EECS151 : Introduction to Digital Design and ICs

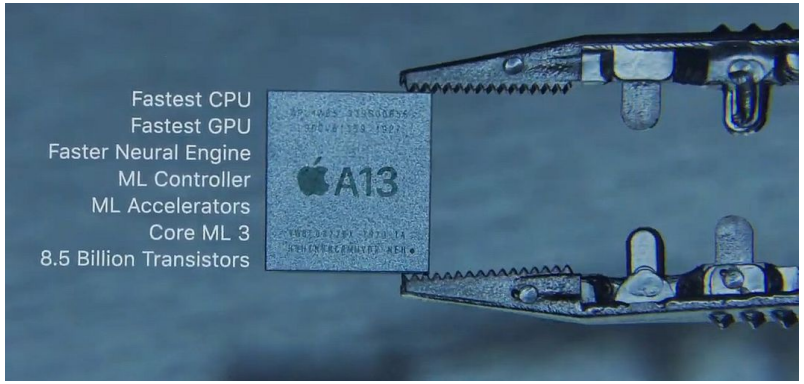
Lecture 5 – Combinational Logic

Bora Nikolić and Sophia Shao

Apple A13 Bionic Chip

(Apple Special Event, Sept 10, 2019)

CPU, GPU, and Neural Engine



Review

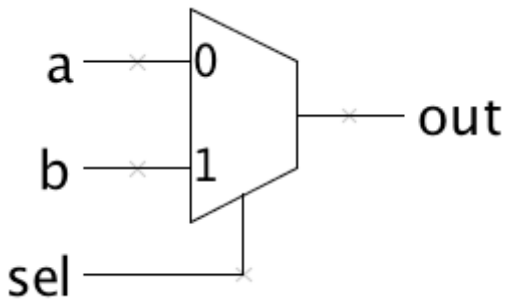
- Verilog is commonly used to describe RTL
  - Structural or
  - Behavioral
- Always block
  - Trigger events under certain conditions
- Generate
  - Use parameters to generalize our designs and testing.
- Sequential Logic
  - Latches and Flip-Flops

Sequential Logic, Take 2.5

The Sequential always Block

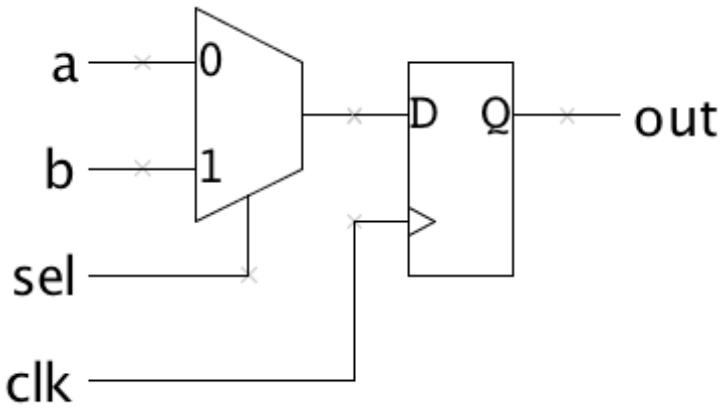
Combinational

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



Sequential

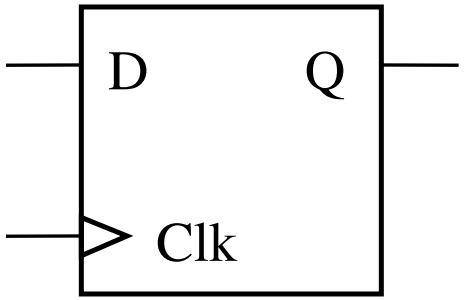
```
module seq(input a, b, sel, clk,
           output reg out);
  always @(posedge clk) begin
    if (sel) out <= b;
    else out <= a;
  end
endmodule
```



Latches vs. Flip-Flops

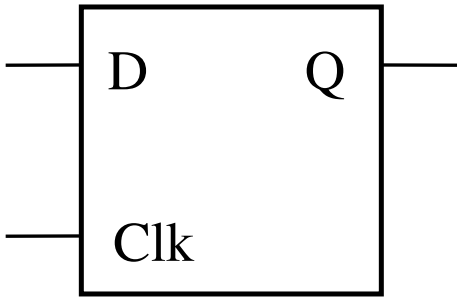
Flip-Flop

```
module flipflop
(
   input clk,
   input d,
   output reg q
);

  always @(posedge clk)
  begin
    q <= d;
  end
endmodule
```



