

# Chapter 2: Bits and Bytes

## Topics

- Why bits?
- Binary Logic
- Representing information as bits
  - Binary/Hexadecimal
  - Byte representations
    - » ints, floats, double precision

# Announcements

**Data Lab is available on the moodle, due Friday Sept 12 by 11:55 pm**

- Sign up at [moodle.cs.colorado.edu](https://moodle.cs.colorado.edu), enrollment key = **systems14**

**No TA recitation next Monday due to Labor Day**

- May hold extra office hours

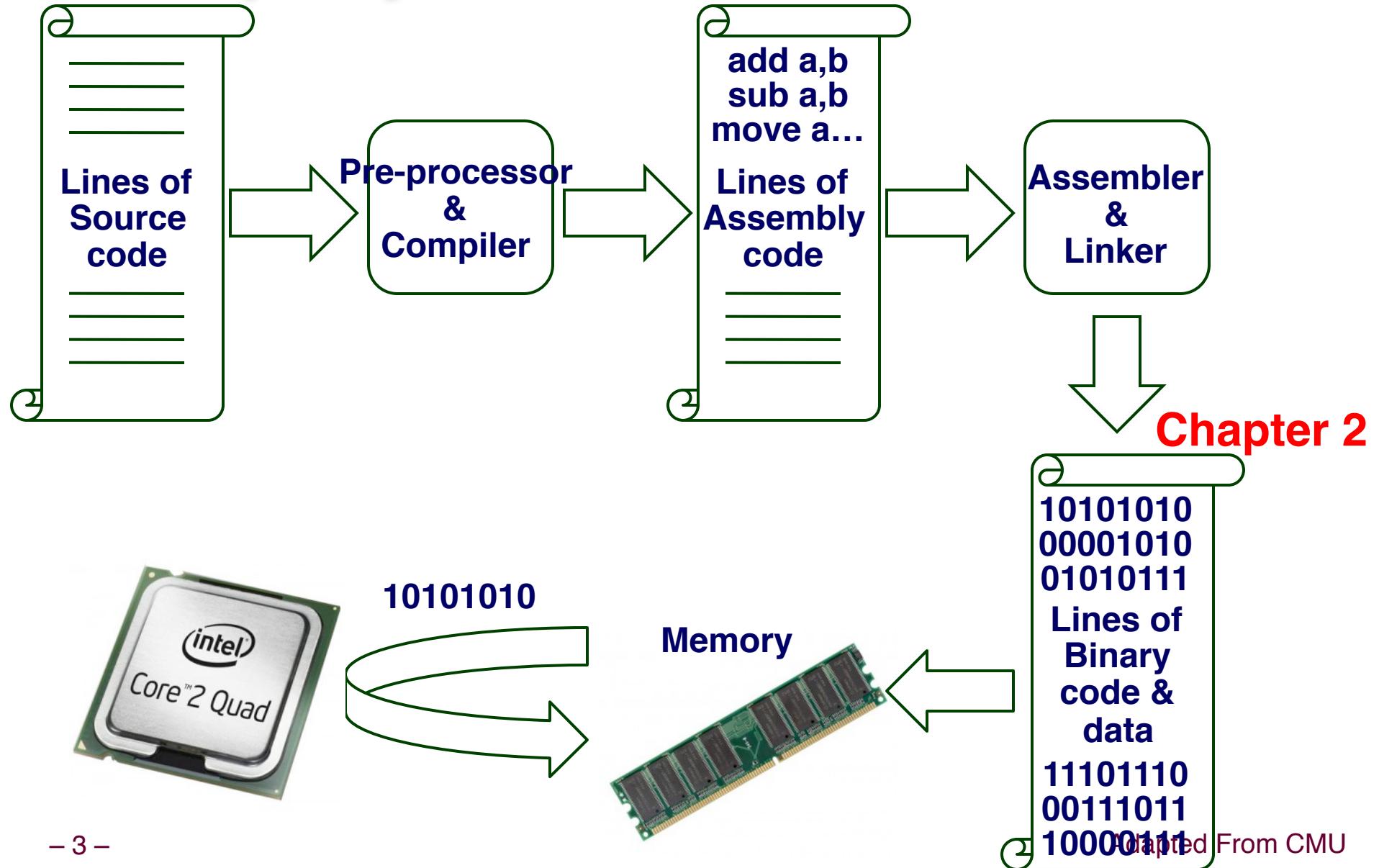
**Lecture slides online: ppt's vs pdf's**

- pdf's lose animation and notes at bottom of each slide

**Essential that you read the textbook in detail & do the practice problems**

- Solutions to practice problems are at the end of each chapter
- Read Chapter 2 (2.1-2.3)

# Recap: Systems In a Nutshell



# Decimal Representation

**Decimal representation (Base 10) is what we humans naturally gravitate to, given 10 digits (fingers)!**

- $7126 \text{ base } 10 = 7126_{10}$   
 $= 7000 + 100 + 20 + 6$   
 $= 7*10^3 + 1*10^2 + 2*10^1 + 6*10^0$

**By convention, we write the most significant digit on the left, followed by the second most significant digit, then the 3<sup>rd</sup> most significant digit, etc., until we reach the least significant digit**

# Binary Representation – Why Bits?

$$\begin{aligned}0101_2 &= 0*2^3 + 1*2^2 + 0*2^1 + 1*2^0 \\&= 0 + 4 + 0 + 1 = 5_{10}\end{aligned}$$

Each binary digit is called a ‘bit’

