Chapter 2: Bits and Bytes

Topics

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

Announcements

Moodle is having difficulty emailing announcements

Data Lab is due Friday Feb 3 by 11:55 pm

- Bit manipulation operations

C assessment quiz due Friday Jan 20 by noon

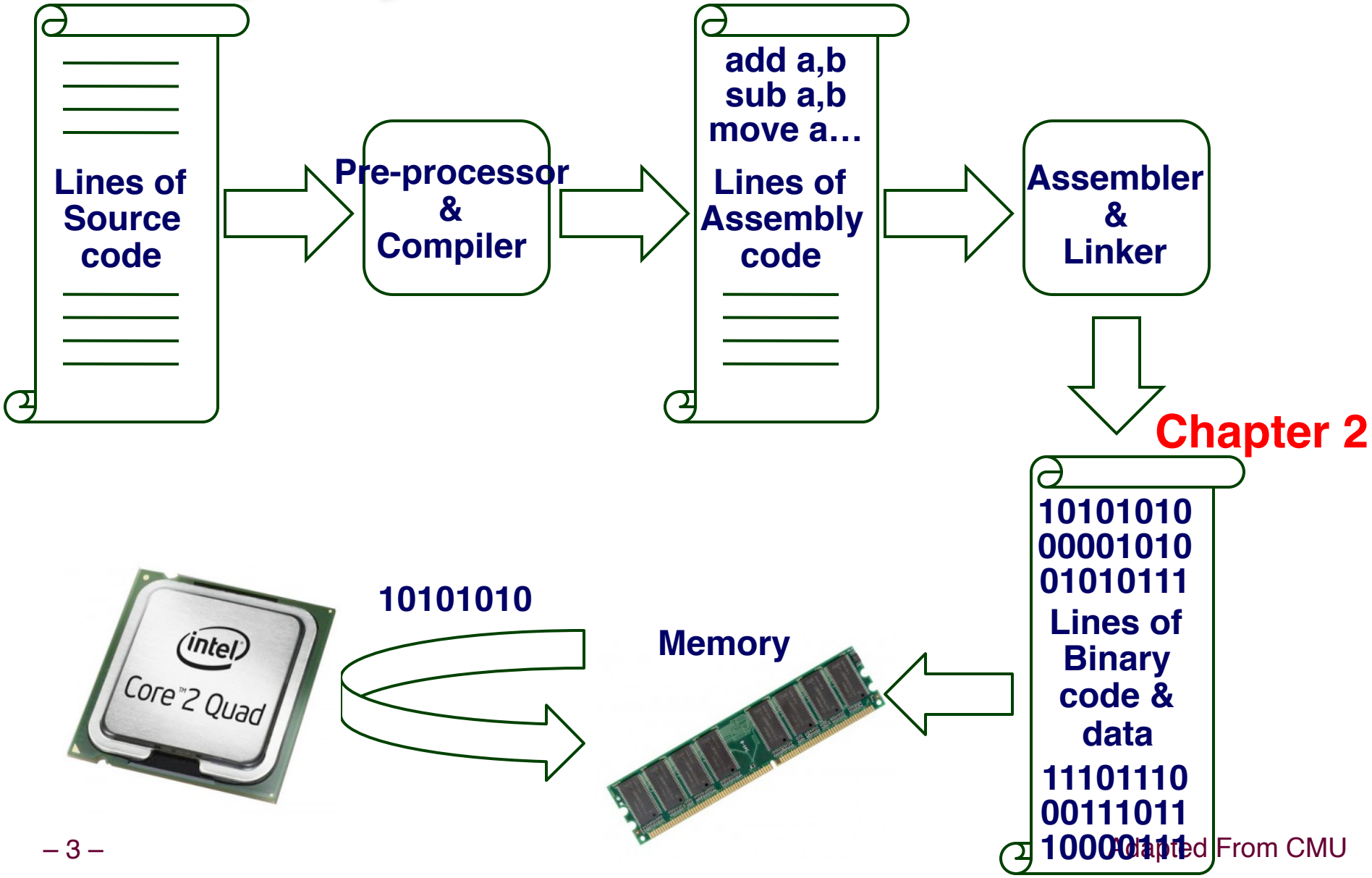
- Unlimited # of attempts
- If you score <=70%, then attend C tutorial Friday evening (TA Yogesh will announce)

Chap 2 Data Quiz is due Monday Jan 30 by noon

- Addition, subtraction, signed, overflow

Read Chapter 2.1-2.3 and do practice problems

Recap: Systems In a Nutshell



Adapted From CMU

Binary Representation – Why Bits?