Latch

```
module latch
(
   input clk,
   input d,
   output reg q
);

  always @(clk or d)
  begin
    if ( clk )
      q <= d;
  end
endmodule
```



Importance of the Sensitivity List

D-Register with synchronous clear

```
module dff_sync_clear(
  input d, clearb, clock,
  output reg q);

  always @(posedge clock)
    begin
      if (!clearb) q <= 1'b0;
      else q <= d;
    end
endmodule
```

always block entered only at each positive clock edge

D-Register with asynchronous clear

```
module dff_async_clear(
  input d, clearb, clock,
  output reg q);

  always @(negedge clearb or posedge clock)
    begin
      if (!clearb) q <= 1'b0;
      else q <= d;
    end
endmodule
```

always block entered immediately when (active-low) clearb is asserted

Blocking vs. Nonblocking Assignments

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

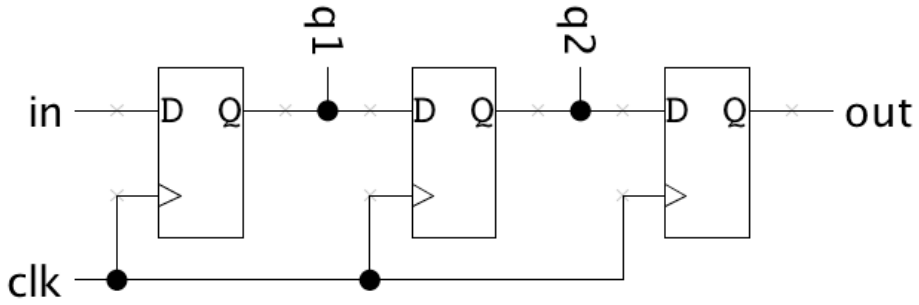
```
always @(*) begin
  x = a | b;      // 1. evaluate a|b, assign result to x
  y = a ^ b ^ c;  // 2. evaluate a^b^c, assign result to y
  z = b & ~c;     // 3. evaluate b&(~c), assign result to z
end
```

  - 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

What we want:
Register-based digital delay line
(a.k.a. shift-register)



Will non-blocking and blocking assignments both produce the desired result?

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

endmodule
```

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

endmodule
```

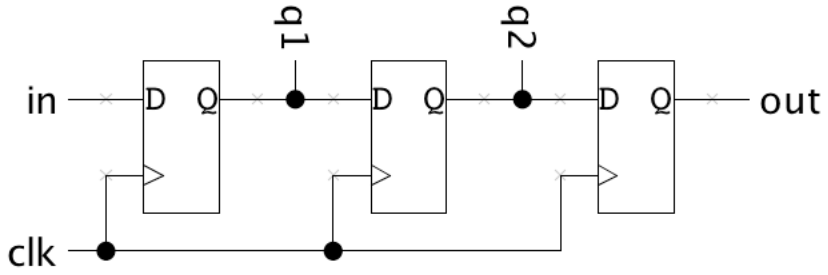
Use Nonblocking for Sequential Logic

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

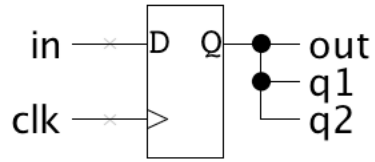
```
always @(posedge clk) begin
    q1 = in;
    q2 = q1;    // uses new q1
    out = q2;   // uses new q2
  end
```

