



UC Berkeley  
Teaching Professor  
Dan Garcia

# CS61C

## Great Ideas in Computer Architecture (a.k.a. Machine Structures)

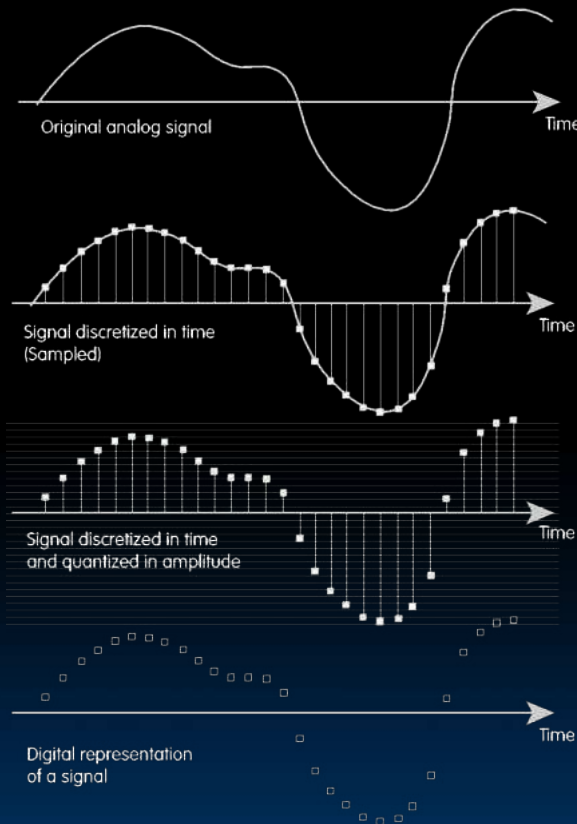


UC Berkeley  
Professor  
Bora Nikolić

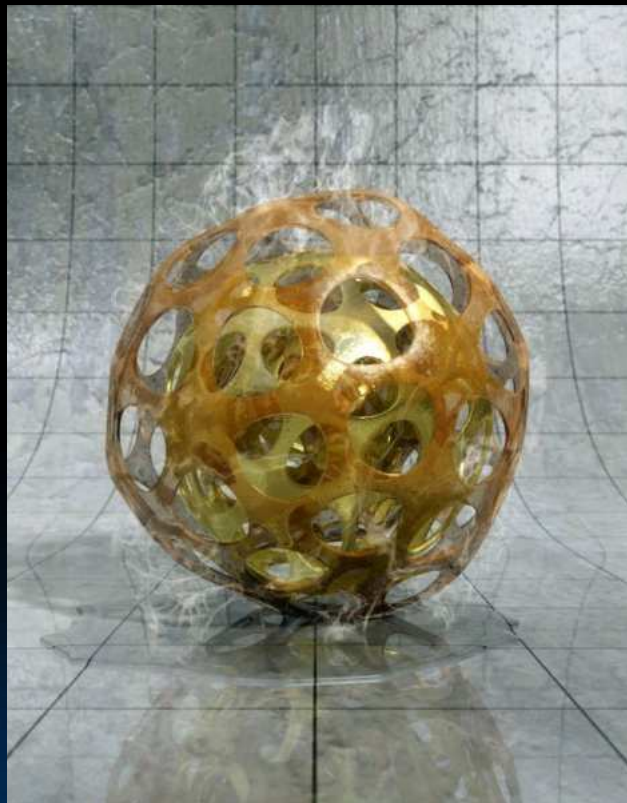
## Number Representation

# Data input: Analog → Digital

- Real world is analog!
- To import analog information, we must do two things
  - Sample
    - E.g., for a CD, every 44,100ths of a second, we ask a music signal how loud it is.
  - Quantize
    - For every one of these samples, we figure out where, on a 16-bit (65,536 tic-mark) “yardstick”, it lies.



# Digital data not necessarily born Analog...





# BIG IDEA: Bits can represent anything!!

- Characters?
  - 26 letters  $\Rightarrow$  5 bits ( $2^5 = 32$ )
  - upper/lower case + punctuation  $\Rightarrow$  7 bits (in 8) (“ASCII”)
  - standard code to cover all the world’s languages  $\Rightarrow$  8,16,32 bits (“Unicode”)  
[www.unicode.com](http://www.unicode.com)
- Logical values?
  - $0 \rightarrow \text{False}$ ,  $1 \rightarrow \text{True}$
- colors ? Ex: Red (00) Green (01) Blue (11)
- locations / addresses? commands?
- **MEMORIZE**: N bits  $\Leftrightarrow$  at most  $2^N$  things



**Binary**  
**Decimal**  
**Hex**



# Base 10 (Ten) #s, Decimals

---

Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Example:

$$3271 = 3271_{10} = (3 \times 10^3) + (2 \times 10^2) + (7 \times 10^1) + (1 \times 10^0)$$