$0\ 1\ 0\ 1_2 = 0*2^3 + 1*2^2 + 0*2^1 + 1*2^0$

$= 0 + 4 + 0 + 1 = 5_{10}$

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

Common Digital Logic Operations

And "&"

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

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

Or "|"

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

|	0	1
0	0	1
1	1	1

Not "~"

- ~A = 1 only when A=0

~	
0	1
1	0

Exclusive-Or (Xor) "^"

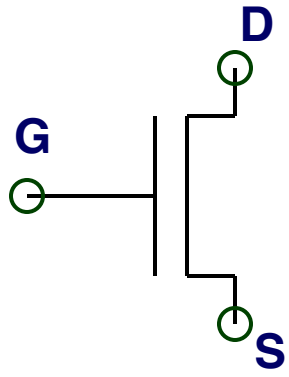
- A^B = 1 when either A=1 or B=1, but not both

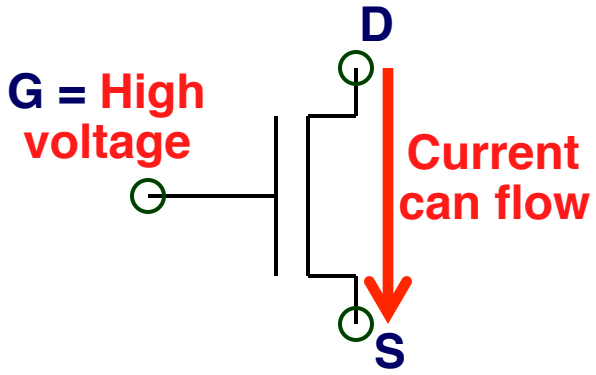
^	0	1
0	0	1
1	1	0

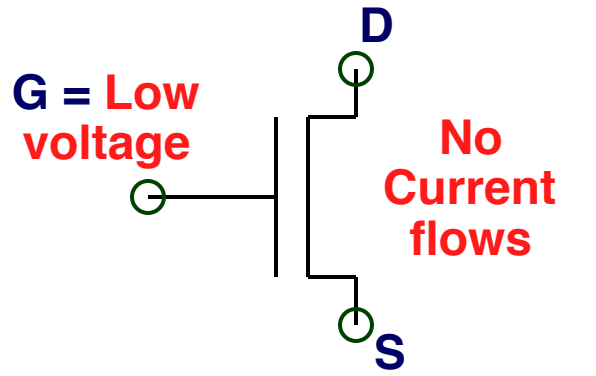
Other operators: NAND (Not AND), NOR (Not OR), etc.