Base 2 or Binary is easier to represent via electronic circuits than Base 10

- Digital transistor circuits can easily represent a ‘0’ and a ‘1’
  - A ‘1’ = +5 volts, a ‘0’ = 0 volts, so only 2 voltage levels
  - Don’t have to implement 10 different voltage levels as in base 10
- Easy to implement Base 2 arithmetic
  - much of Base 10 arithmetic using Base 2 arithmetic (integers are exact, but floats are approximations)
- Easy to implement digital logic (AND, OR, etc.) with binary

# Digital Logic via Boolean Algebra

Developed by George Boole in 19th Century

- Algebraic representation of logic
  - Encode “True” as 1 and “False” as 0

And “&”

- $A \& B = 1$  only when both  $A=1$  and  $B=1$

		B	
		&	0    1
		0	0    0
A	1	0	1

A	B	&
0	0	0
1	0	0
0	1	0
1	1	1

Truth Tables  
(two forms)

# Common Digital Logic Operations

## And “&”

- $A \& B = 1$  only when both  $A=1$  and  $B=1$

		B
&		0 1
A	0	0 0
	1	0 1

## Or “|”

- $A | B = 1$  when either  $A=1$  or  $B=1$

		0 1
		0 1
0	0	0 1
	1	1 1

## Not “~”

- $\sim A = 1$  only when  $A=0$

		~
		1
0	0	1
	1	0

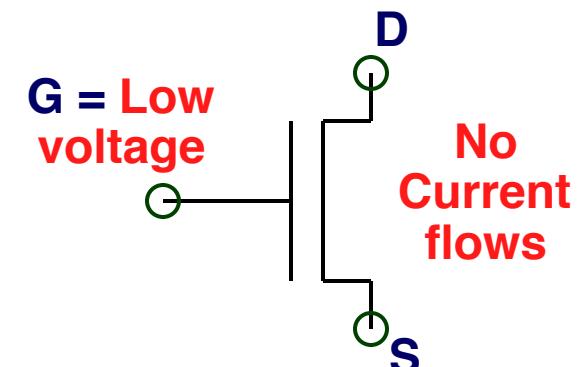
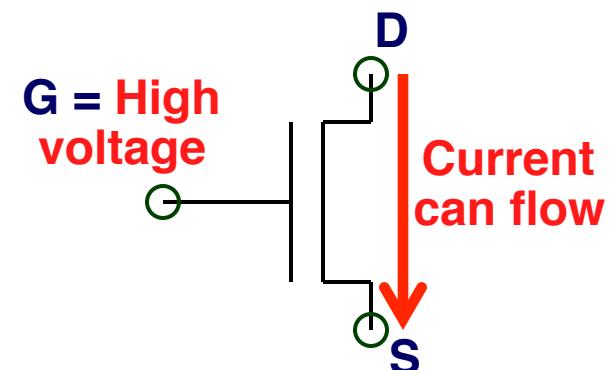
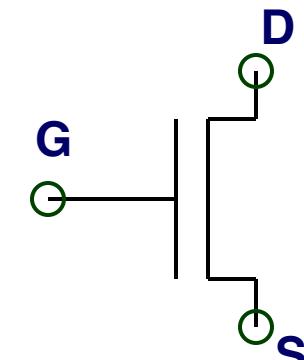
## Exclusive-Or (Xor) “^”

- $A ^ B = 1$  when either  $A=1$  or  $B=1$ , but not both

		0 1
^		0 1
0	0	0 1
	1	1 0

# Implementing Digital Logic

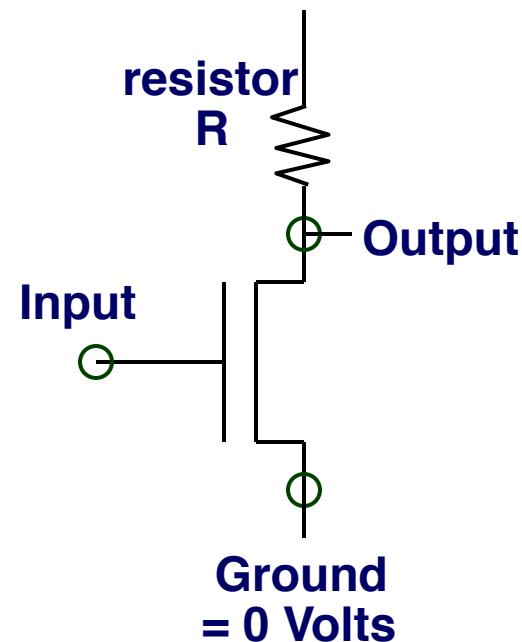
- **Silicon transistor (NMOS)**
  - Three terminals. Input is the Gate G
  - Acts like a simple ON/OFF switch
  - Current flows from high voltage to low voltage
  - When a high voltage is applied to the input Gate G, then this connects the two terminals D and S, i.e. closes the switch, and current flows between them
  - When a low voltage is applied to Gate, then the two terminals are disconnected, i.e. opens the switch, and no current flows



