CSC 252: Computer Organization
Spring 2026: Lecture 2

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Announcement

- Make sure you can access CSUG machines!!!
- Programming assignment 1 will be posted this week.
    - I will send an announcement when it's out.
    - It is in C language. Seek help from TAs.
    - TAs are best positioned to answer your questions about programming assignments!!!
- Programming assignments do NOT repeat the lecture materials. They ask you to synthesize what you have learned from the lectures and work out something new.

Previously in 252…

Problem

Algorithm

- How is a human-readable program translated to a repre
com
unde

- How
mod
exec
prog

	Renting
Service provider	Landlord
Service receiver	YOU
Contract	Lease
Contract's language	Natural language (e.g., English)

:t
e and

Circuit

Previously in 252…

C Program

```
void add() {
  int a = 1;
  int b = 2;
  int c = a + b;
}
```

→

Assembly program

```
movl    $1, -4(%rbp)
movl    $2, -8(%rbp)
movl    -4(%rbp), %eax
addl    -8(%rbp), %eax
```

Previously in 252…

Assembly program → *Executable Binary*

```
movl   $1, -4(%rbp)
movl   $2, -8(%rbp)
movl   -4(%rbp), %eax
addl   -8(%rbp), %eax
```

```
00011001   …
01101010   …
11010101   …
01110001   …
```

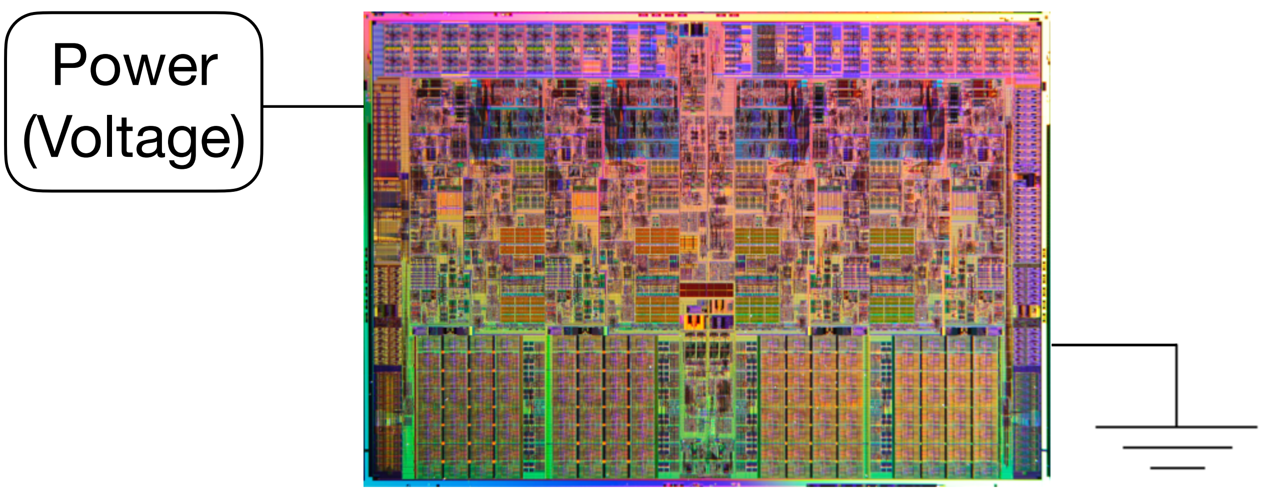
- What's the difference between an assembly program and an executable binary?
  - They refer to the same thing — a list of instructions that the software asks the hardware to perform
  - They are just different representations
- Instruction = Operator + Operand(s)

Today: Representing Information in Binary

- Why Binary (bits)?
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting

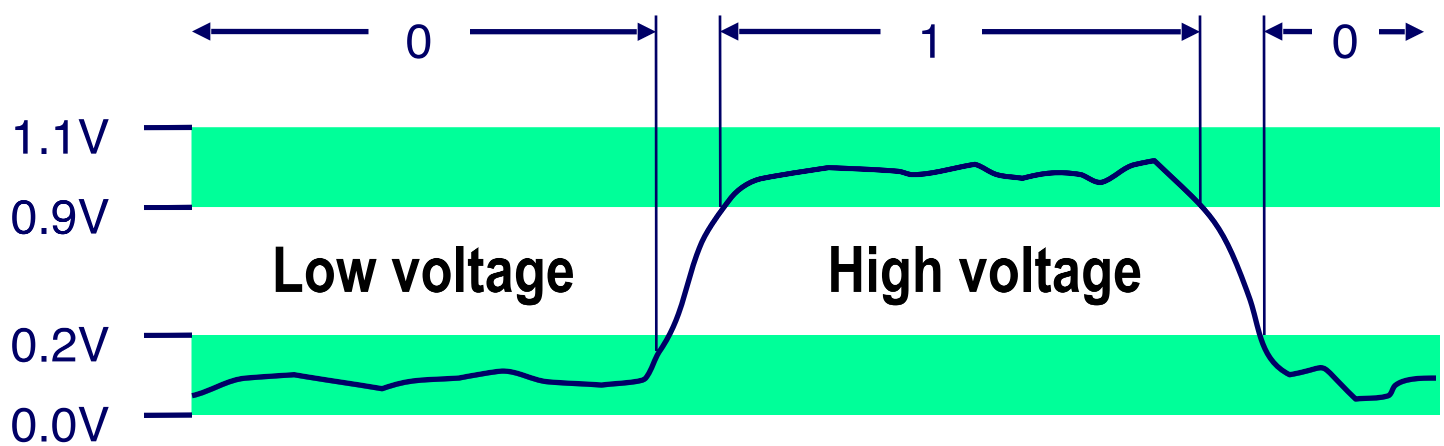
Everything in Computers is bits

- Each bit is 0 or 1. Bits are how programs talk to the hardware
- Programs encode instructions in bits
- Hardware then interprets the bits
- Why bits? Electronic Implementation



