

# CS211 Computer Architecture

## Digital Logic

- Topics
  - Transistors (Design & Types)
  - Logic Gates
  - Combinational Circuits
  - K-Maps

Figures & Tables borrowed from:  
• [http://www.allaboutcircuits.com/vol\\_4/index.html](http://www.allaboutcircuits.com/vol_4/index.html)

## Class Checkpoint

- What have discussed up until now & why:
  - **C Programming language**
    - More low-level than Java.
    - Better idea about what's really going on.
  - **Covered data representation**
    - Computers manipulate data. How is this data stored and manipulated?
  - **Covered Machine-Level representation of programs (assembly language - x86)**
    - Computers don't work on C code (or Java). Need instructions closer to hardware.
- What's next? **Processor Design...** how does machine instructions actually make the processor work?

## But first...

- Before we go into processor design, we're going to cover some topics in **Digital Logic**.
- Book covers this only sparingly, we're going to go into a bit more detail.
- Specifically:
  - Transistors
  - Logic gates
  - Combinational & Sequential Circuits
  - Flip-Flops

## Transistor: Building Block of Computers

- Microprocessors contain millions of transistors
  - **Intel Pentium 4 (2000)**: 48 million
  - **IBM PowerPC 750FX (2002)**: 38 million
  - **IBM/Apple PowerPC G5 (2003)**: 58 million
- Logically, each transistor acts as a switch
- Combined to implement logic functions
  - AND, OR, NOT
- Combined to build higher-level structures
  - Adder, multiplexer, decoder, register, ...
- Combined to build processor

### Simple Switch Circuit

2.9V  $V_{out}$

- **Switch open:**
  - No current through circuit
  - Light is **off**
  - $V_{out}$  is **+2.9V**
- **Switch closed:**
  - Short circuit across switch
  - Current flows
  - Light is **on**
  - $V_{out}$  is **0V**

*Switch-based circuits* can easily represent two states: on/off, open/closed, voltage/no voltage.

©2010 Badri Computer Architecture Assembly II

5

### n-type MOS Transistor

- MOS = Metal Oxide Semiconductor
  - two types: n-type and p-type
- **n-type**
  - when Gate has **positive** voltage, short circuit between #1 and #2 (switch **closed**)
  - when Gate has **zero** voltage, open circuit between #1 and #2 (switch **open**)

Gate #1 #2 GND

Gate = 1

Gate = 0

Terminal #2 must be connected to GND (0V).

©2010 Badri Computer Architecture Assembly II

6

### p-type MOS Transistor

- **p-type** is *complementary* to n-type
  - when Gate has **positive** voltage, open circuit between #1 and #2 (switch **open**)
  - when Gate has **zero** voltage, short circuit between #1 and #2 (switch **closed**)

+2.9V Gate #1 #2

Gate = 1

Gate = 0

Terminal #1 must be connected to +2.9V.

©2010 Badri Computer Architecture Assembly II

7

### Inverter (NOT Gate)

In Out

P-type N-type

In=0 Out=1

P-type N-type

In=1 Out=0

*Truth table*

In	Out
0 V	2.9 V
2.9 V	0 V

In	Out
0	1
1	0

©2010 Badri Computer Architecture Assembly II

8

### OR Gate

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

Add inverter to NOR.

@2010 Badri Computer Architecture Assembly II  
9

### NOR Gate

A=0  
B=1  
C=0

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

Note: Serial structure on top, parallel on bottom.

@2010 Badri Computer Architecture Assembly II  
10

### AND Gate

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

Add inverter to get NAND.

@2010 Badri Computer Architecture Assembly II  
11

### NAND Gate (AND-NOT)

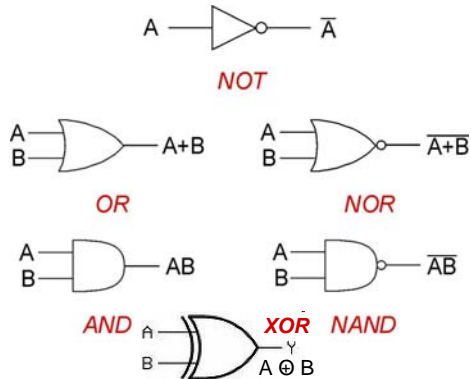
A=0  
B=1  
C=1

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

Note: Parallel structure on top, serial on bottom.

@2010 Badri Computer Architecture Assembly II  
12

## Basic Logic Gates Symbols



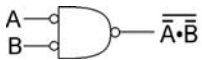
## XOR: truth table $A \oplus B$

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

$$A \oplus B = \overline{A} \cdot B + A \cdot \overline{B}$$

## DeMorgan's Law

- Converting AND to OR (with some help from NOT)
- Consider the following gate:



To convert AND to OR (or vice versa), invert inputs and output.