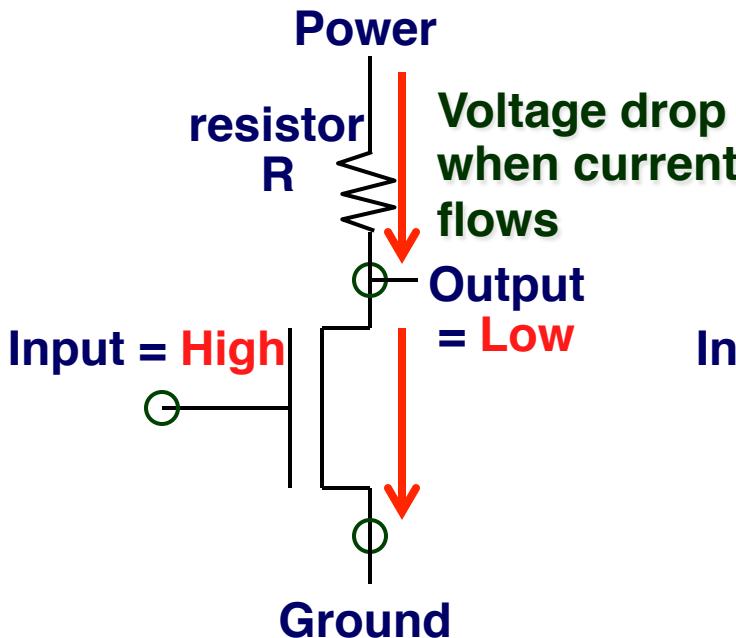
# Implementing an Inverter or Not “~” Operation

An Inverter circuit

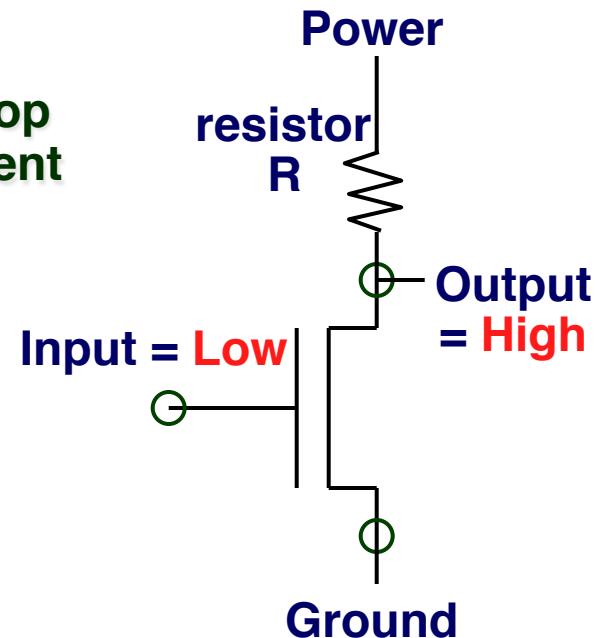
Power = +3 or +5 Volts



High input connects Output to Ground



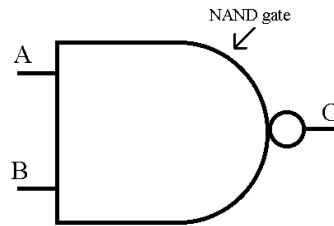
Low input means no current flows, so output stay high



- Thus we have an inverter or Not “~” operator:
  - Input ‘1’ (High voltage) => output ‘0’ (Low voltage)
  - Input ‘0’ (Low voltage) => output ‘1’ (High voltage)

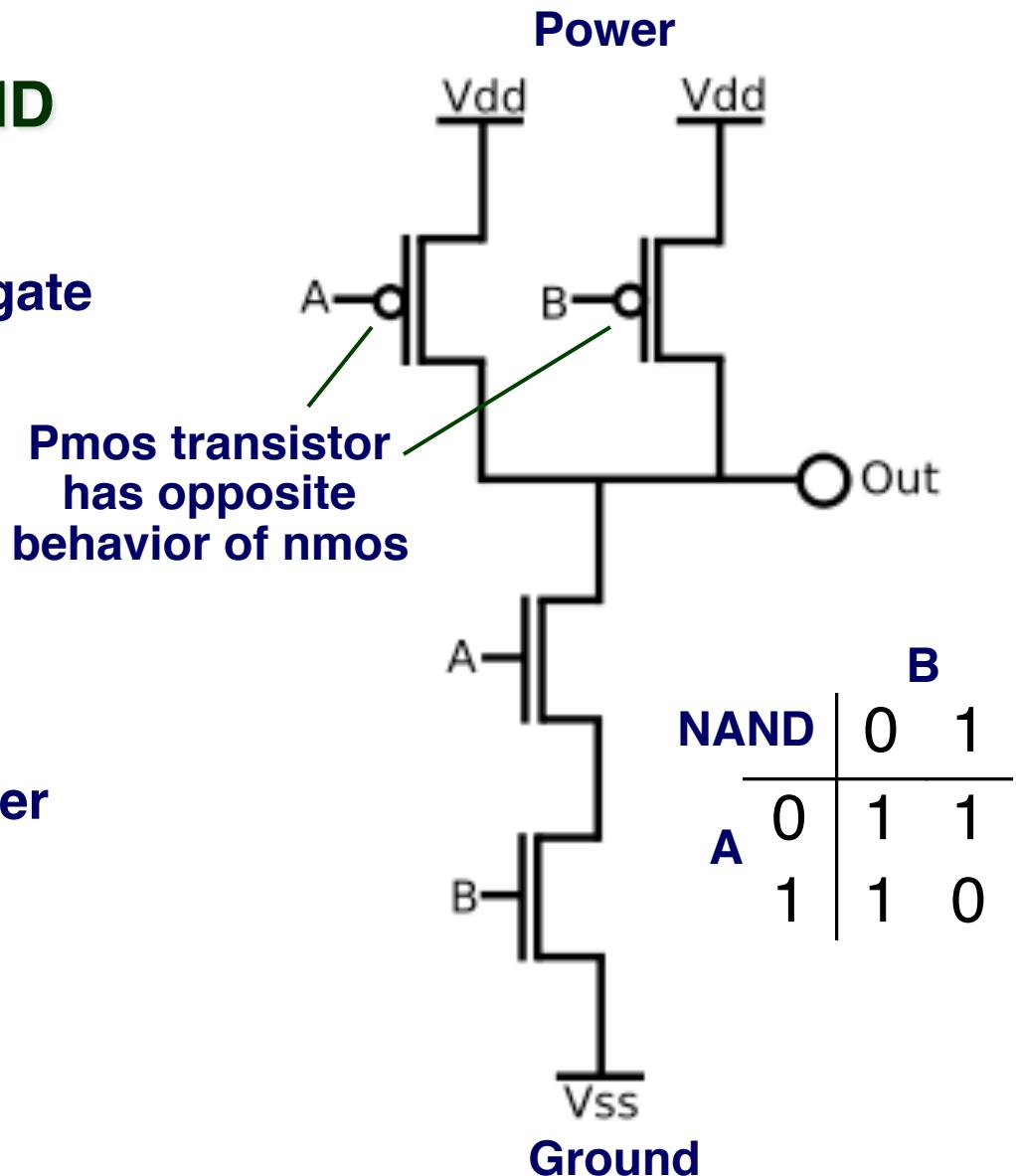
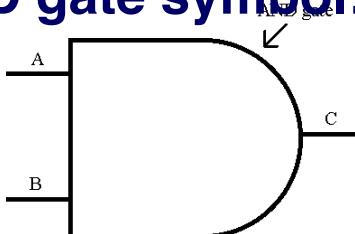
# Implementing a NAND operation