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

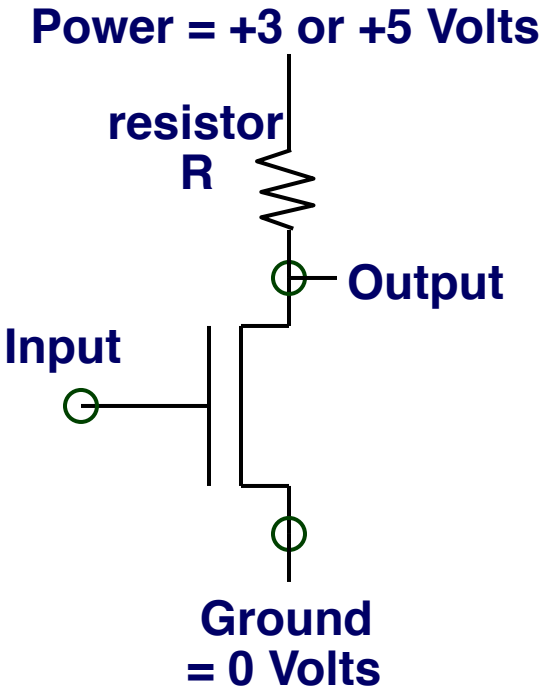




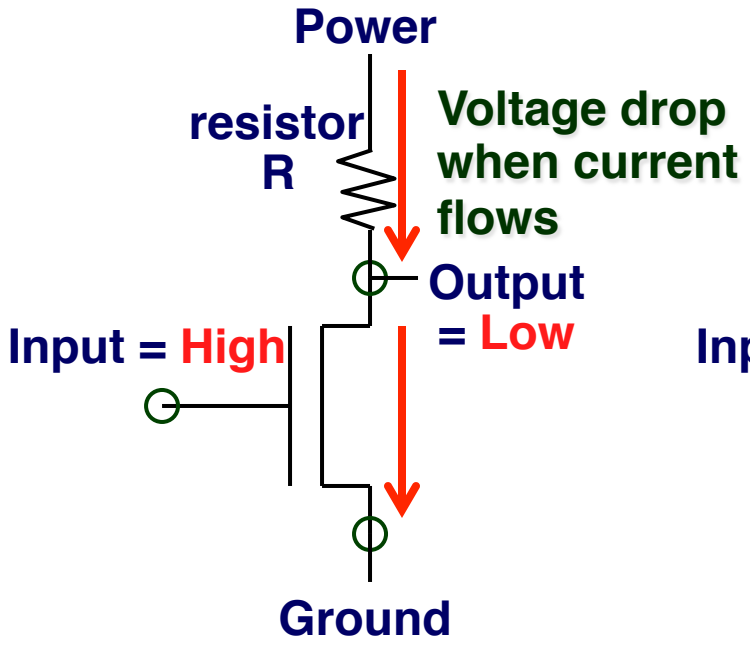


Implementing an Inverter or Not "~" Operation

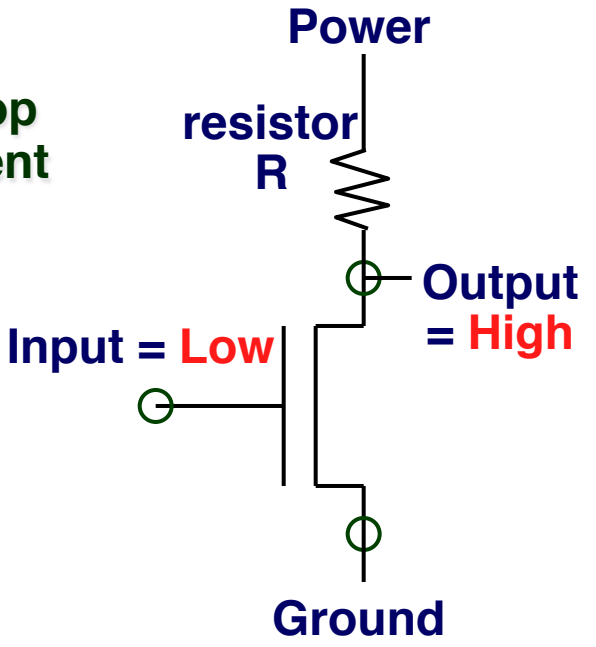
An Inverter circuit



High input connects Output to Ground



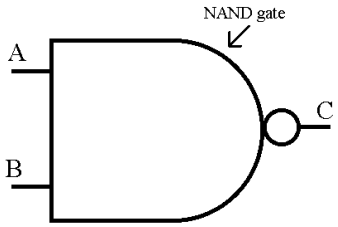
Low input means no current flows, so output stay high



- Thus we have an inverter or Not "~" operator:
  - Input '1' (High voltage) => output '0' (Low voltage)
  - Input '0' (Low voltage) => output '1' (High voltage)