Everything in Computers is bits

- Each bit is 0 or 1. Bits are how programs talk to the hardware
- Programs encode instructions in bits
- Hardware then interprets the bits
- Why bits? Electronic Implementation
  - Use high voltage to represent 1
  - Use low voltage to represent 0



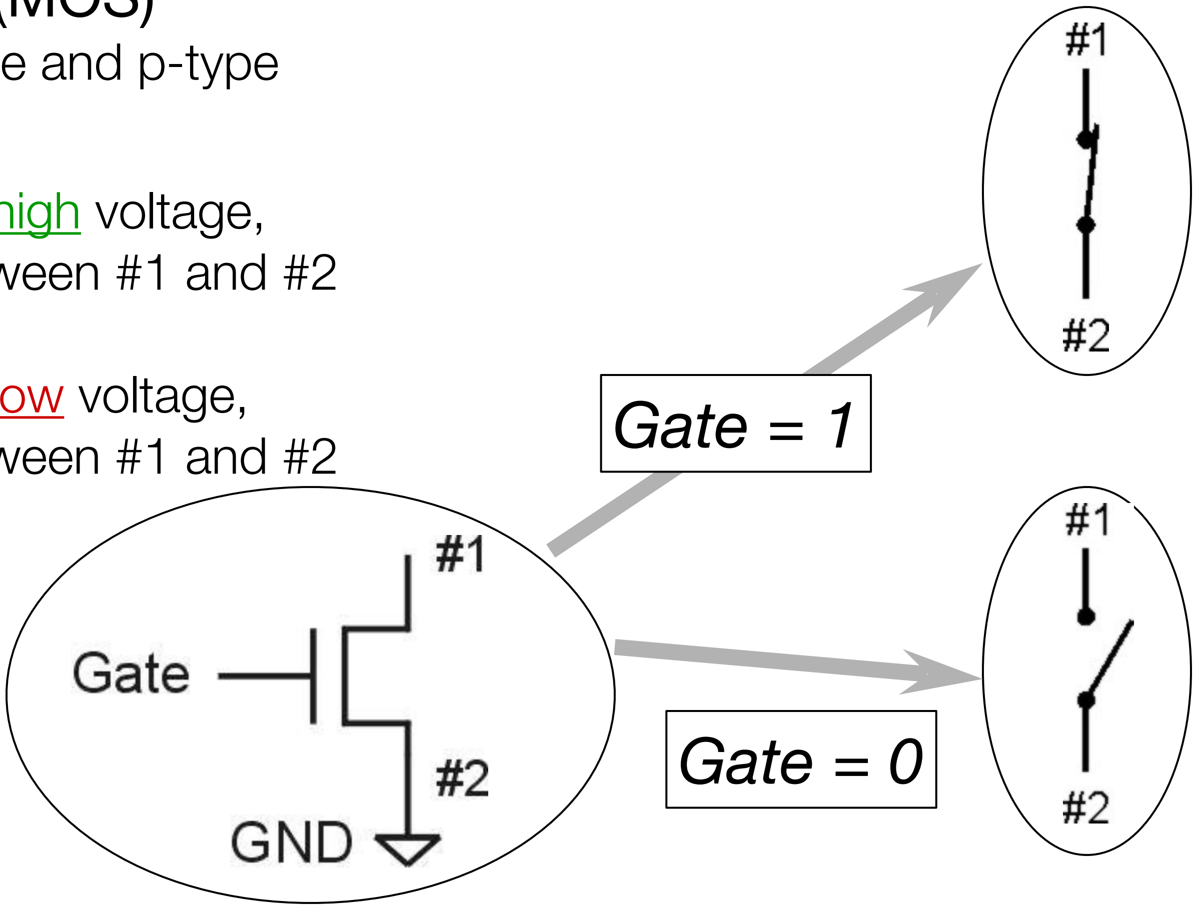
Why Bits?