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

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



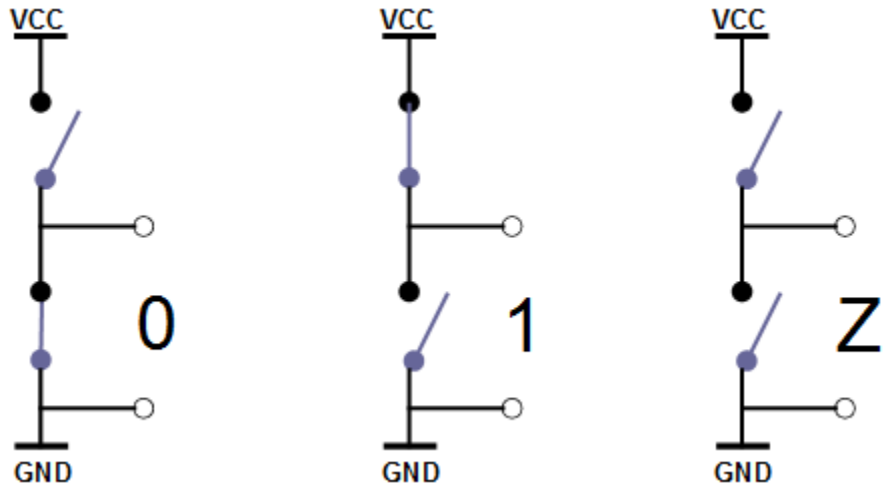
"At each rising clock edge, q1 = in.
After that, q2 = q1.
After that, out = q2.
Therefore out = in."



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

Verilog Logic Values

Logic	Description
0	Logic '0' or false
1	Logic '1' or true
x	Don't care or unknown value
z	High impedance state



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.
- Favor continuous assign and avoid always blocks unless:
  - No other alternative: ex: state elements, case
  - Helps readability and clarity of code: ex: large nested if else
- Use named ports.
- Verilog is a big language. This is only an introduction.
  - Harris & Harris book chapter 4 is a good source.
  - ***Be careful of what you read on the web.*** Many bad examples out there.
  - We will be introducing more useful constructs throughout the semester. Stay tuned!

Final Thoughts on Verilog Examples

Verilog looks like C, but it describes hardware:

Entirely different semantics: multiple physical elements with parallel activities and temporal relationships.

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.

As you get more practice, you will know how to best write Verilog for a desired result.

Be suspicious of the synthesis tools! Check the output of the tools to make sure you get what you want.

Combinational Logic

Introduction

Boolean Algebra

Boolean Simplification

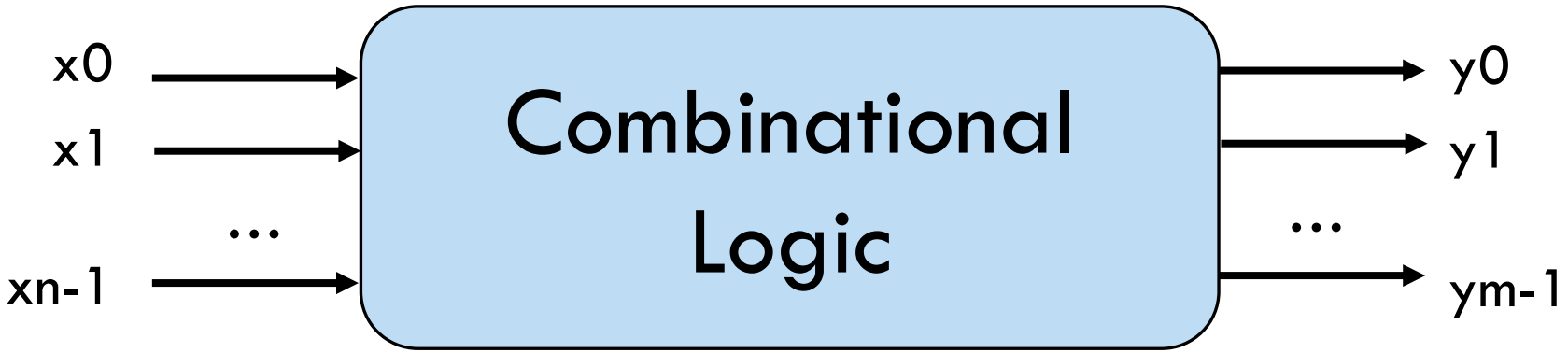
Combinational Logic

Introduction

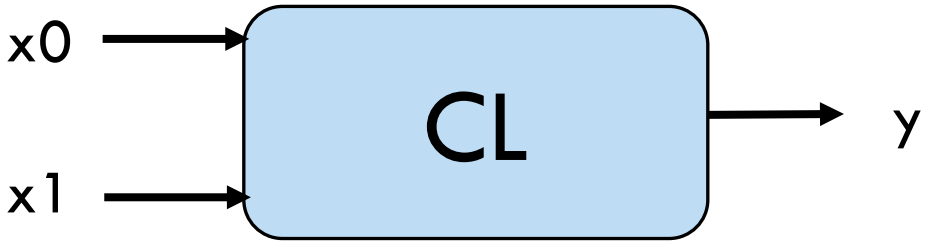
Boolean Algebra
Boolean Simplification

Combinational Logic

- The outputs depend *only* on the current values of the inputs.
  - Memoryless: compute the output values using the current inputs.