A	B	$\overline{A}$	$\overline{B}$	$\overline{A \cdot B}$	$\overline{\overline{A} \cdot \overline{B}}$
0	0	1	1	1	0
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	0	1

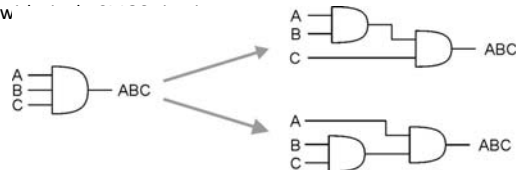
Same as  $A+B$ !

## More than 2 Inputs?

- AND/OR can take any number of inputs.

- AND = 1 if all inputs are 1.
- OR = 1 if any input is 1.
- Similar for NAND/NOR.

- Can implement with multiple two-input gates,



## Building Functions from Logic Gates

### •Combinational Logic Circuit

- output depends only on the current inputs
- stateless

### •Sequential Logic Circuit

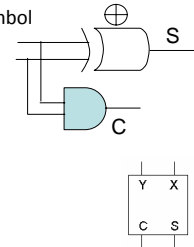
- output depends on the sequence of inputs (past and present)
- stores information (state) from past inputs

•We'll first look at some useful combinational circuits, then show how to use sequential circuits to store information.

## Half adder

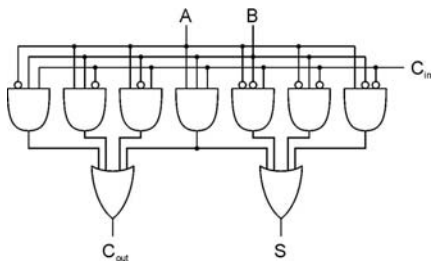
- A half adder is used to add just two bits.
- The result consists of two bits: a sum (the right bit) and a carry out (the left bit)
- Here is the circuit and its block symbol

X	Y	CS
0	0	00
0	1	01
1	0	01
1	1	10



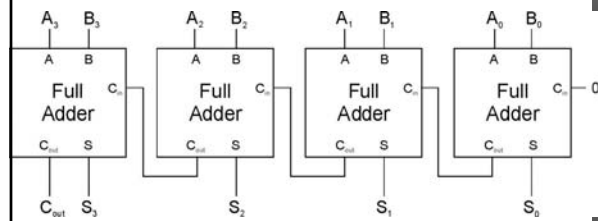
## Full Adder

•Add two bits and carry-in, produce one-bit sum and carry-out.



A	B	C <sub>in</sub>	S
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

## Four-bit Adder (carry-ripple adder)



Disadvantage: Delay through N-1 Stages

## Carry Save adder

- Compute Sum and Carry independently

```

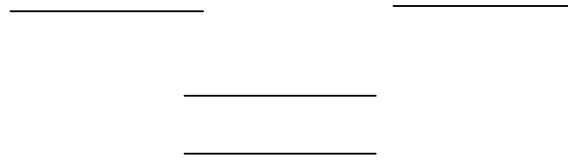
•101      101
•101      101
•-----
•000      101-
    
```

```

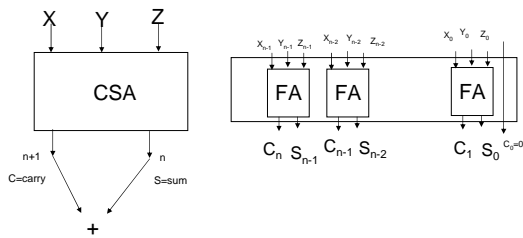
•Then add S + C  000
•                101-
•                -----
•                1010
    
```

## Carry Save adder

- Delay reduced compared to Carry ripple adder
- Add 3 Numbers and Produce two numbers S and C
- Final result is S + shifted carry



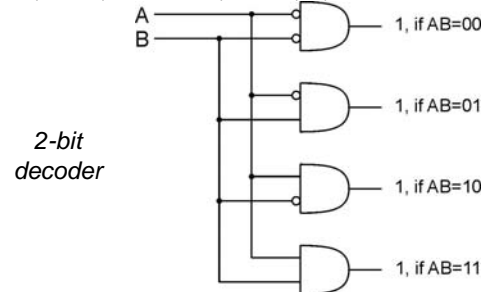
## Carry Save Adder Design



## N Bit Carry Save Adder Block

## Decoder

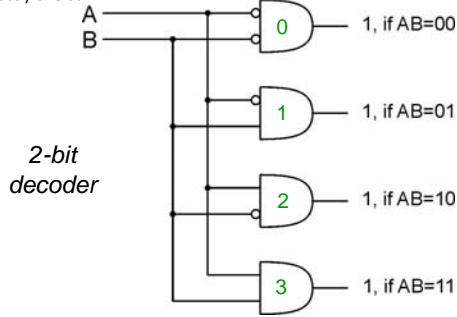
- $n$  inputs,  $2^n$  outputs
- exactly one output is 1 for each possible input pattern



