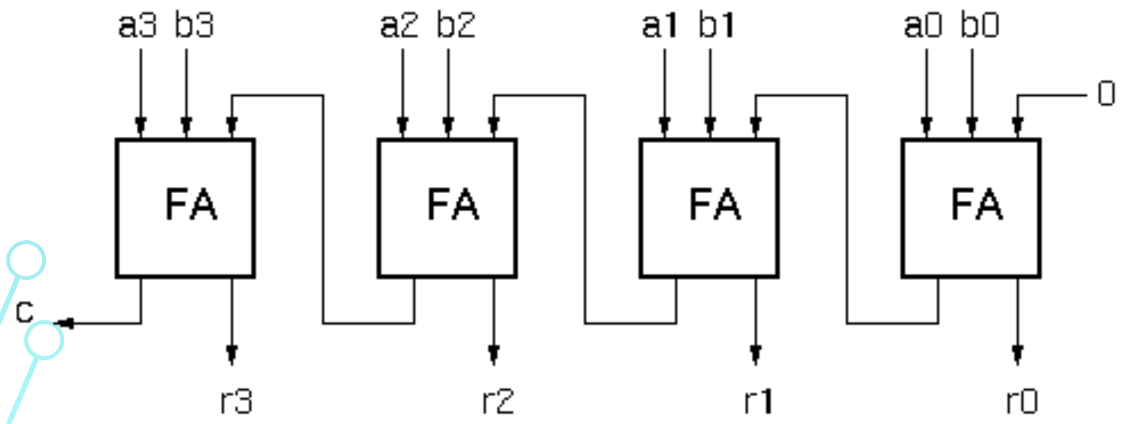
  $c_{out} = a_i c_{in} + a_i b_i + b_i c_{in} = c_{in}(a_i + b_i) + a_i b_i$

- Now, the 4-bit adder:

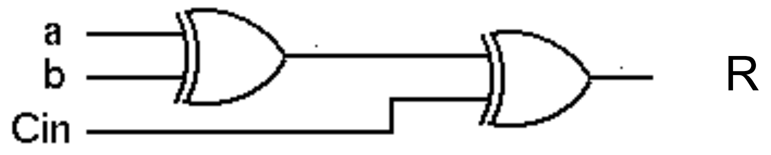


"Full adder cell"





"ripple carry" adder



- Combinational Logic
  - Introduction
  - Boolean Algebra
    - DeMorgan’s Law
    - Sum of Products
    - Product of Sums

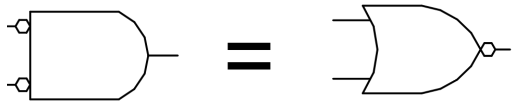
Laws of Boolean Algebra

- Identities:
  - X+0=X, X•1=X
  - X+1=1, X•0=0
- Idempotence:
  - X+X=X, X•X=X
- Complements:
  - X+X´=1, X•X´=0
- Commutative
  - X+Y=Y+X, X•Y=Y•X
- Associative:
  - (X + Y) + Z= X + (Y + Z)= X + Y + Z
  - (X • Y) • Z = X • (Y • Z) = X • Y • Z
- Distributive:
  - X • (Y+Z) = (X•Y) + (X•Z)
  - X + (Y•Z) = (X+Y) • (X+Z)
- Absorptive:
  - X + (X•Y) = (X) • (1+Y)=X
  - X • (X+Y) = (X+0) • (X+Y)=X +(0•Y)=X
- Duality
  - AND -> OR and vice versa
  - 0 -> 1 and vice versa
  - Leave literals unchanged

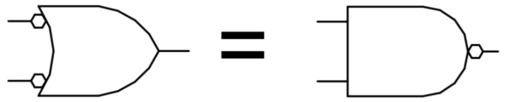
$$\{F(x_1, x_2, \ldots, x_n, 0, 1, +, \bullet)\}^D = \{F(x_1, x_2, \ldots, x_n, 1, 0, \bullet, +)\}$$

DeMorgan's Law

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



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

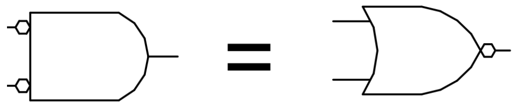


- The product of the complement of each term.
  - Is equal to the complement of the sum of all the terms
- The sum of the complement of each term.
  - Is equal to the complement of the product of all the terms
- Powerful tool in digital design
  - A NAND gate is equivalent to an OR gate with inverted inputs.
  - A NOR gate is equivalent to an AND gate with inverted inputs.

- Bubble Pushing
  - Pushing a bubble from input through the gate
    - Bubble comes out in the output
    - The gate flips from AND to OR or vice versa.