Combinational Logic Example



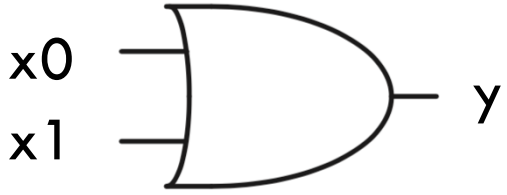
Boolean Equations:

```
y = x0 OR x1
  = x0 + x1
```

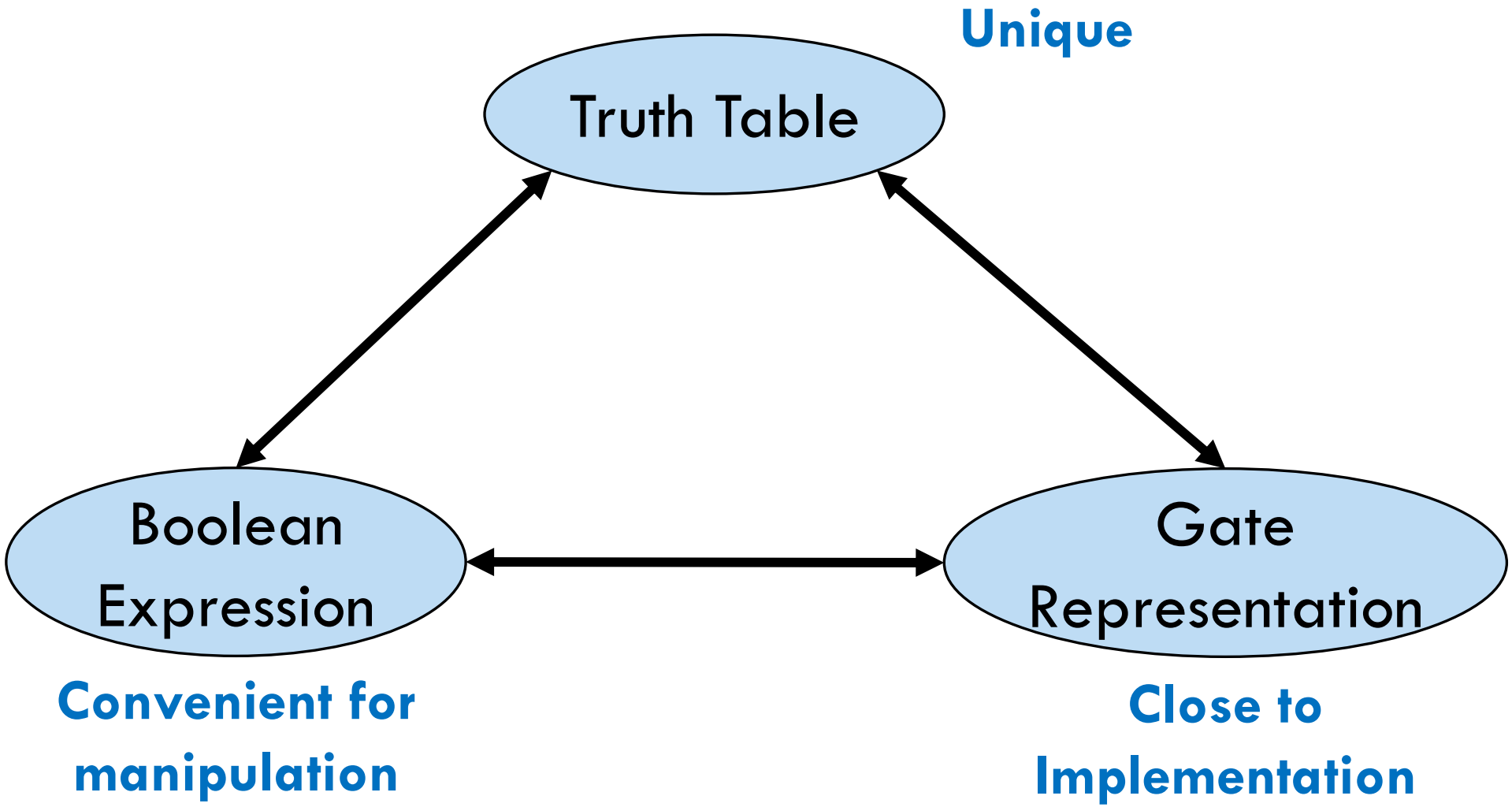
Truth Table Description:

x0	x1	y
0	0	0
0	1	1
1	0	1
1	1	1

Gate Representations:



Relationship Among Representations



Administrivia

- Homework 2 is out. Due Friday.
- Each lab has a max capacity of 24.
  - Thursday -> Monday/Tuesday

Combinational Logic

Introduction

Boolean Algebra

Boolean Simplification

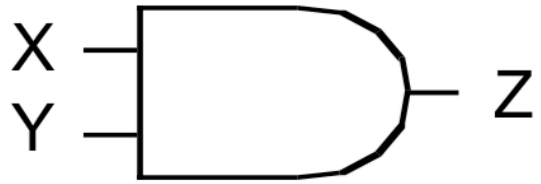
Boolean Algebra Background

- 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.

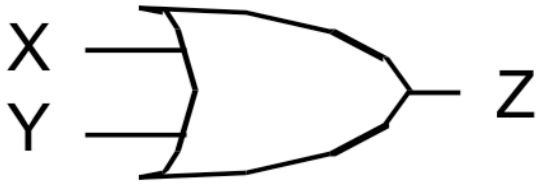
Boolean Algebra Fundamentals

- Two elements {0, 1}
- Two binary operators: AND (·), OR (+)
- One unary operator: NOT ($\bar{}$, $'$)

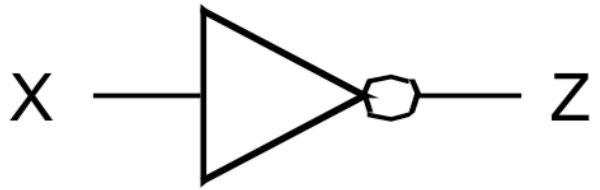
Operator	Description
&&	Logical AND
||	Logical OR
!	Logical NOT



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

- 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

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)
- Duality
  - AND -> OR and vice versa
  - 0 -> 1 and vice versa
  - Leave literals unchanged

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

Proving Distributive Law

- $X \bullet (Y+Z) = (X \bullet Y) + (X \bullet Z)$

X	Y	Z	(Y+Z)	X • (Y+Z)	(X•Y)	(X•Z)	(X•Y) + (X•Z)
0	0	0					
0	0	1					
0	1	0					
0	1	1					
1	0	0					
1	0	1					
1	1	0					
1	1	1					

Proving Distributive Law

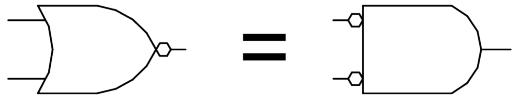
- X • (Y+Z) = (X•Y) + (X•Z)

X	Y	Z	(Y+Z)	X • (Y+Z)	(X•Y)	(X•Z)	(X•Y) + (X•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

DeMorgan's Law

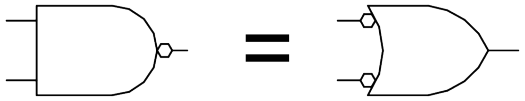
- Procedure for complementing a complex function.

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



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

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

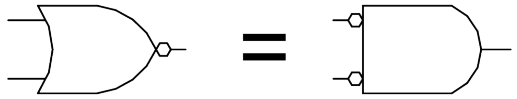


x	y	x'	y'	(x y)'	x' + y'
0	0				
0	1				
1	0				
1	1				

DeMorgan's Law

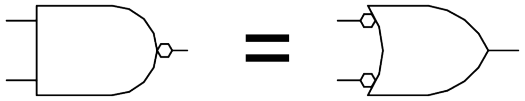
- Procedure for complementing a complex function.