- **NAND = Negative AND**
  - Output =  $\sim(A \text{ AND } B)$
  - Symbol for a NAND gate is:

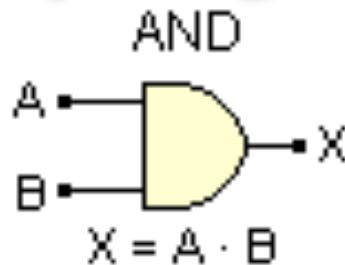


**How do I make an AND gate?**

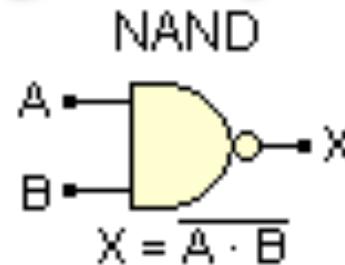
- Concatenate an inverter after a NAND gate!
- AND gate symbol:



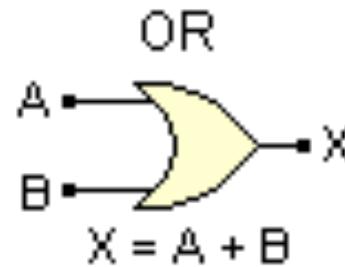
# Binary Digital Logic Symbols



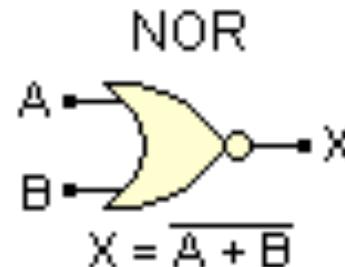
A	B	X
0	0	0
0	1	0
1	0	0
1	1	1



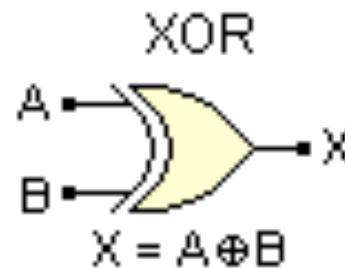
A	B	X
0	0	1
0	1	1
1	0	1
1	1	0



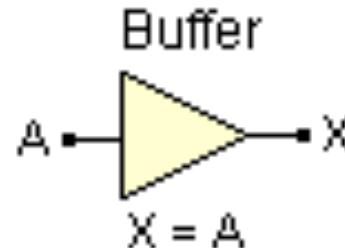
A	B	X
0	0	0
0	1	1
1	0	1
1	1	1



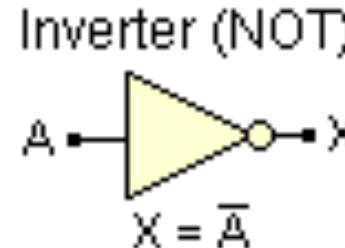
A	B	X
0	0	1
0	1	0
1	0	0
1	1	0



A	B	X
0	0	0
0	1	1
1	0	1
1	1	0



A	X
0	0
1	1



A	X
0	1
1	0

# Building a Binary Adder

## One-bit Half Adder circuit

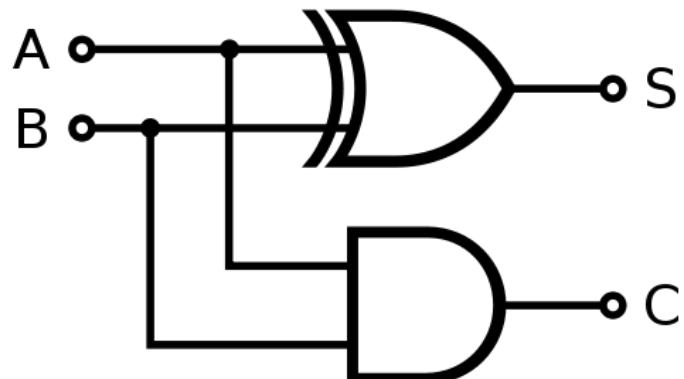
- Sum bit  $S = A \text{ XOR } B$
- Carry bit  $C = A \text{ AND } B$

- One-bit Full Adder circuit extends Half Adder to also account for a bit carried in, not just the bit carried out

- Example: an 8-bit Full Adder can be built by concatenating 8 one-bit Full Adders

Bit A	Bit B	Sum S	Carry C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

## One-bit Half Adder Circuit



# Binary Representation (2)

Base 2 vs. Decimal conversion example:

89 base 10 =  $89_{10}$  = what in base 2 or binary?