# Base 2 (Two) #s, Binary (to Decimal)

Digits: 0, 1 (binary digits □ bits)

Example: “**1101**” in binary? (“**0b1101**”)

$$\begin{aligned} \mathbf{1101}_2 &= (\mathbf{1} \times 2^3) + (\mathbf{1} \times 2^2) + (\mathbf{0} \times 2^1) + (\mathbf{1} \times 2^0) \\ &= 8 + 4 + 0 + 1 \\ &= 13 \end{aligned}$$



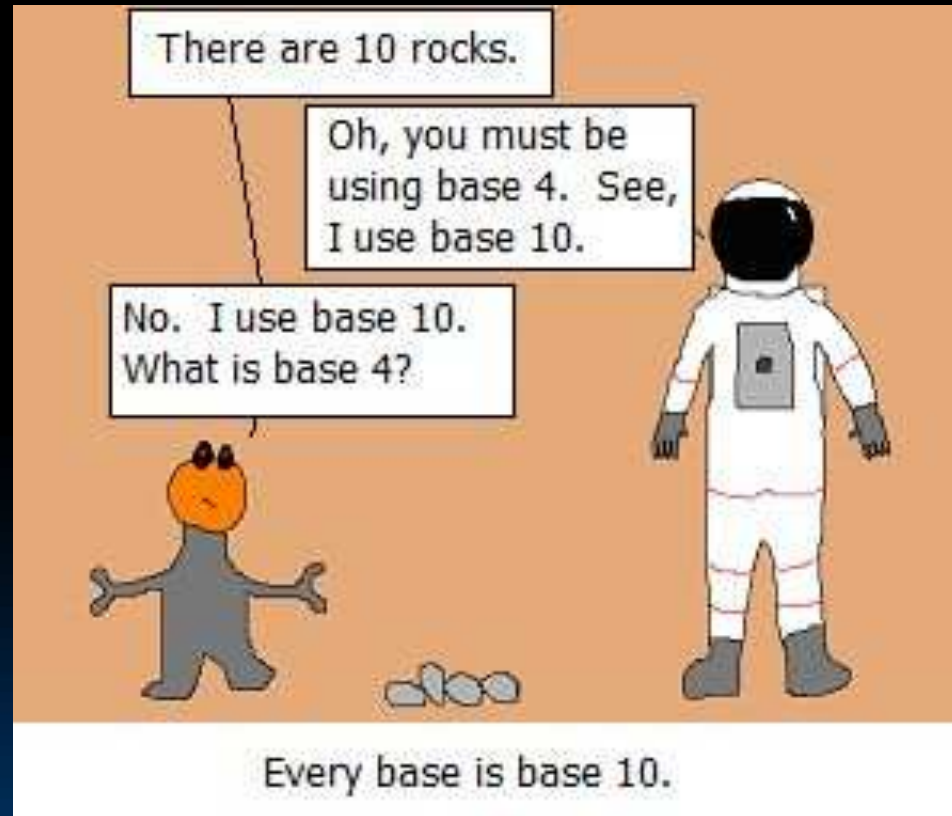
# Base 16 (Sixteen) #s, Hexadecimal (to Decimal)

Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F  
10, 11, 12, 13, 14, 15

Example: “A5” in Hexadecimal?

$$\begin{aligned} 0xA5 &= A5_{16} = (10 \times 16^1) + (5 \times 16^0) \\ &= 160 + 5 \\ &= 165 \end{aligned}$$

# Every Base is Base 10...



# Convert from Decimal to Binary

- E.g., 13 to binary?
- Start with the columns

1	2 <sup>3</sup> = 8	2 <sup>2</sup> = 4	2 <sup>1</sup> = 2	2 <sup>0</sup> = 1
0	1	1	0	1

- Left to right, is (column)  $\leq$  number **n**?
  - If yes, put how many of that column fit in **n**, subtract col \* that many from **n**, keep going.
  - If not, put 0 and keep going. (and Stop at 0)

# Convert from Decimal to Hexadecimal

- E.g., 165 to hexadecimal?
- Start with the columns

<u>16</u>	$16^3 = 4096$	$16^2 = 256$	$16^1 = 16$	$16^0 = 1$
<u>35</u>				
0	0	0	(10) A	5

- Left to right, is (column)  $\leq$  number **n**?
  - ▢ If yes, put how many of that column fit in **n**, subtract col \* that many from **n**, keep going.
  - ▢ If not, put 0 and keep going. (and Stop at 0)



# Convert Binary ☐ ☐ Hexadecimal

- Binary ☐ Hex? Easy!

- ☐ Always **left-pad** with 0s to make full 4-bit values, then look up!

- ☐ E.g., 0b11110 to Hex?