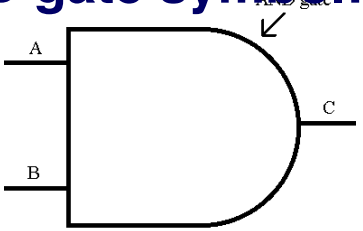
Implementing a NAND operation

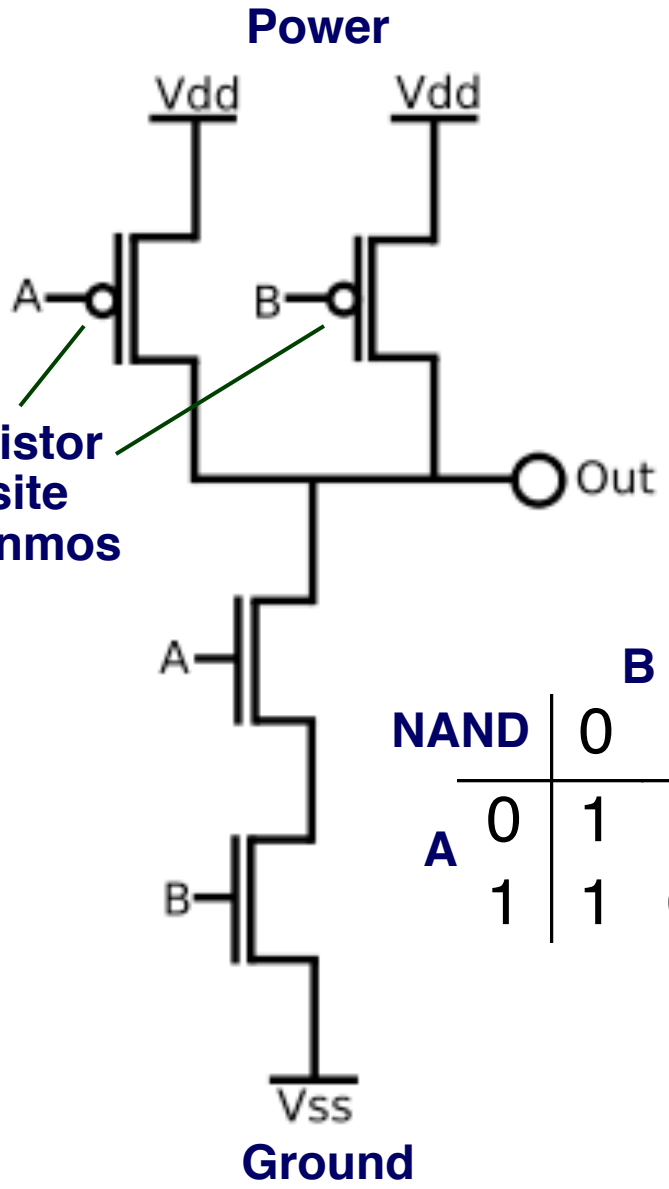
- **NAND = Negative AND**
  - **Output = ~(A AND B)**
  - **Symbol for a NAND gate is:**



How do I make an AND gate?

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

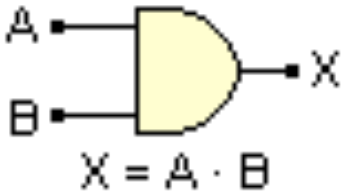




NAND	B = 0	B = 1
A = 0	1	1
A = 1	1	0

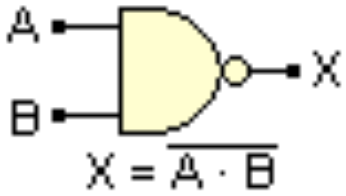
Binary Digital Logic Symbols

AND



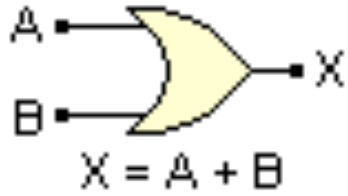
A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

NAND



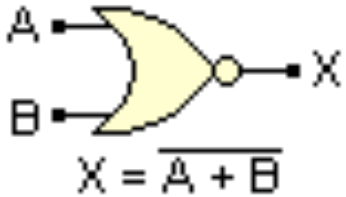
A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

OR



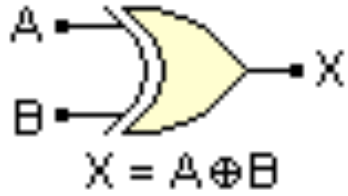
A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

NOR



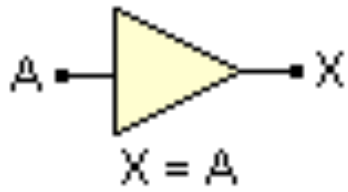
A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

XOR



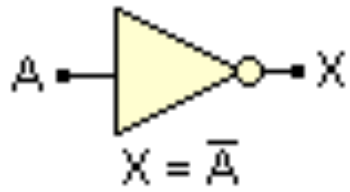
A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

Buffer



A	X
0	0
1	1

Inverter (NOT)



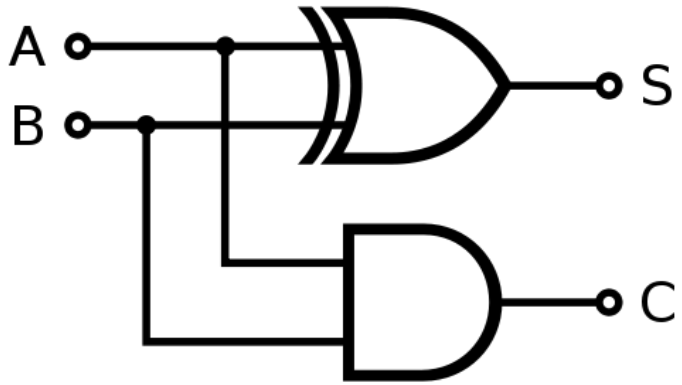
A	X
0	1
1	0

Building a Binary Adder

- One-bit Half Adder circuit
  - Sum bit S = A XOR B
  - Carry bit C = A 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 <= # and then subtract.
Keep iterating until #=0.