Processors are made of transistors, which are Metal Oxide Semiconductor (MOS)

- two types: n-type and p-type

n-type (NMOS)

- when Gate has high voltage, short circuit between #1 and #2 (switch closed)
- when Gate has low voltage, open circuit between #1 and #2 (switch open)

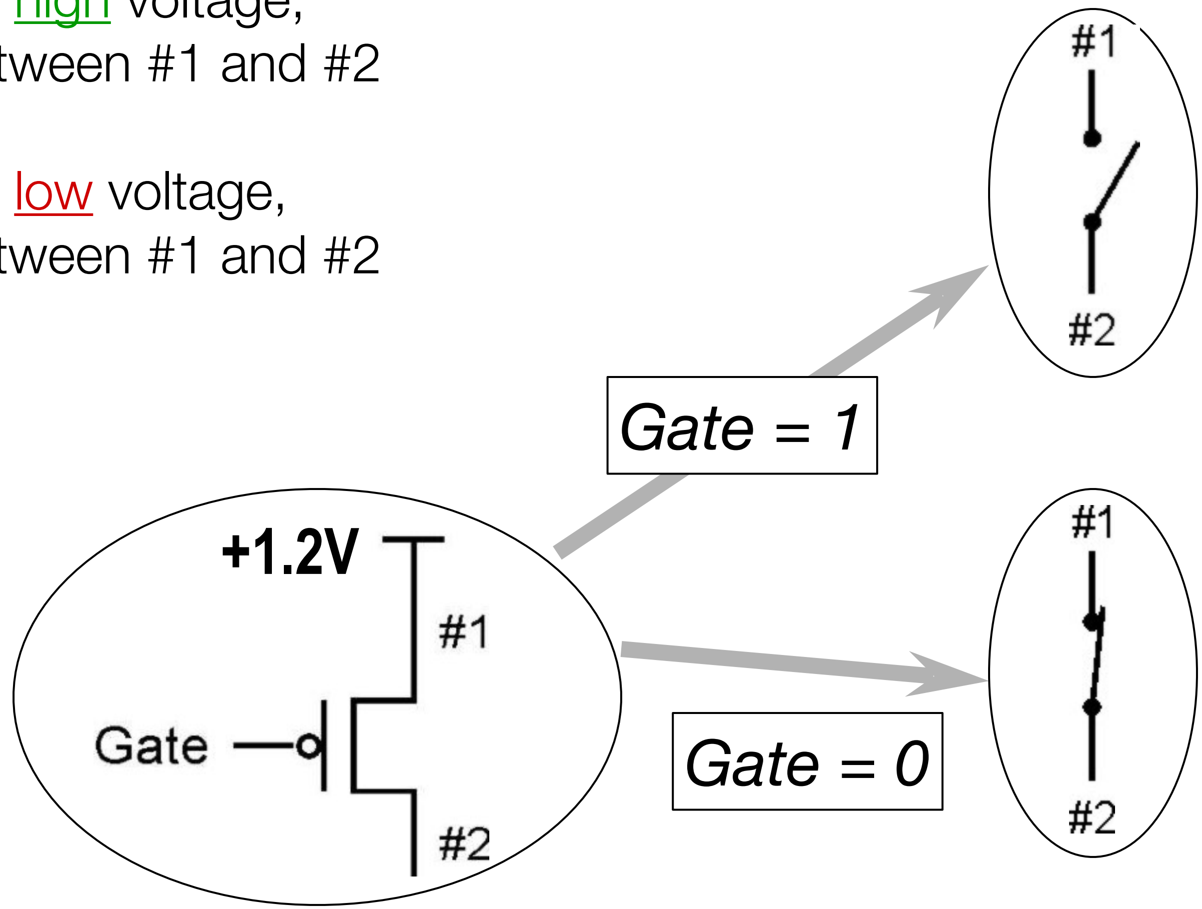


Terminal #2 must be connected to GND (0V).

Why Bits?

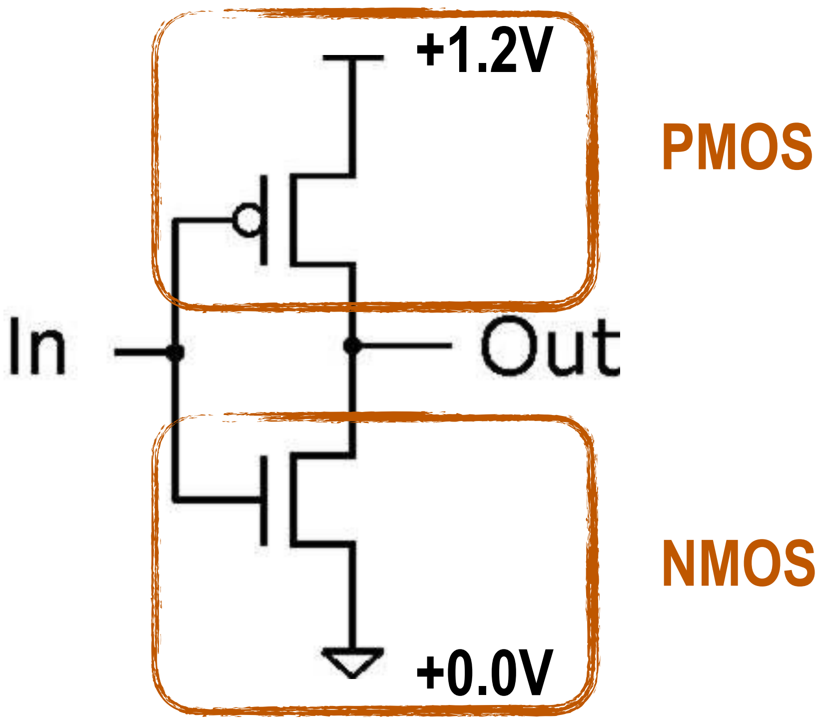
p-type is *complementary* to n-type (PMOS)