 Find the largest power of 2 that is  $\leq$  # and then subtract.  
Keep iterating until # = 0.

$$\begin{aligned} 89_{10} &= 64 + 16 + 8 + 1 \\ &= 2^6 + 2^4 + 2^3 + 2^0 \\ &= 0*2^7 + 1*2^6 + 0*2^5 + 1*2^4 + 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0 \\ &= 0\ 1\ 0\ 1\ 1\ 0\ 0\ 1_2 \end{aligned}$$

# Hexadecimal Representation

**Hexadecimal (base 16) is more convenient  
(it's power of 2) and compact notation  
for larger binary #'s**

- Consider a 32-bit number:

$10101110001100101000101000011001_2$

This is hard to read. So form 4-bit groups.

=  $1010\ 1110\ 0011\ 0010\ 1000\ 1010\ 0001\ 1001_2$

We then use hexadecimal notation:

Use characters '0' to '9' and 'A' to 'F'

= A E 3 2 8 A 1 9<sub>16</sub>

= 0xAE328A19

“0x” signifies hexadecimal or just hex.

Note lower case 0xae328a19 is also valid.

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

# How We Use These Representations

Since these are just numbers, we can use them to describe anything with numerical value

We'll use them to describe the numerical *values* of data stored at various memory locations

- Different data values, like real and integer values, positive and negative

We'll also use them to describe the *addresses* of memory locations

Later, our code instructions will be encoded in binary as well

# Byte-Addressed Memory

Byte = 8 bits

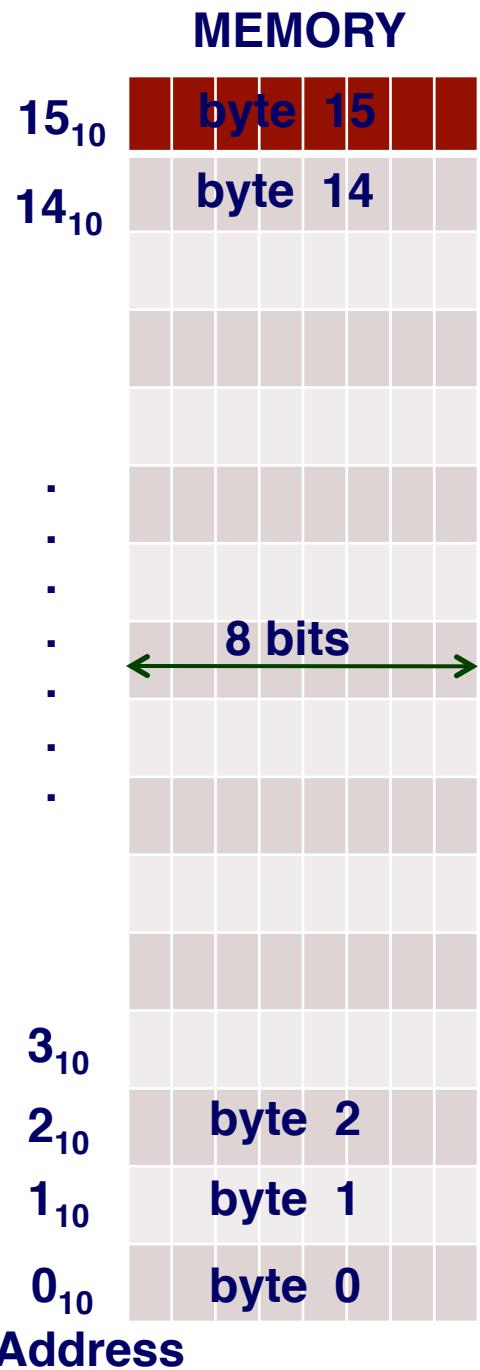
- Binary  $00000000_2$  to  $11111111_2$
- Decimal:  $0_{10}$  to  $255_{10}$
- Hexadecimal  $00_{16}$  to  $FF_{16}$
- Memory addresses are referenced by their *byte* number
  - Example: a 16 byte memory
  - Memory address  $15_{10}$ 
    - = 16<sup>th</sup> byte (since memory starts at 0)
    - = binary address

$000000000000000000000000000000001111_2$  in bytes

= hex address  $0x0000000F$  in bytes

This memory stores how many bits?

Answer:  $8 * 16 = 128$  bits!