$89_{10} = 64 + 16 + 8 + 1$

$= 2^6 + 2^4 + 2^3 + 2^0$

$= 0*2^7 + 1*2^6 + 0*2^5 + 1*2^4 + 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0$

$= 0\ 1\ 0\ 1\ 1\ 0\ 0\ 1_{\ 2}$

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:**

  $1010111000110010100010100001100 1_2$

  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'

  $= \mathrm{A}\ \mathrm{E}\ 3\ 2\ 8\ \mathrm{A}\ 1\ 9_{16}$

  $=$ 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

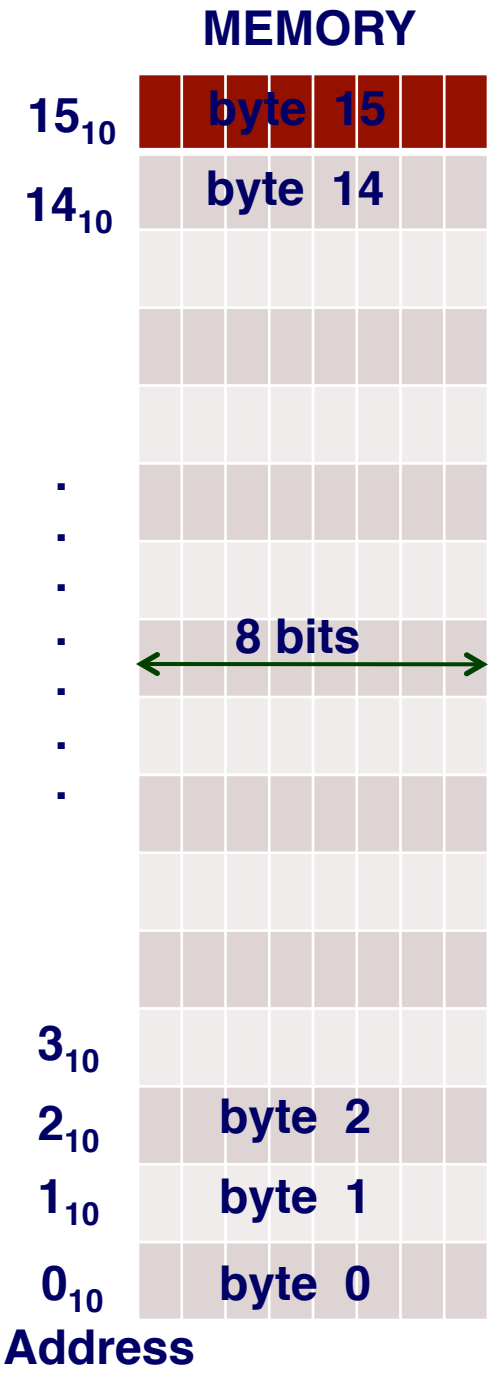
Byte-Addressed Memory

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^{th} byte (since memory starts at 0)
    = binary address $0000000000000000000000000001111_2$ 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".
  - `addw` and `movw` are assembly instructions where the "`w`" suffix stands for a 16-bit word and implies that 16-bit values are being added or moved
- Most machines became 32 bits (4 bytes)
  - `addl` and `movl` are assembly instructions where the "`l`" suffix stands for a "long word" and implies that 32-bit values are being added or moved
  - Limits addresses to 4GB
  - Becoming too small for memory-intensive applications