DeMorgan's Law

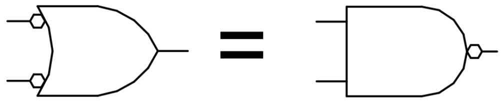
x' y’ = (x + y)'



x'	y'	x' y'
1	1	1
1	0	0
0	1	0
0	0	0

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

x' + y’= (x y)'



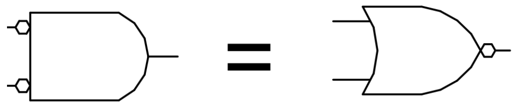
x'	y'	x' + y'
1	1	1
1	0	1
0	1	1
0	0	0

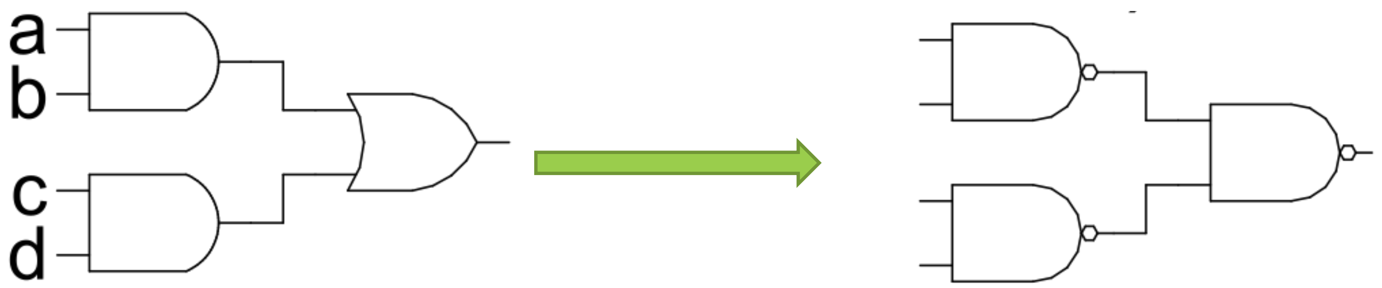
x	y	(x y)'
0	0	1
0	1	1
1	0	1
1	1	0

DeMorgan's Law

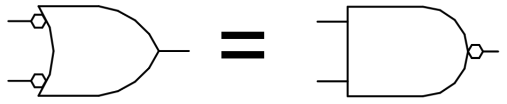
- Mapping from AND/OR -> NAND/NOR

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

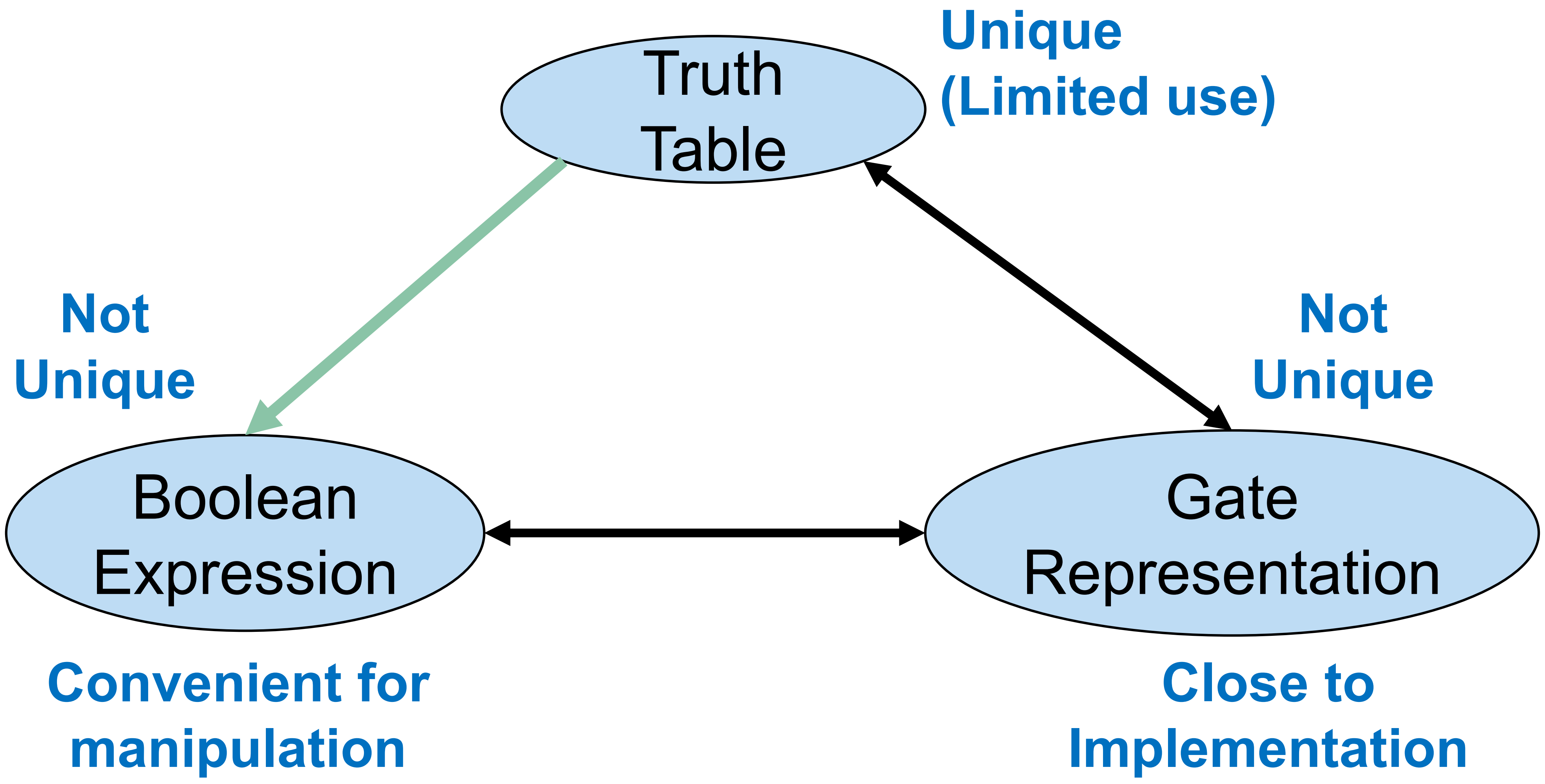




$$x' + y' = (x y)'$$



Relationship Among Representations



Canonical Forms

- From truth table -> Boolean Expression
- Two types:
  - Sum of Products (SOP)
  - Product of Sums (POS)
- Sum of Products
  - a.k.a Disjunctive normal form, minterm expansion
  - Minterm: a product (AND) involving all inputs for the term to be 1
  - SOP: Summing minterms for which the output is True

Minterms	a	b	c	f	f′
a'b'c'	0	0	0	0	1
a'b'c	0	0	1	0	1
a'bc'	0	1	0	0	1
a'bc	0	1	1	1	0
ab'c'	1	0	0	1	0
ab'c	1	0	1	1	0
abc'	1	1	0	1	0
abc	1	1	1	1	0

One product (and) term for each 1 in f:

f = a′bc + ab′c′ + ab′c + abc′ + abc

f′ = a′b′c′ + a′b′c + a′bc′

Quiz

- Derive the sum of products form of $\bar{Y}$ based on the truth table.

a) $\bar{Y} = (A + B)(A + \bar{B})$

b) $\bar{Y} = A\bar{B} + AB$

c) $\bar{Y} = \bar{A}\bar{B} + \bar{A}B$

A	B	Y	$\bar{Y}$
0	0	0	1
0	1	0	1
1	0	1	0
1	1	1	0

Simplifying Sum of Products

- Canonical Forms are usually not minimal:
- Example:

```
f  = a'bc + ab'c' + ab'c + abc' + abc  (xy' + xy = x)
```

```
f' = a'b'c' + a'b'c + a'bc'
   =