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



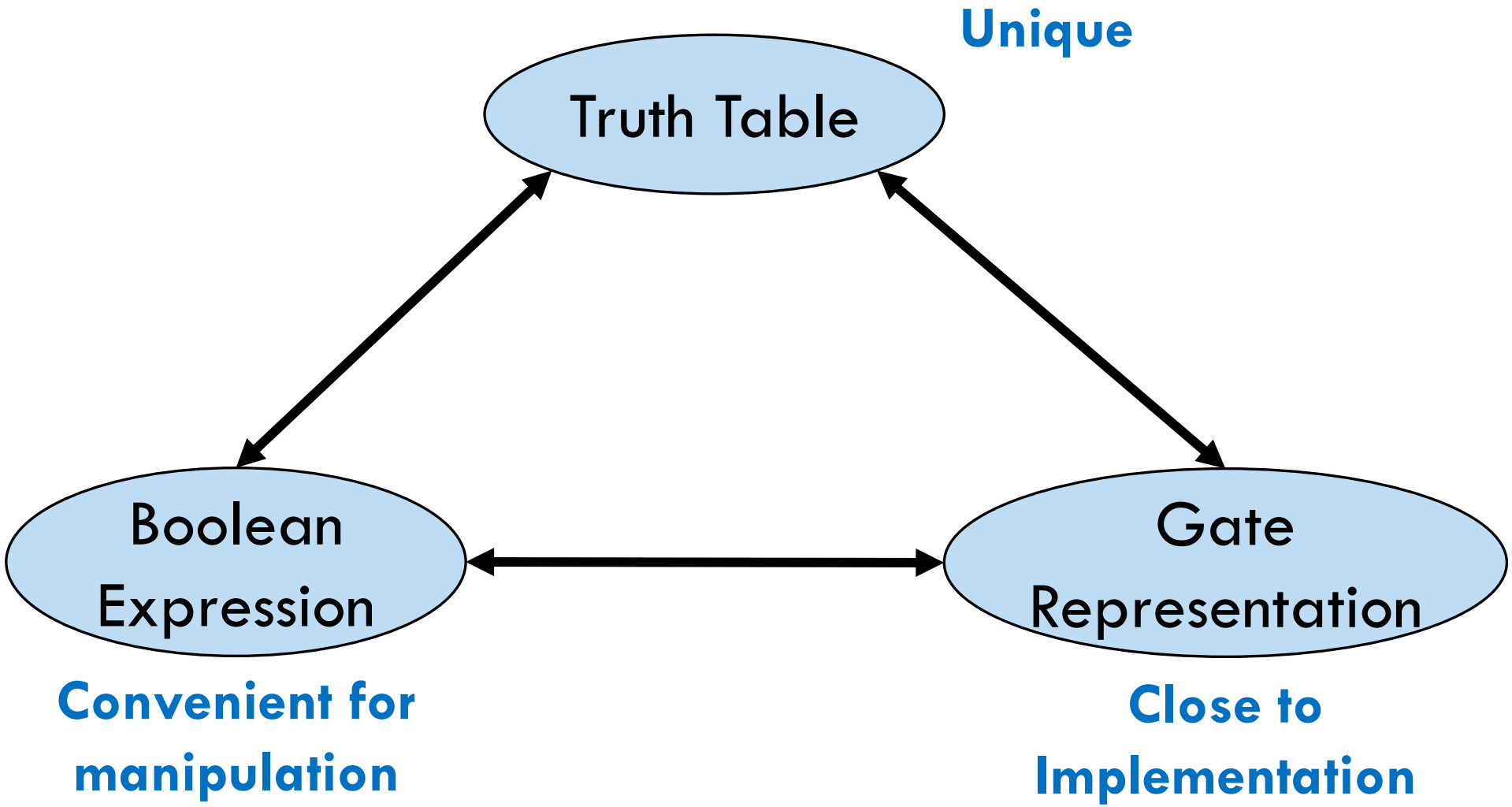
x	y	x'	y'	(x + y)'	x' y'
0	0	1	1	1	1
0	1	1	0	0	0
1	0	0	1	0	0
1	1	0	0	0	0

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



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

Relationship Among Representations



Canonical Forms

- 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
  - 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'
```

Sum of Products (cont.)

- 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' )
   = a'b' + a'c'
```