Machine Words

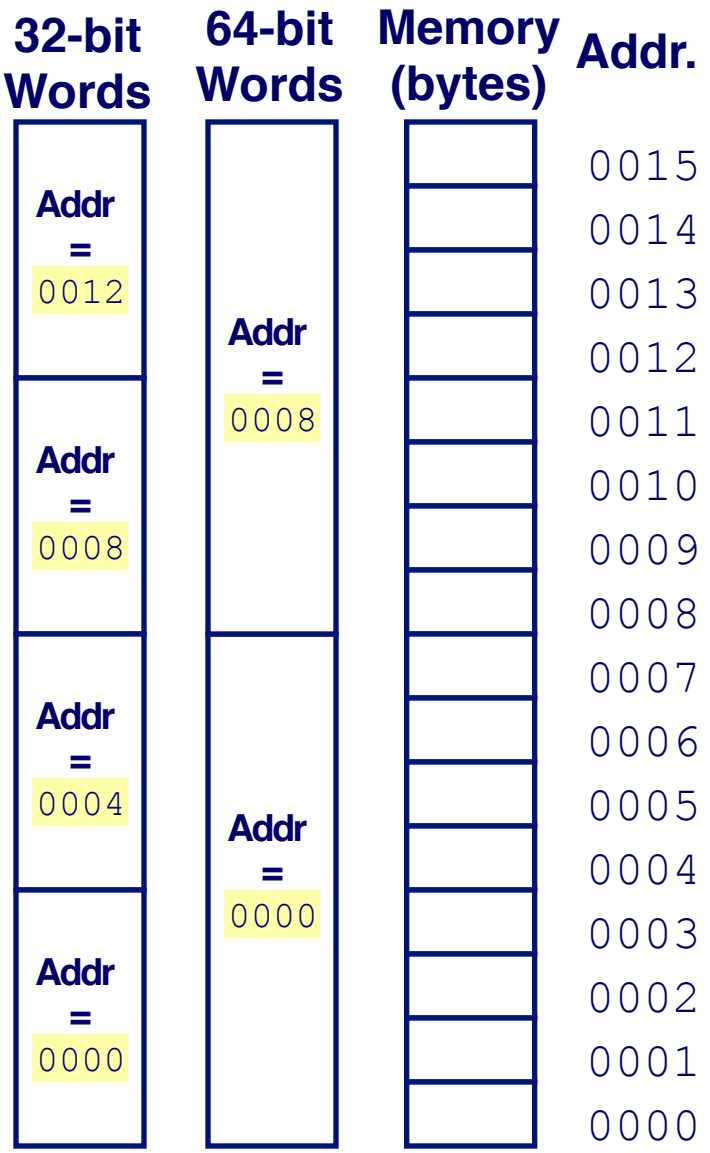
Each CPU Has "Word Size"

- Most newer systems are 64 bits (8 bytes)
  - Potentially address ≈ 1.8 X 10^{19} bytes or 18 *Exa*bytes
  - `addq` and `movq` are assembly instructions where the "`q`" suffix stands for "quad word" (4*16=64) and implies that 64-bit values are being added or moved
- 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 $-2.7143*10^{(-3)}$. A double just provides more precision than a float, something like $2.71433273882*10^{(-3)}$

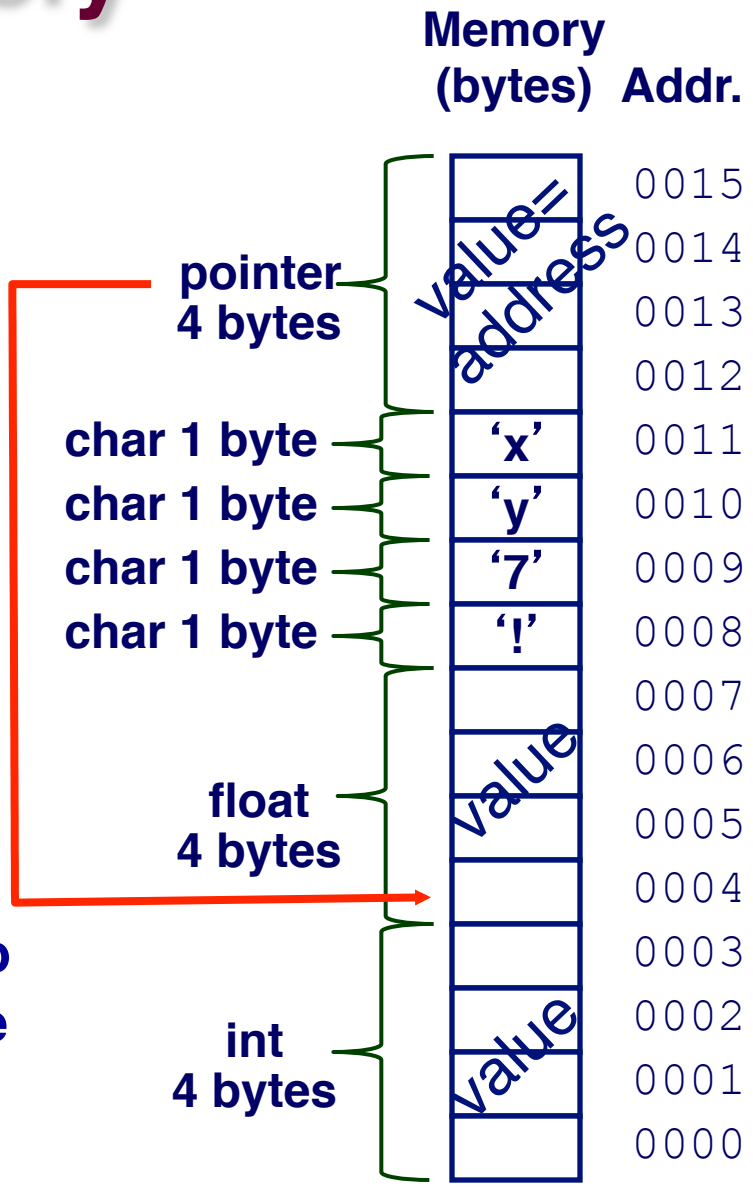
A pointer stores an integer–like value that is interpreted as a memory location/address

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.