# Machine Words

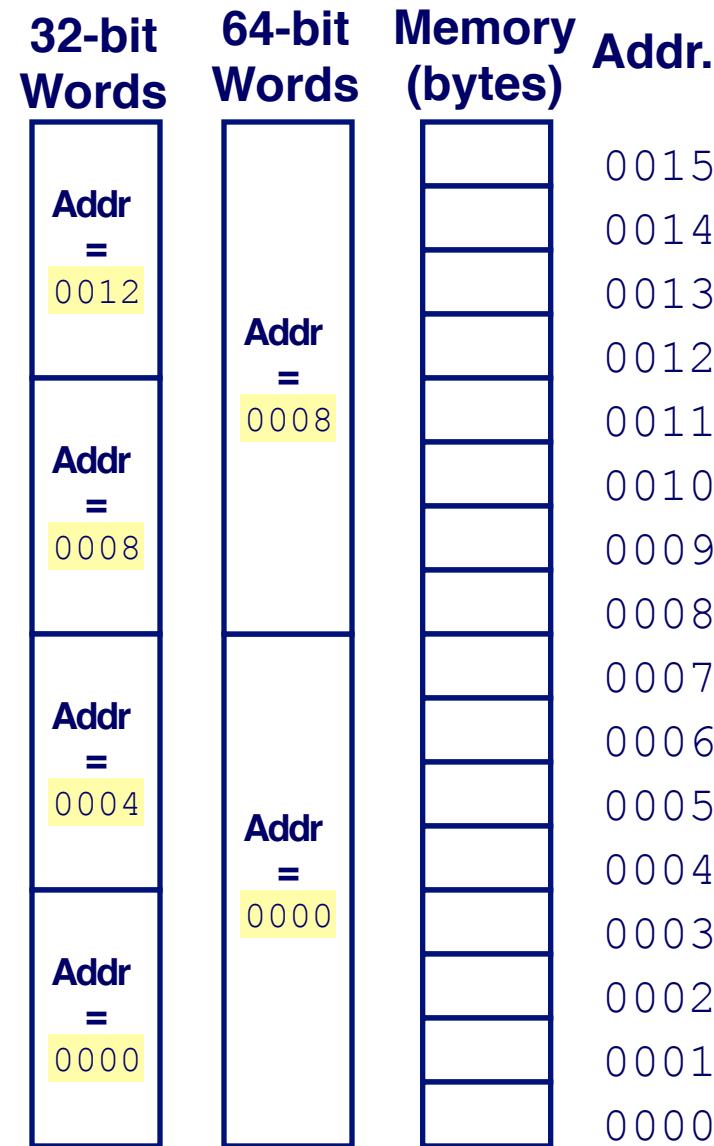
## Each CPU Has “Word Size”

- Nominal size of integer-valued data
  - Including addresses
- Early microprocessors were 8-bit CPUs. Today, many embedded microcontrollers are still 8-bit.
- Intel evolved to 16-bit processors, and called these “words”.
- Most machines were 32 bits (4 bytes)
  - Limits addresses to 4GB
  - Becoming too small for memory-intensive applications
- Most newer systems are 64 bits (8 bytes)
  - Potentially address  $\approx 1.8 \times 10^{19}$  bytes or 18 Exabytes
- Machines support multiple data formats
  - Fractions or multiples of word size
  - Always integral number of bytes

# Word-Oriented Memory Organization

## Addresses Specify Byte Locations

- Address of first byte in word
- Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



# Data Representations

## Sizes of C Objects (in Bytes)

■ C Data Type	Intel_x64	Typical 32-bit	Intel IA32
● int	4	4	4
● long int	8	4	4
● char	1	1	1
● short	2	2	2
● float	4	4	4
● double	8	8	8
● long double	10/16	8	10/12
● char *	8	4	4

» Or any other pointer

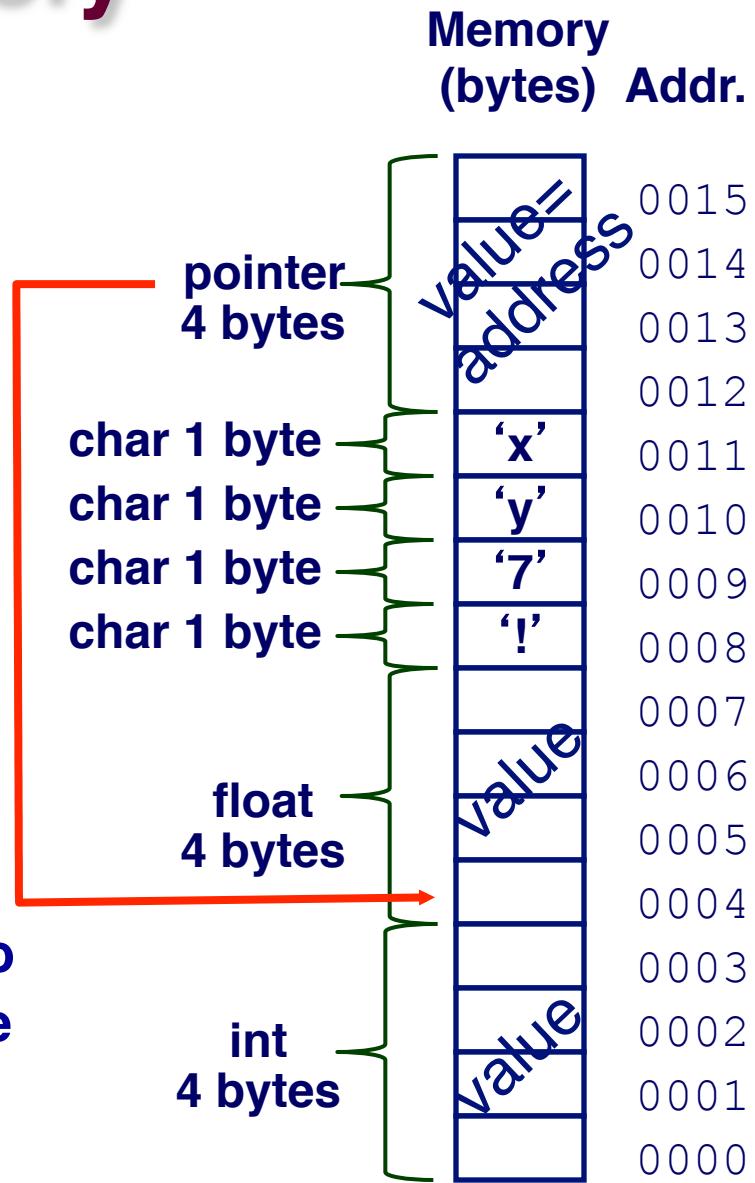
A float is a real # like  $-2.7143 \times 10^{-3}$ . A double just provides more precision than a float, something like  $2.71433273882 \times 10^{-3}$