- when Gate has high voltage, open circuit between #1 and #2 (switch open)
- when Gate has low voltage, short circuit between #1 and #2 (switch closed)

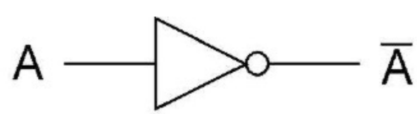


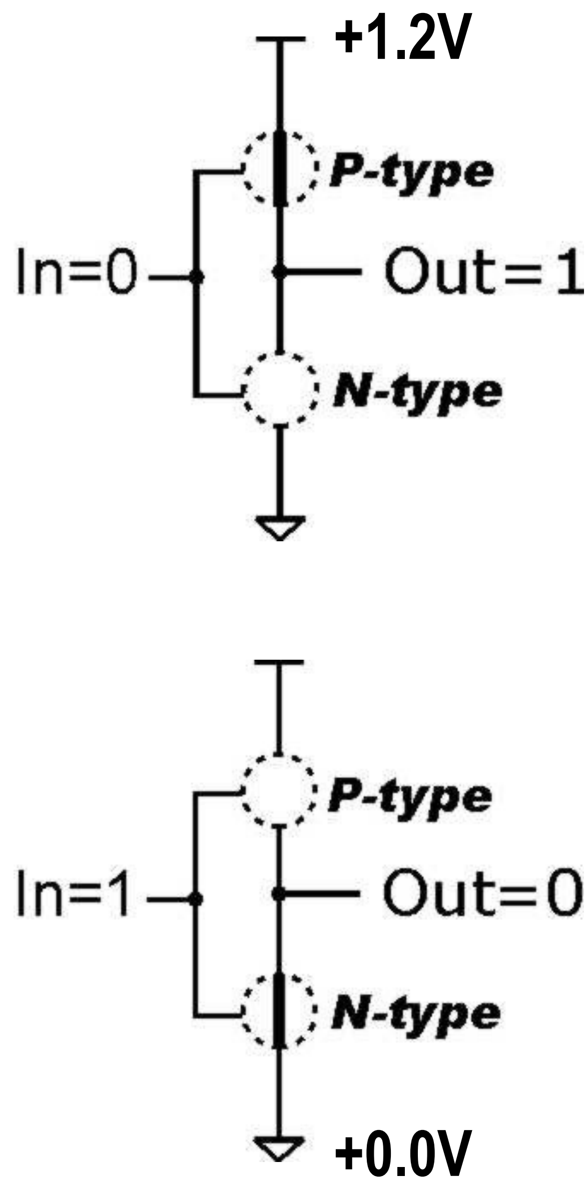
Terminal #1 must be connected to +1.2V

Inverter

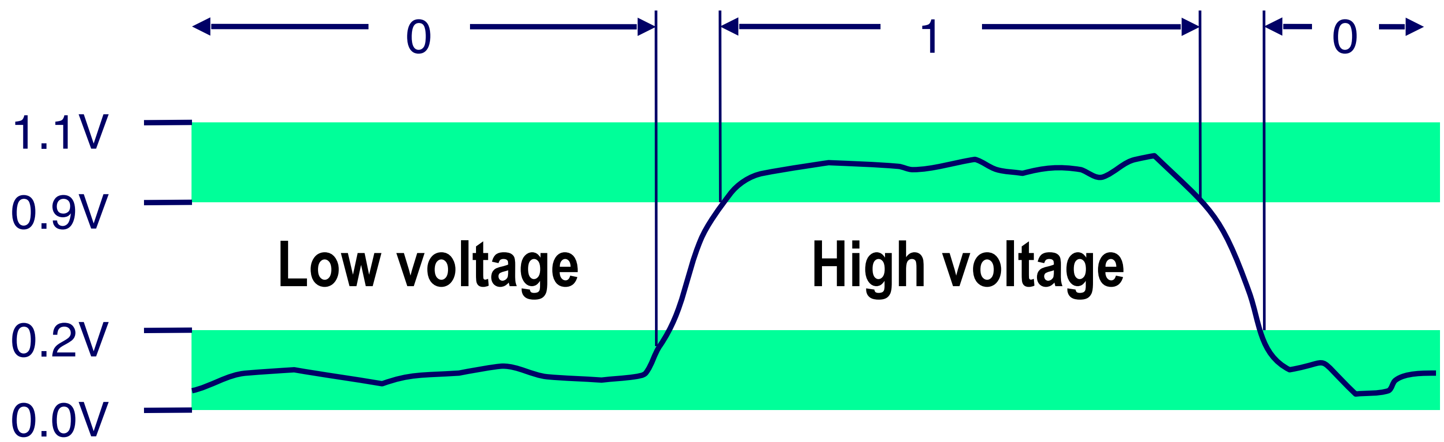


In	Out
0	1
1	0

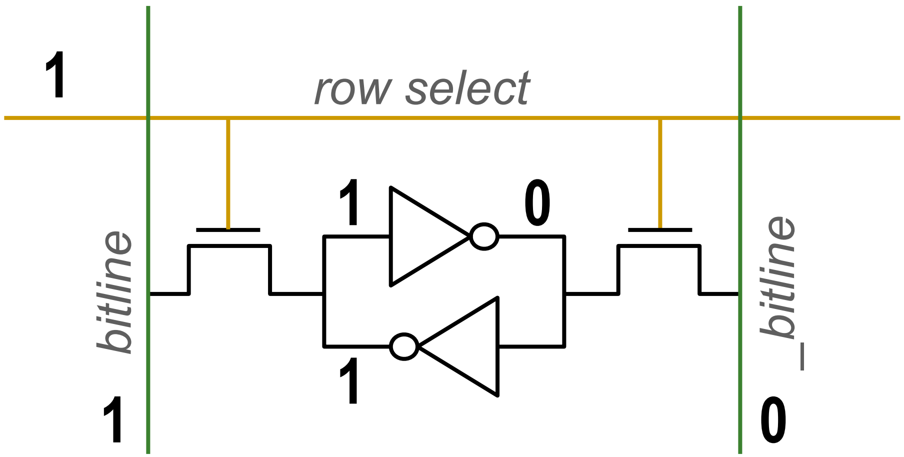




Store/Access Data

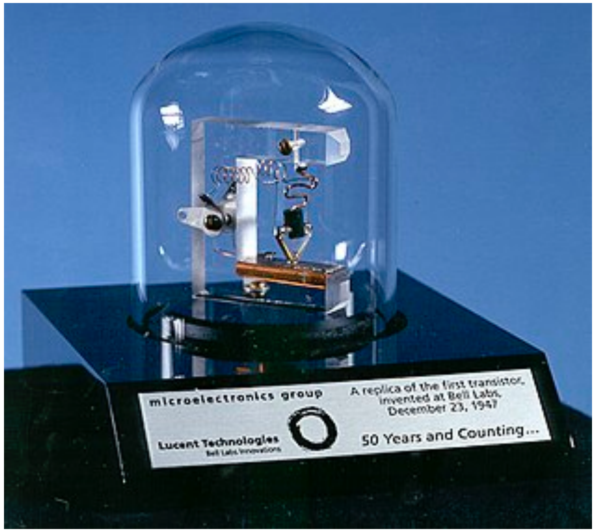