Relations Between Logic Operations

DeMorgan's Laws

- Express & in terms of |, and vice-versa
  - A & B = ~(~A | ~B)
    - » A and B are true if and only if neither A nor B is false
  - A | B = ~(~A & ~B)
    - » A or B are true if and only if A and B are not both false

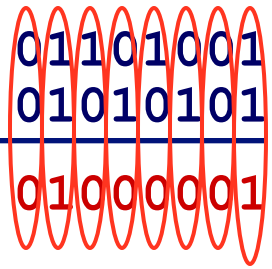
Exclusive-Or using Inclusive Or

- A ^ B = (~A & B) | (A & ~B)
  - » Exactly one of A and B is true
  - » This is Shannon's circuit.
- A ^ B = (A | B) & ~(A & B)
  - » Either A is true, or B is true, but not both

General Boolean Algebras

Operate on Bit Vectors

- Operations applied bitwise



```
  01101001          01101001          01101001
& 01010101        | 01010101        ^ 01010101        ~ 01010101
----------        ----------        ----------        ----------
  01000001          01111101          00111100          10101010
```

All of the Properties of Boolean Algebra Apply

Using Boolean Operators for Representing & Manipulating Sets

Representation

- **Width w bit vector represents subsets of** $\{0, \ldots, w-1\}$
- $a_j = 1$ **if** $j \in A$

```
01101001                    { 0, 3, 5, 6 }
76543210

01010101                    { 0, 2, 4, 6 }
76543210
```

Operations

■	**&**	**Intersection**	`01000001` { 0, 6 }
■	**|**	**Union**	`01111101` { 0, 2, 3, 4, 5, 6 }
■	**^**	**Symmetric difference**	`00111100` { 2, 3, 4, 5 }
■	**~**	**Complement**	`10101010` { 1, 3, 5, 7 }

Bit-Level Operations in C

Operations &, |, ~, ^ Available in C

- Apply to any "integral" data type
  - `long, int, short, char`
- View arguments as bit vectors
- Arguments applied bit-wise

- **Examples (Char data type)**
  - `~0x41 --> 0xBE`

    $\sim 01000001_2 \rightarrow 10111110_2$
  - `~0x00 --> 0xFF`

    $\sim 00000000_2 \rightarrow 11111111_2$
  - `0x69 & 0x55 --> 0x41`

    $01101001_2 \;\&\; 01010101_2 \rightarrow 01000001_2$
  - `0x69 | 0x55 --> 0x7D`

    $01101001_2 \mid 01010101_2 \rightarrow 01111101_2$

Supplementary Slides

Integer Algebra vs. Boolean Algebra

Integer Arithmetic

- ⟨Z, +, *, –, 0, 1⟩ **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

- ⟨{0,1}, |, &, ~, 0, 1⟩ **forms a "Boolean algebra"**
- **OR is "sum" operation**
- **AND is "product" operation**
- **~ is "complement" operation (not additive inverse)**
- **0 is identity for sum**
- **1 is identity for product**

Boolean Algebra ≈ Integer Ring

- ***Commutativity***

  $A \mid B = B \mid A$ $\qquad$ $A + B = B + A$

  $A \,\&\, B = B \,\&\, A$ $\qquad$ $A * B = B * A$

- ***Associativity***

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

  $(A \,\&\, B) \,\&\, C = A \,\&\, (B \,\&\, C)$ $\qquad$ $(A * B) * C = A * (B * C)$

- ***Product distributes over sum***

  $A \,\&\, (B \mid C) = (A \,\&\, B) \mid (A \,\&\, C)$ $\qquad$ $A * (B + C) = A * B + A * C$

- ***Sum and product identities***

  $A \mid 0 = A$ $\qquad$ $A + 0 = A$

  $A \,\&\, 1 = A$ $\qquad$ $A * 1 = A$

- ***Zero is product annihilator***

  $A \,\&\, 0 = 0$ $\qquad$ $A * 0 = 0$

- ***Cancellation of negation***

  $\sim (\sim A) = A$ $\qquad$ $-(-A) = A$

Boolean Algebra ≠ Integer Ring

- Boolean: *Sum distributes over product*

  A | (B & C) = (A | B) & (A | C) A + (B * C) ≠ (A + B) * (A + C)