A pointer stores an integer-like value that is interpreted as a memory location/address

# Storing Data in Memory

## IA32 Example:

- Address of int is 0x00000000
- Address of float is 0x00000004
- Address of character = '7' is 0x00000009
- Address of pointer is 0x0000000c
  - Note: the pointer points to another memory location, i.e. stores a memory location *address*  
e.g. if pointer = 0x00000004 it means the pointer is pointing to the float! (actually the first byte of the float)



# Pointers in C

To declare a pointer to say an integer, here's some code:

```
int x;  
int *p = &x;
```

**int \* p declares the variable p to be a pointer to an integer**

**note how it differs from the declaration ‘int p’ , which would just declare p to be an integer, not a pointer to an integer**

**&x means find the memory location/address of x**

**Thus, int \*p = &x means initialize the value of the pointer p with the address of integer variable x.**

# Dereferencing Pointers in C

To find the value of a variable being pointed to, you must dereference a pointer. Here's some code:

```
int x;  
int *p = &x;  
int y = *p;
```

p is a pointer to an integer. To get the value of the integer that p is pointing to, you use the dereference operator (\*).

note how \* is used first to declare a pointer, and is used later to dereference a pointer to find the value of the variable it is pointing to

Here, y = x.

# Pointer Values

Since the pointer value is an address, then it doesn't matter what type of variable is being pointed to, the size of the pointer value is always 4 bytes (IA32)

```
char c;  
int x;  
double d;  
char *p1 = &c;  
int *p2 = &x;  
double *p3 = &d;
```

p1 is a pointer to a one-byte long char, but is 4 bytes long, because the address of the char is 4 bytes long.

p3 is a pointer to an 8-byte long double, but is 4 bytes long, because the address of the double (lowest byte) is 4 bytes long.

# **Supplementary Slides**

# Integer Algebra vs. Boolean Algebra

## Integer Arithmetic

- $\langle \mathbb{Z}, +, *, -, 0, 1 \rangle$  forms a “ring”
- Addition is “sum” operation
- Multiplication is “product” operation
- $-$  is additive inverse
- 0 is identity for sum
- 1 is identity for product

## Boolean Algebra

- $\langle \{0,1\}, \text{I}, \&, \sim, 0, 1 \rangle$  forms a “Boolean algebra”
- OR is “sum” operation
- AND is “product” operation
- $\sim$  is “complement” operation (not additive inverse)
- 0 is identity for sum
- 1 is identity for product

# Boolean Algebra $\approx$ Integer Ring

## ■ Commutativity

$$A \mid B = B \mid A$$

$$A \& B = B \& A$$

$$A + B = B + A$$

$$A * B = B * A$$

## ■ Associativity

$$(A \mid B) \mid C = A \mid (B \mid C)$$

$$(A \& B) \& C = A \& (B \& C)$$

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

$$(A * B) * C = A * (B * C)$$

## ■ Product distributes over sum

$$A \& (B \mid C) = (A \& B) \mid (A \& C)$$

$$A * (B + C) = A * B + A * C$$

## ■ Sum and product identities

$$A \mid 0 = A$$

$$A \& 1 = A$$

$$A + 0 = A$$

$$A * 1 = A$$

## ■ Zero is product annihilator

$$A \& 0 = 0$$

$$A * 0 = 0$$

## ■ Cancellation of negation

$$\sim(\sim A) = A$$

$$-(-A) = A$$

# Boolean Algebra $\neq$ Integer Ring

## ■ Boolean: *Sum distributes over product*

$$A \mid (B \& C) = (A \mid B) \& (A \mid C) \quad A + (B * C) \neq (A + B) * (A + C)$$

## ■ Boolean: *Idempotency*

$$A \mid A = A$$

$$A + A \neq A$$

- “A is true” or “A is true” = “A is true”

$$A \& A = A$$

$$A * A \neq A$$

## ■ Boolean: *Absorption*

$$A \mid (A \& B) = A$$

$$A + (A * B) \neq A$$

- “A is true” or “A is true and B is true” = “A is true”

$$A \& (A \mid B) = A$$

$$A * (A + B) \neq A$$

## ■ Boolean: *Laws of Complements*

$$A \mid \sim A = 1$$

$$A + \sim A \neq 1$$

- “A is true” or “A is false”

## ■ Ring: *Every element has additive inverse*

$$A \mid \sim A \neq 0$$

$$A + \sim A = 0$$

## Boolean Ring

- $\langle \{0,1\}, \wedge, \&, I, 0, 1 \rangle$
- Identical to integers mod 2
- $I$  is identity operation:  $I(A) = A$   
 $A \wedge A = 0$