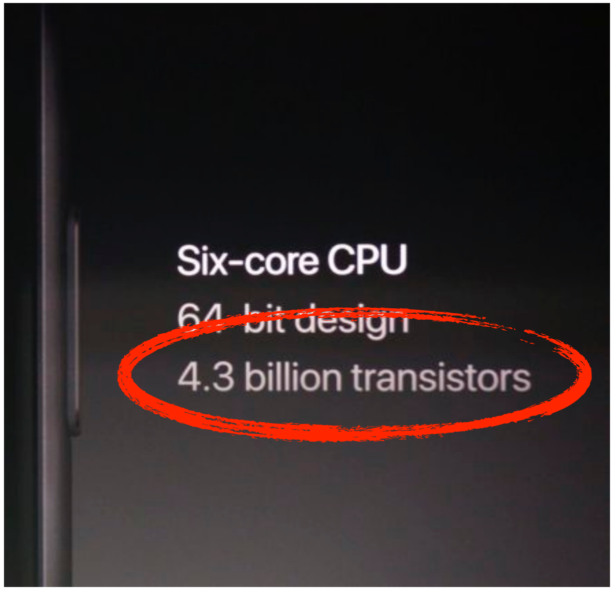
- Two cross coupled inverters store a single bit
  - Feedback path persists the value in the “cell”
  - 4 transistors for storage
  - 2 transistors for access
  - A “6T” cell



Transistors

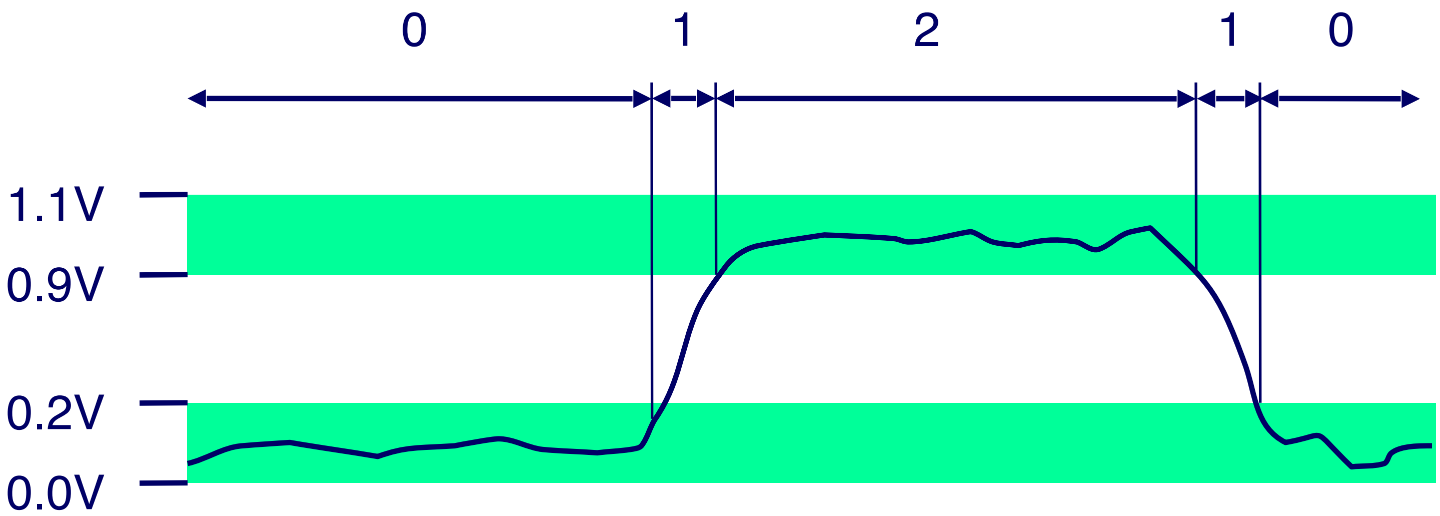
- Computers are made of transistors
- Transistors have become smaller over the years
  - Not so much anymore…





Why Limit Ourselves Only to Binary?