- Boolean: *Idempotency*

  A | A = A A + A ≠ A

  - "A is true" or "A is true" = "A is true"

  A & A = A A * A ≠ A

- Boolean: *Absorption*

  A | (A & B) = A A + (A * B) ≠ A

  "A is true" or "A is true and B is true" = "A is true"

  A & (A | B) = A A * (A + B) ≠ A

- Boolean: *Laws of Complements*

  A | ~A = 1 A + –A ≠ 1

  - "A is true" or "A is false"

- Ring: *Every element has additive inverse*

  A | ~A ≠ 0 A + –A = 0

Properties of & and ^

Boolean Ring

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

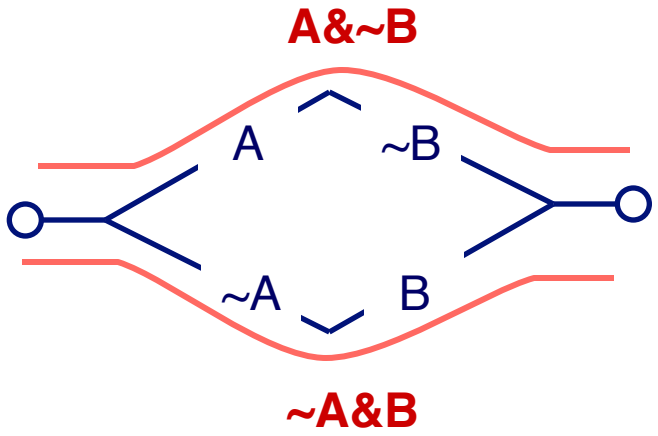
  A ^ A = 0

Property	Boolean Ring
Commutative sum	A ^ B = B ^ A
Commutative product	A & B = B & A
Associative sum	(A ^ B) ^ C = A ^ (B ^ C)
Associative product	(A & B) & C = A & (B & C)
Prod. over sum	A & (B ^ C) = (A & B) ^ (A & C)
0 is sum identity	A ^ 0 = A
1 is prod. identity	A & 1 = A
0 is product annihilator	A & 0 = 0
Additive inverse	A ^ 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

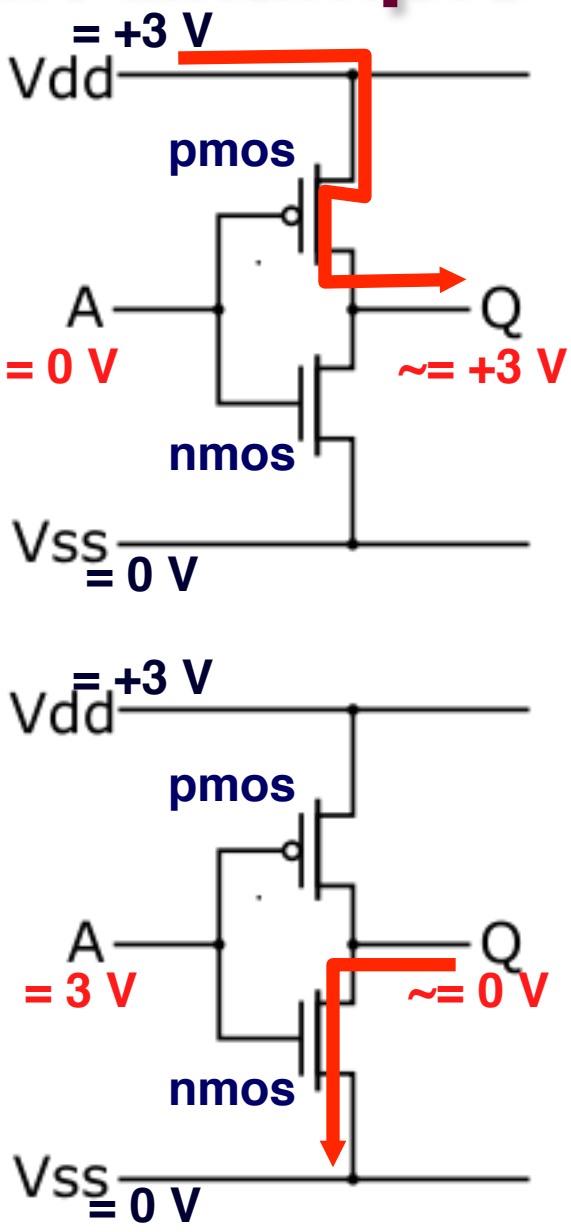
A&~B I ~A&B

= A^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 = ~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