$x + x'y = x + y$

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

Canonical Forms

- 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
  - 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')
```

Quiz

- Derive the Product of sums 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

http://www.yellkey.com/fear

Combinational Logic

Introduction

Boolean Algebra

Boolean Simplification

Example: Full Adder (FA) Carry out

Cout = a'bc + ab'c + abc' + abc

= a'bc + ab'c + abc' + abc + abc

= a'bc + abc + ab'c + abc' + abc

= (a' + a)bc + ab'c + abc' + abc

= (1)bc + ab'c + abc' + abc

= bc + ab'c + abc' + abc + abc

= bc + ab'c + abc + abc' + abc

= bc + a(b' +b)c + abc' +abc

= bc + a(1)c + abc' + abc

= bc + ac + ab(c' + c)

= bc + ac + ab(1)

= bc + ac + ab

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

Why do Boolean simplification?

- Minimize number of gates in circuit
  - Gates take area
- Minimize amount of wiring in circuit
  - Wiring takes space and is difficult to route
  - Physical gates have limited number of inputs
- Minimize number of gate levels
  - Faster is better
- How to systematically simplify Boolean logics?
  - Use tools!

Practical methods for Boolean simplification

- Still based on Boolean algebra, but more systematic
- 2-level simplification -> multilevel
- Key tool: The Uniting Theorem

$$xy' + xy = x\,(y' + y) = x\,(1) = x$$

ab	f
00	0
01	0
10	1
11	1

$$f = ab' + ab = a(b'+b) = a$$

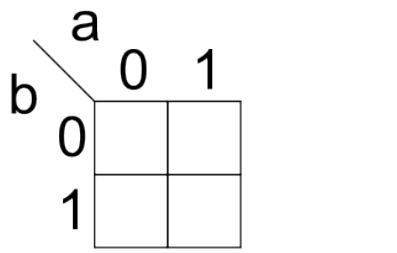
b values change within rows

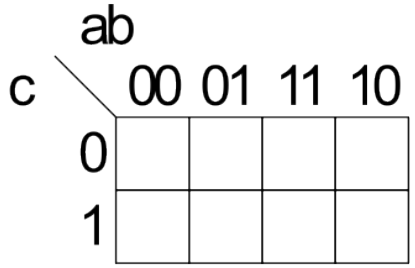
a values don't change

b is eliminated, a remains

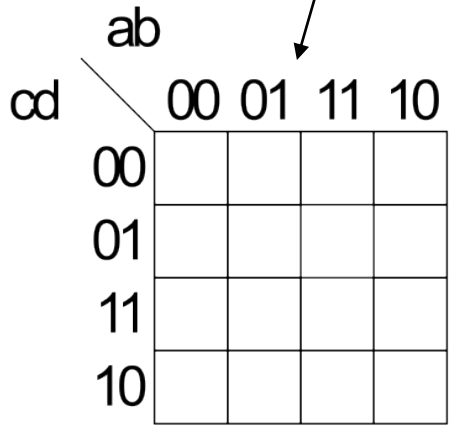
Karnaugh Map Method