```

x + x'y = x + y

- Recall distributive theorem X+YZ = (X+Y)(X+Z)

Simplifying Sum of Products

- Canonical Forms are usually not minimal:
- Example:

```
f  = a'bc + ab'c' + ab'c + abc′ + abc  (xy' + xy = x)
   = a'bc + ab' + ab
   = a'bc + a
   = a + bc

f' = a'b'c' + a'b'c + a'bc'
   = a'b' + a'bc'
   = a' ( b' + bc' )
   = a' ( b' + c' )
```

$x + x'y = x + y$

- Recall distributive theorem $X+YZ = (X+Y)(X+Z)$

Canonical Forms

- From truth table -> Boolean Expression
- Two types:
  - Sum of Products (SOP)
  - Product of Sums (POS)
- Product of Sums:
  - a.k.a. conjunctive normal form, maxterm expansion
  - Maxterm: a sum (OR) involving all inputs for the term to be 0.
  - POS: Product (AND) maxterms for which the output is FALSE
  - Can obtain POSs from applying DeMorgan's law to the SOPs of F (and vice versa)

Maxterms	a	b	c	f	f'
a+b+c	0	0	0	0	1
a+b+c′	0	0	1	0	1
a+b′+c	0	1	0	0	1
a+b′+c′	0	1	1	1	0
a′+b+c	1	0	0	1	0
a′+b+c′	1	0	1	1	0
a′+b′+c	1	1	0	1	0
a′+b′+c′	1	1	1	1	0

One sum (**or**) term for each **0** in f:

```
f  = (a+b+c)(a+b+c')(a+b'+c)
f' = (a+b'+c')(a'+b+c)(a'+b+c')
     (a'+b'+c)(a+b+c')
```

Summary

- Combinational circuits:
  - The outputs only depend on the current values of the inputs (memoryless).
  - The functional specification of a combinational circuit can be expressed as:
    - A truth table
    - A Boolean equation
- Boolean algebra
  - Deal with variables that are either True or False.
  - Map naturally to hardware logic gates.
  - Use theorems of Boolean algebra and Karnaugh maps to simplify equations.
- Common job interview questions ☺