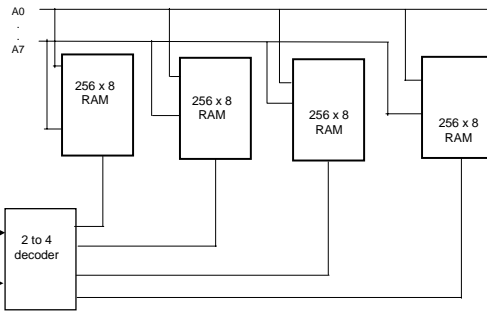
## Decoder

•  $n$  inputs,  $2^n$  outputs

- exactly one output is 1 for each possible input pattern



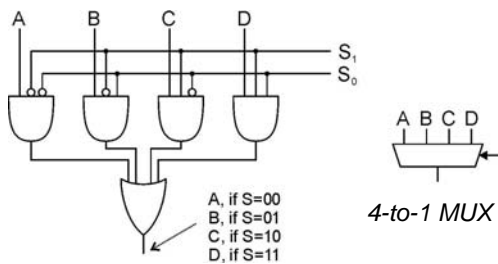
## Selecting Memory



## Multiplexer (MUX)

•  $n$ -bit selector and  $2^n$  inputs, one output

- output equals one of the inputs, depending on selector



## Circuit Design

- Designing circuits is a process...
  1. Have a good idea. What kind of circuit might be useful?
  2. Derive a truth table for this circuit.
  3. Derive a Boolean expression for the truth table.
  4. Build a circuit given the Boolean expression
- Building the circuit involves mapping the Boolean expression to actual gates. This part is easy.
- Deriving the Boolean expression is easy. Deriving a good one is tricky.

## Converting Truth Table to Boolean Expression

sensor inputs

A	B	C	Output
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

- Given a circuit, isolate that rows in which the output of the circuit should be **true**.

## Converting Truth Table to Boolean Expression

sensor inputs

A	B	C	Output
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

sensor inputs

A	B	C	Output
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

$\bar{A}BC = 1$   
 $A\bar{B}C = 1$   
 $AB\bar{C} = 1$   
 $ABC = 1$

- Given a circuit, isolate that rows in which the output of the circuit should be **true**.
- A product term that contains exactly one instance of every variable is called a **minterm**.

## Converting Truth Table to Boolean Expression

sensor inputs

A	B	C	Output
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

$\bar{A}BC = 1$   
 $A\bar{B}C = 1$   
 $AB\bar{C} = 1$   
 $ABC = 1$

Output =  $\bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$

- Given the expressions for each row, build a larger Boolean expression for the entire table.
  - This is a **sum-of-products (SOP)** form.

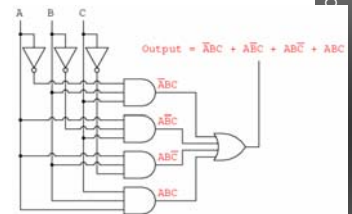
## Converting Truth Table to Boolean Expression

sensor inputs

A	B	C	Output
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

$\bar{A}BC = 1$   
 $A\bar{B}C = 1$   
 $AB\bar{C} = 1$   
 $ABC = 1$

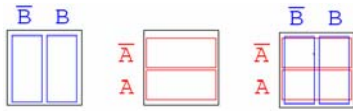
Output =  $\bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$



- Finally build the circuit.
  - Problem: SOP forms are often not minimal.
  - Solution: Make it minimal. We'll go over two ways.