- Voltage is continuous. Why interpret it only as 0s and 1s?
- Answer: Noise



Binary Notation

- Base 2 Number Representation (Binary)
- C.f., Base 10 number representation (Decimal)
- $21_{10} = 1*10^0 + 2*10^1 = 21$
- Weighted Positional Notation
  - Each bit has a weight depending on its position
- $1011_2 = 1*2^0 + 1*2^1 + 0*2^2 + 1*2^3 = 11_{10}$
- $b_3b_2b_1b_0 = b^0*2^0 + b^1*2^1 + b^2*2^2 + b^3*2^3$
- Binary Arithmetic

```
  0110
+ 0101
------
  1011
```

```
  6
+ 5
---
 11
```

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

Hexdecimal (Hex) Notation

- Base 16 Number Representation
  - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
  - Four bits per Hex digit
  - $11111110_2 = FE_{16}$
- Write $FA1D37B_{16}$ in C as
  - 0xFA1D37B
  - 0xfa1d37b

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

Bit, Byte, Word

- Byte = 8 bits
  - Binary 00000000_2 to 11111111_2; Decimal: 0_{10} to 255_{10}; Hex: 00_{16} to FF_{16}
  - Least Significant Bit (LSb) vs. Most Significant Bit (MSb)

$$\boxed{1}011011\boxed{1}$$

MSb **LSb**

- Word = 4 Bytes (32-bit machine) / 8 Bytes (64-bit machine)
  - Least Significant Byte (LSB) vs. Most Significant Byte (MSB)

Today: Representing Information in Binary

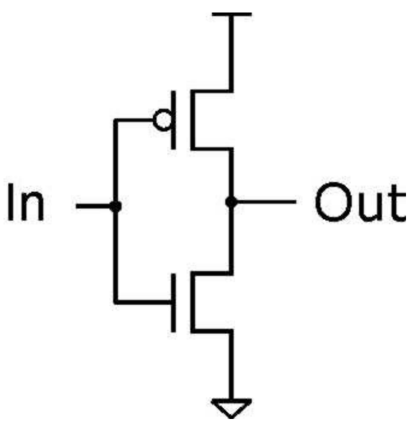
- Why Binary (bits)?
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting

Bit-level manipulations

Not

- ~A = 1 when A=0

~	
0	1
1	0



Or

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

|	0	1
0	0	1
1	1	1

And

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

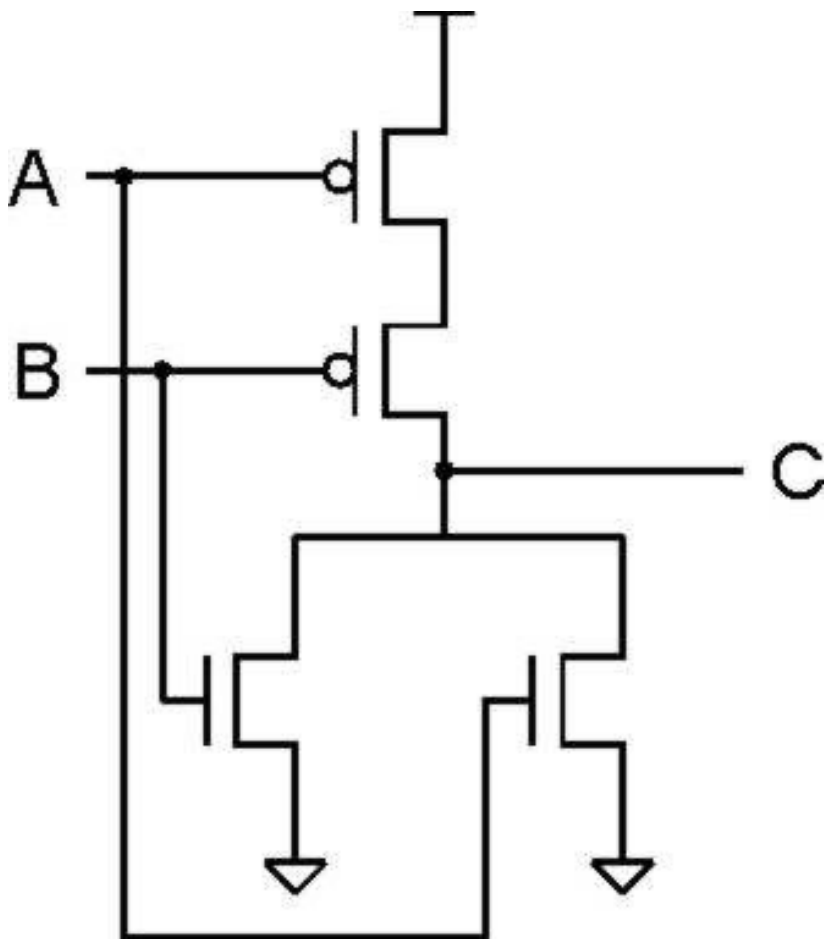
&	0	1
0	0	0
1	0	1

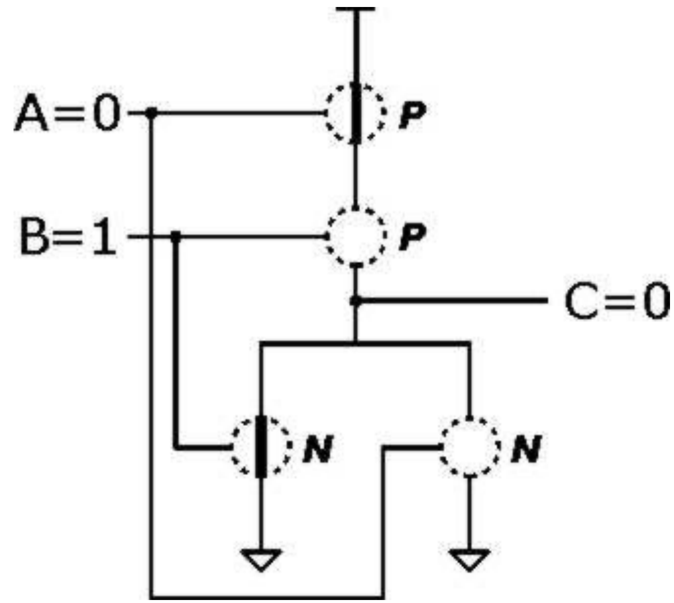
Exclusive-Or (Xor)

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

^	0	1
0	0	1
1	1	0

NOR (OR + NOT)





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

Bit Vector Operations

- Operate on Bit Vectors
  - Operations applied bitwise

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

Bit-Level Operations in C

- Operations &, |, ~, ^ Available in C
  - Apply to any "integral" data type
    - `long, int, short, char, unsigned`
  - View arguments as bit vectors
  - Arguments applied bit-wise
- Examples (Char data type)
  - `~0x41` → `0xBE`
    - `~01000001`$_2$ → `10111110`$_2$
  - `~0x00` → `0xFF`
    - `~00000000`$_2$ → `11111111`$_2$
  - `0x69 & 0x55` → `0x41`
    - `01101001`$_2$ `&` `01010101`$_2$ → `01000001`$_2$
  - `0x69 | 0x55` → `0x7D`
    - `01101001`$_2$ `|` `01010101`$_2$ → `01111101`$_2$

Aside: Logic Operations in C

- Contrast to Logical Operators
  - `&&, ||, !`
    - View 0 as "False"
    - Anything nonzero as "True"
    - Always return 0 or 1
    - Early termination (e.g., 0 && 1 && 1)
- Examples (char data type)
  - `!0x41 → 0x00`
  - `!0x00 → 0x01`
  - `!!0x41 → 0x01`
  - `0x69 && 0x55 → 0x01`
  - `0x69 || 0x55 → 0x01`
  - `p && *p` (avoids null pointer access)

Shift Operations

- Left Shift: `x << y`
  - Shift bit-vector **x** left **y** positions
    - Throw away extra bits on left
  - Fill with 0's on right
- Right Shift: `x >> y`
  - Shift bit-vector **x** right **y** positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left
- Undefined Behavior
  - Shift amount < 0 or ≥ total amount of bits

Argument x	01100010
<< 3	00010*000*
Log. >> 2	*00*011000
Arith. >> 2	*00*011000

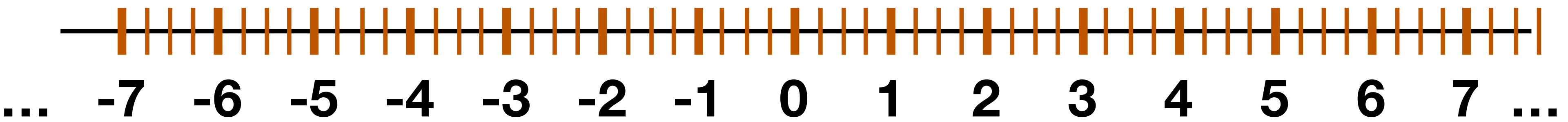
Argument x	10100010
<< 3	00010*000*
Log. >> 2	*00*101000
Arith. >> 2	*11*101000