## Property

- Commutative sum
- Commutative product
- Associative sum
- Associative product
- Prod. over sum
- 0 is sum identity
- 1 is prod. identity
- 0 is product annihilator
- Additive inverse

## Properties of & and $\wedge$

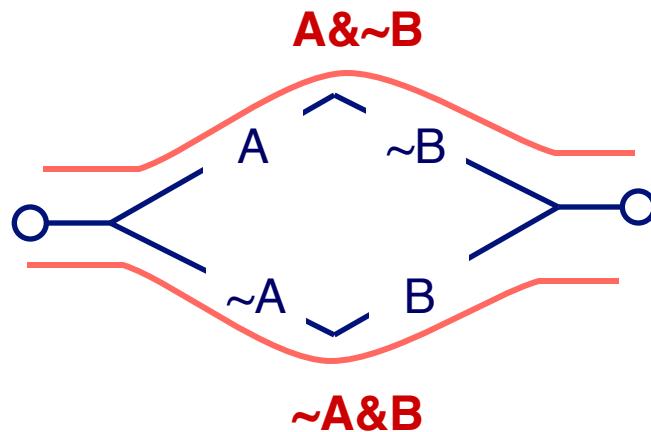
### Boolean Ring

- $A \wedge B = B \wedge A$
- $A \& B = B \& A$
- $(A \wedge B) \wedge C = A \wedge (B \wedge C)$
- $(A \& B) \& C = A \& (B \& C)$
- $A \& (B \wedge C) = (A \& B) \wedge (A \& C)$
- $A \wedge 0 = A$
- $A \& 1 = A$
- $A \& 0 = 0$
- $A \wedge A = 0$

# Application of Boolean Algebra

## Applied to Digital Systems by Claude Shannon

- 1937 MIT Master's Thesis
- Reason about networks of relay switches
  - Encode closed switch as 1, open switch as 0



Connection when

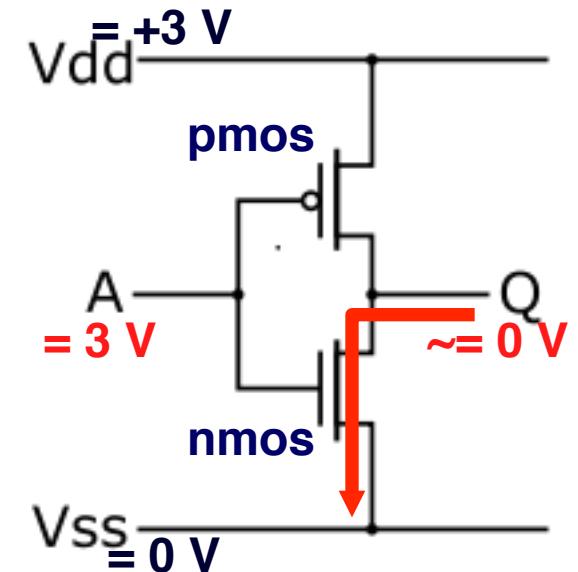
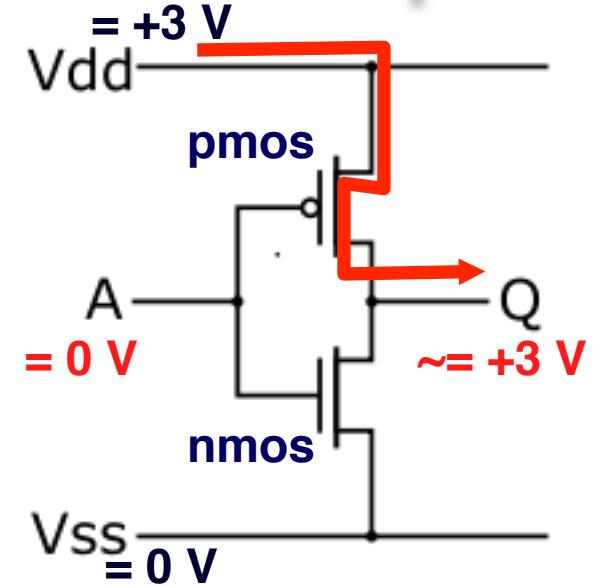
$$A \& \sim B \mid \sim A \& B$$

$$= A \wedge B$$

# Binary Digital Logic Circuit Example

- **CMOS inverter circuit**

- **Vdd = Power = +3 Volts, Vss = Ground = 0 Volts**
- **Current flows from high voltage to low voltage**
- **When input A = 0 V (Ground), then nmos transistor is shut off, while the pmos transistor is turned on, so Q is connected to Vdd = +3 V**
- **When input A = +3 V (Power), then pmos is off and nmos is on, so Q is connected to ground and is pulled down to 0 V**
- **So Q is the opposite or inverse of A!**  
i.e.  $Q = \sim A$
- **AND, OR, NAND, NOR, ADD, SUBTRACT, etc. are also implemented this way...**



# Byte-Oriented Memory Organization

## Programs Refer to Virtual Addresses

- Conceptually very large array of bytes
- Actually implemented with hierarchy of different memory types
  - SRAM, DRAM, disk
  - Only allocate for regions actually used by program
- Address space private to particular “process”
  - Program being executed
  - Program can clobber its own data, but not that of others

## Compiler + Run-Time System Control Allocation

- Where different program objects should be stored
- Multiple mechanisms: static, stack, and heap
- In any case, all allocation within single virtual address space