- 0b11110 ☐ 0b00011110

- Then look up: 0x1E

- Hex ☐ Binary? Easy!


- ☐ Just look up, drop leading 0s

- 0x1E ☐ 0b00011110 ☐ 0b11110

D	H	B
00	0	0000
01	1	0001
02	2	0010
03	3	0011
04	4	0100
05	5	0101
06	6	0110
07	7	0111
08	8	1000
09	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111



# Decimal vs Hexadecimal vs Binary

- 4 Bits
  - 1 “Nibble”
  - 1 Hex Digit = 16 things
- 8 Bits
  - 1 “Byte”
  - 2 Hex Digits = 256 things
  - Color is usually
    - 0-255 Red,
    - 0-255 Green,
    - 0-255 Blue.
    - #D0367F= 

D	H	B
00	0	0000
01	1	0001
02	2	0010
03	3	0011
04	4	0100
05	5	0101
06	6	0110
07	7	0111
08	8	1000
09	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111



# Which base do we use?

- **Decimal:** great for humans, especially when doing arithmetic
- **Hex:** if human looking at long strings of binary numbers, its much easier to convert to hex and see 4 bits/symbol
  - Terrible for arithmetic on paper
- **Binary:** what computers use;  
you will learn how computers do +, -, \*, /
  - To a computer, numbers always binary
  - Regardless of how number is written:
  - $32_{\text{ten}} == 32_{10} == 0x20 == 100000_2 == 0b100000$
  - Use subscripts “ten”, “hex”, “two” in book, slides when might be confusing



# The computer knows it, too...

```
#include <stdio.h>

int main() {
    const int N = 1234;

    printf("Decimal: %d\n", N);
    printf("Hex:      %x\n", N);
    printf("Octal:     %o\n", N);

    printf("Literals (not supported by all compilers):\n");
    printf("0x4d2      = %d (hex)\n", 0x4d2);
    printf("0b10011010010 = %d (binary)\n", 0b10011010010);
    printf("02322      = %d (octal, prefix 0 - zero)\n", 02322);
}
```

**Output**      Decimal: 1234  
Hex:          4d2  
Octal:        2322  
Literals (not supported by all compilers):  
0x4d2        = 1234 (hex)  
0b10011010010 = 1234 (binary)  
02322        = 1234 (octal, prefix 0 - zero)

# Number Representatio ns

# What to do with representations of numbers?

- What to do with number representations?

- Add them
- Subtract them
- Multiply them
- Divide them
- Compare them

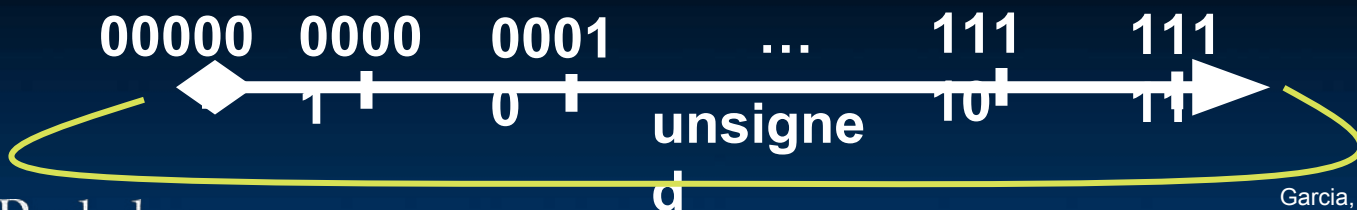
$$\begin{array}{r}
 1 \ 0 \ 1 \ 0 \\
 + \ 0 \ 1 \ 1 \ 1 \\
 \hline
 \end{array}$$

- Example:  $10 + 7 = 17$

- ...so simple to add in binary that we can build circuits to do it!
- Subtraction just as you would in decimal
- Comparison: How do you tell if  $X > Y$  ?

# What if too big?

- Binary bit patterns are simply representatives of numbers. Abstraction!
  - Strictly speaking they are called "numerals".
- Numerals really have an  $\infty$  number of digits
  - with almost all being same (00...0 or 11...1) except for a few of the rightmost digits
  - Just don't normally show leading digits
- If result of add (or -, \*, / ) cannot be represented by these rightmost HW bits, we say overflow occurred