Today: Representing Information in Binary

- Why Binary (bits)?
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting

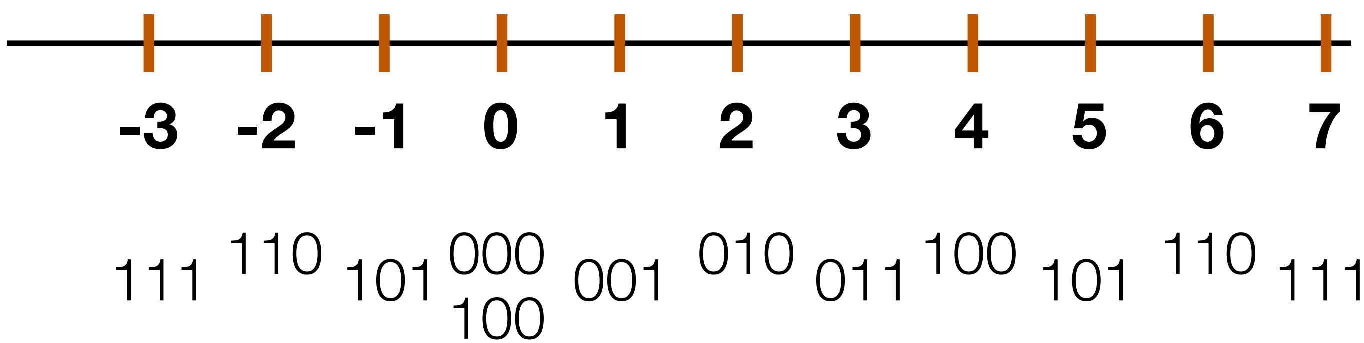
Representing Numbers in Binary

- Different types of number
  - Integer (Negative and Non-negative)
  - Fractions
  - Irrationals



Encoding Negative Numbers

- So far we have been discussing non-negative numbers: so called unsigned. How about negative numbers?
- Solution 1: Sign-magnitude
  - First bit represents sign; 0 for positive; 1 for negative
  - The rest represents magnitude



Sign-Magnitude Implications

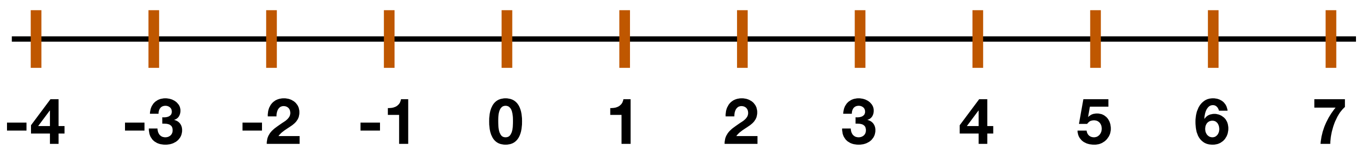
- Bits have different semantics
  - Two zeros…
  - Normal arithmetic doesn't work
  - Make hardware design harder

```
    010          2
+)  101      +) -1
-------      -----
    111         -3
```

Signed Value	Binary
0	000
1	001
2	010
3	011
-0	100
-1	101
-2	110
-3	111

Encoding Negative Numbers

- Solution 2: Two's Complement



Signed Weight	Unsigned Weight	Bit Position
2^0	2^0	0
2^1	2^1	1
-2^2	2^2	2

$$101_2 = 1*2^0 + 0*2^1 - 1*2^2 = -3_{10}$$

Signed	Unsigned	Binary
0	0	000
1	1	001
2	2	010
3	3	011
-4	4	100
-3	5	101
-2	6	110
-1	7	111

Two-Complement Encoding Example

```
x =      15213: 00111011 01101101
y =     -15213: 11000100 10010011
```

Weight	15213		-15213	
1	1	1	1	1
2	0	0	1	2
4	1	4	0	0
8	1	8	0	0
16	0	0	1	16
32	1	32	0	0
64	1	64	0	0
128	0	0	1	128
256	1	256	0	0
512	1	512	0	0
1024	0	0	1	1024
2048	1	2048	0	0
4096	1	4096	0	0
8192	1	8192	0	0
16384	0	0	1	16384
-32768	0	0	1	-32768
Sum		**15213**		**-15213**

Two-Complement Implications

- Only 1 zero
- Usual arithmetic still works
- There is a bit that represents the sign!
- Most widely used in today's machines

Signed	Binary
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

```
    010
+)  101
-------
    111
```

```
      2
+)   -3
-------
     -1
```

Numeric Ranges

- Unsigned Values
  - *UMin* = 0
    000…0
  - *UMax* = $2^w - 1$
    111…1

- Two's Complement Values
  - *TMin* = -2^{w-1}
    100…0
  - *TMax* = $2^{w-1} - 1$
    011…1
- Other Values
  - Minus 1
    111…1

Values for *W* = 16

	Decimal	Hex	Binary
`UMax`	**65535**	`FF FF`	`11111111 11111111`
`TMax`	**32767**	`7F FF`	`01111111 11111111`
`TMin`	**-32768**	`80 00`	`10000000 00000000`
`-1`	**-1**	`FF FF`	`11111111 11111111`
`0`	**0**	`00 00`	`00000000 00000000`

Data Representations in C (in Bytes)

- By default variables are signed
- Unless explicitly declared as unsigned (e.g., `unsigned int`)
- Signed variables use two-complement encoding

C Data Type	32-bit	64-bit
`char`	1	1
`short`	2	2
`int`	4	4
`long`	4	8

Data Representations in C (in Bytes)

	W			
	8	**16**	**32**	**64**
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

C Data Type	32-bit	64-bit
`char`	1	1
`short`	2	2
`int`	4	4
`long`	4	8

- C Language
  - `#include <limits.h>`
  - Declares constants, e.g.,
    - `ULONG_MAX`
    - `LONG_MAX`
    - `LONG_MIN`
  - Values platform specific