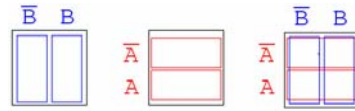
## 2 Variable K-Map



A	B	Output
0	0	0
0	1	1
1	0	0
1	1	1

A <sup>R</sup>	B	1
0	0	
1	0	

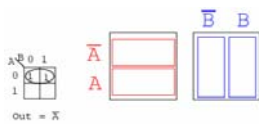
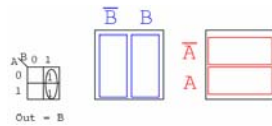
## 2 Variable K-Map



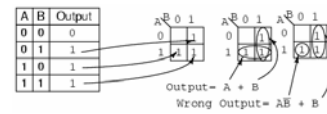
A	B	Output
0	0	0
0	1	1
1	0	0
1	1	1

A <sup>R</sup>	B	1
0	0	
0	1	
1	0	
1	1	

## Finding Commonality

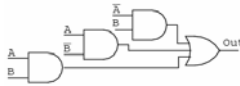


## Finding the "best" solution

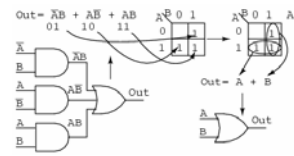
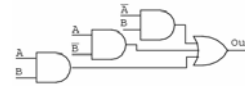


- Grouping become simplified products.
- Both are "correct". "A+B" is preferred.

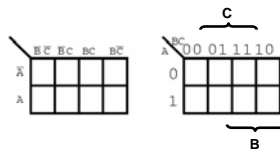
### Simplify Example



### Simplify Example

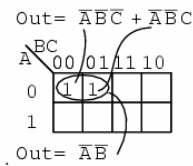
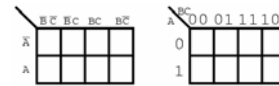


### 3 Variable K-Maps

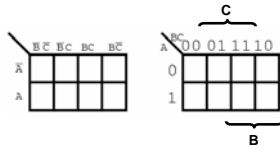


- Note in higher maps, several variables occupy a given axis
- The sequence of 1s and 0s follow a **Gray Code Sequence**.

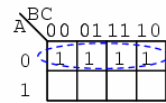
### 3 Variable K-Maps



### 3 Variable K-Maps

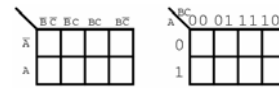


$$\text{Out} = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + \bar{A}BC + A\bar{B}\bar{C}$$

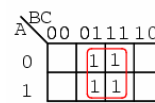


$$\text{Out} = \bar{A}$$

### 3 Variable K-Maps

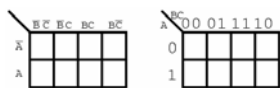


$$\text{Out} = \bar{A}\bar{B}C + \bar{A}BC + A\bar{B}\bar{C} + ABC$$

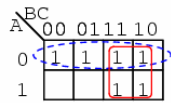


$$\text{Out} = C$$

### 3 Variable K-Maps

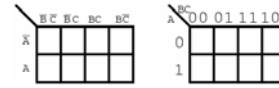


$$\text{Out} = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + \bar{A}BC + A\bar{B}\bar{C} + ABC + A\bar{B}C$$

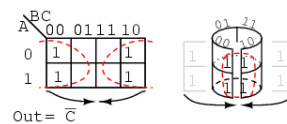


$$\text{Out} = \bar{A} + B$$

### 3 Variable K-Maps

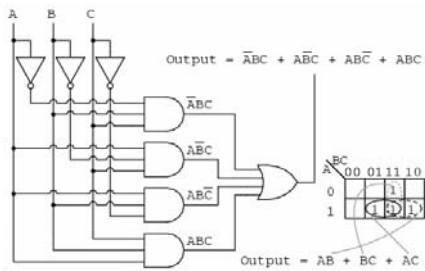


$$\text{Out} = \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + \bar{A}B\bar{C} + AB\bar{C}$$



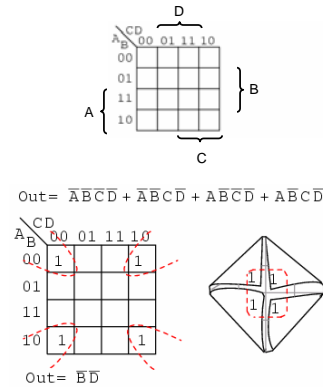
$$\text{Out} = \bar{C}$$

### Back to our earlier example.....

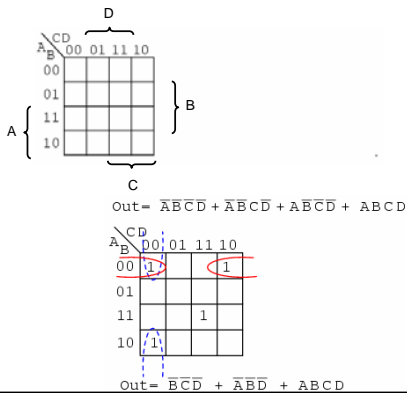


•The K-map and the algebraic produce the same result.

### Up... up... and let's keep going



### Few more examples



### Few more examples