- K-map is an alternative method of representing the truth table and to help visual the adjacencies.





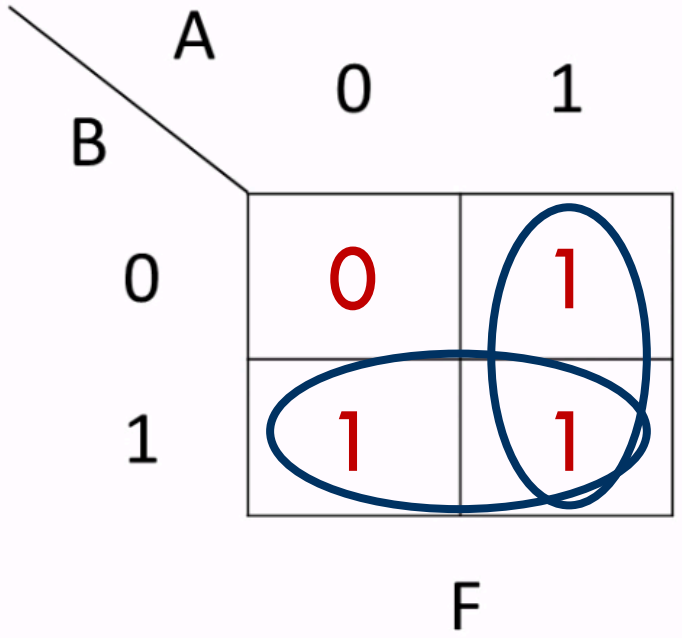
Note: "gray code" labeling.



Karnaugh Map Method

- Adjacent groups of 1's represent product terms

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



$$F = A + B$$

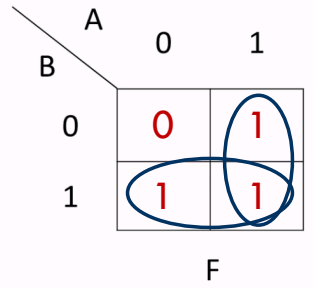
Karnaugh Map Method

1. Draw K-map of the appropriate number of variables.
2. Fill in map with function values from truth table.
3. Form groups of 1's.
   - ✓ Dimensions of groups must be even powers of two (1x1, 1x2, 1x4, …, 2x2, 2x4, …)
   - ✓ Form as large as possible groups and as few groups as possible.
   - ✓ Groups can overlap (this helps make larger groups)
   - ✓ Remember K-map is periodical in all dimensions (groups can cross over edges of map and continue on other side)
4. For each group write a product term.
   - the term includes the "constant" variables (use the uncomplemented variable for a constant 1 and complemented variable for constant 0)
5. Form Boolean expression as sum-of-products.

OR

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

Karnaugh Map

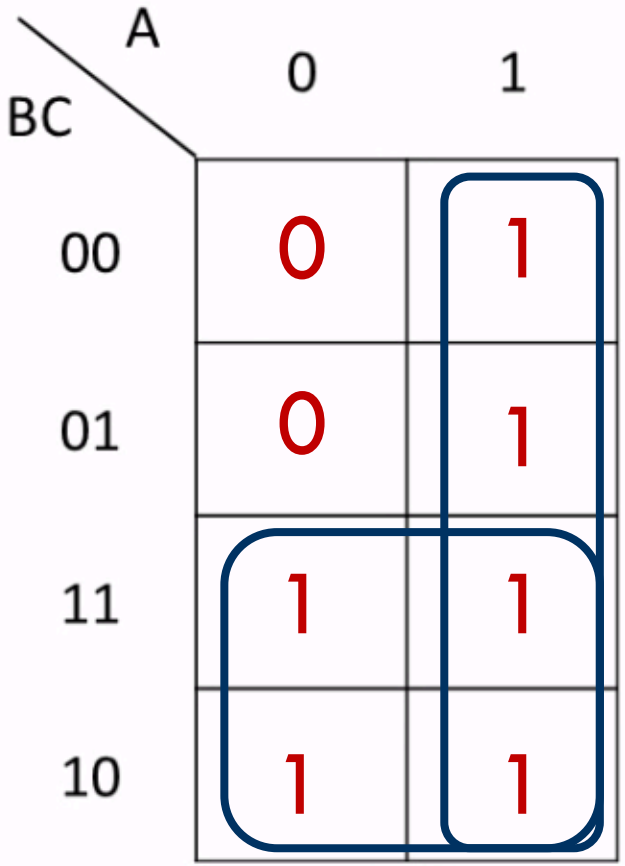


F = A + B

Karnaugh Map Method

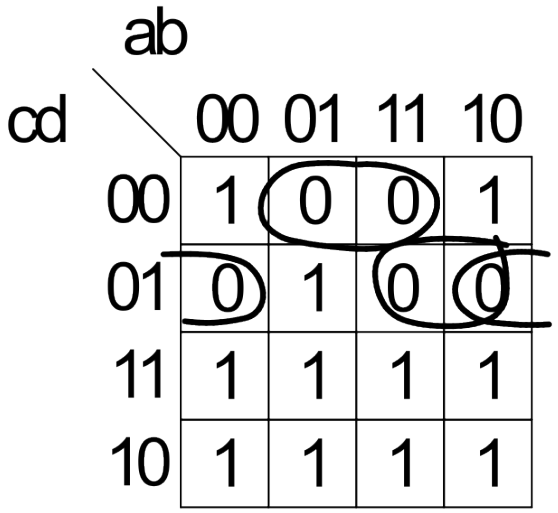
A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

$$F = A + B$$



Product-of-Sums Version

1. Form groups of 0's instead of 1's.
2. For each group write a sum term.
   - the term includes the "constant" variables (use the uncomplemented variable for a constant 0 and complemented variable for constant 1)
3. Form Boolean expression as product-of-sums.



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 ☺