# How to Represent Negative Numbers?

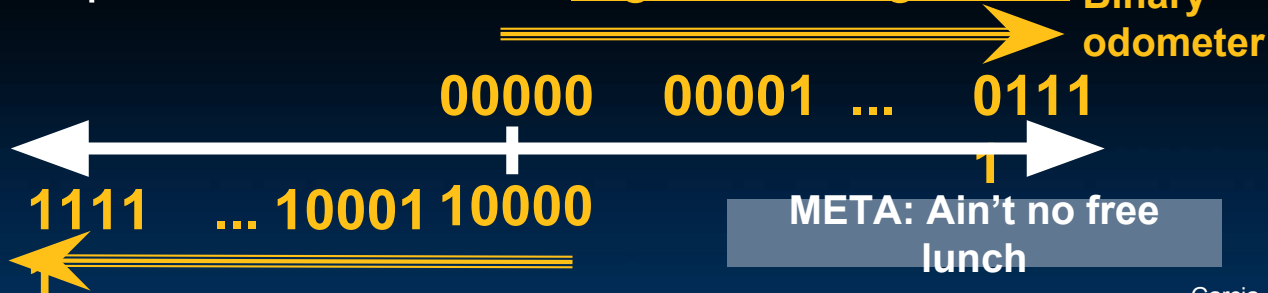
(C's unsigned int, C18's uintN\_t)

- So far, unsigned numbers



- Obvious solution: define leftmost bit to be sign!  
 $\square 0 \square + \quad 1 \square -$  ...and rest of bits are numerical value

- Representation called Sign and Magnitude



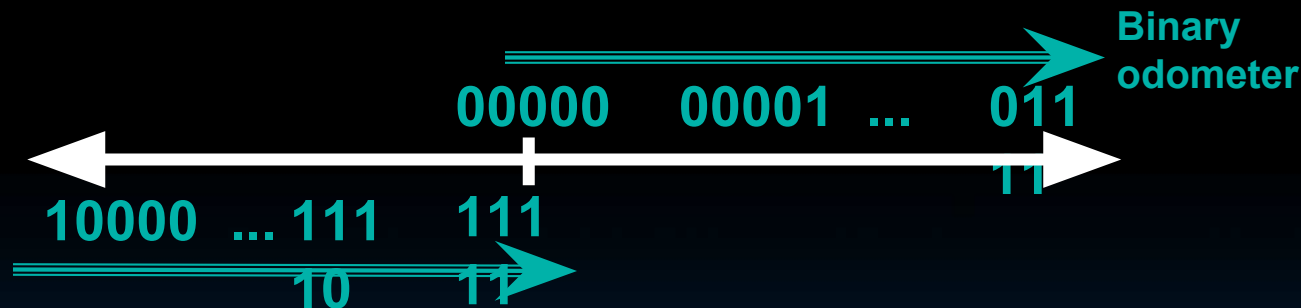


# Shortcomings of Sign and Magnitude?

- Arithmetic circuit complicated
  - Special steps depending on if signs are the same or not
- Also, two zeros
  - $0x00000000 = +0_{\text{ten}}$
  - $0x80000000 = -0_{\text{ten}}$
  - What would two 0s mean for programming?
- Also, incrementing “binary odometer”, sometimes increases values, and sometimes decreases!
- Therefore sign and magnitude used only in signal processors

# Another try: complement the bits

- Example:  $7_{10} = 00111_2$   $-7_{10} = 11000_2$
- Called One's Complement
- Note: positive numbers have leading 0s, negative numbers have leading 1s.



- What is -00000 ? Answer: 11111
- How many positive numbers in N bits?
- How many negative numbers?



# Shortcomings of One's Complement?

- Arithmetic still somewhat complicated
- Still two zeros
  - $0x00000000 = +0_{\text{ten}}$
  - $0xFFFFFFFF = -0_{\text{ten}}$
- Although used for a while on some computer products, one's complement was eventually abandoned because another solution was better.



# Two's Complement & Bias Encoding



# Standard Negative # Representation

- Problem is the negative mappings “overlap” with the positive ones (the two 0s). Want to shift the negative mappings left by one.
  - Solution! For negative numbers, complement, then add 1 to the result
- As with sign and magnitude, & one’s compl. leading 0s □ positive, leading 1s □ negative
  - 000000...xxx is  $\geq 0$ , 111111...xxx is  $< 0$
  - except 1...1111 is -1, not -0 (as in sign & mag.)
- This representation is **Two’s Complement**
  - This makes the hardware simple!

(C’s `int`, C18’s `intN_t`, aka a “signed integer”)

# Two's Complement Formula

- Can represent positive and negative numbers in terms of the bit value times a power of 2:

$$d_{31} \times -(2^{31}) + d_{30} \times 2^{30} + \dots + d_2 \times 2^2 + d_1 \times 2^1 + d_0 \times 2^0$$

- Example:  $1101_{\text{two}}$  in a nibble?

$$= 1 \times -(2^3) + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$= -2^3 + 2^2 + 0 + 2^0$$

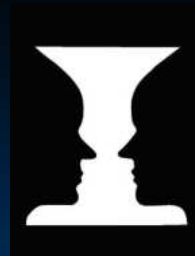
$$= -8 + 4 + 0 + 1$$

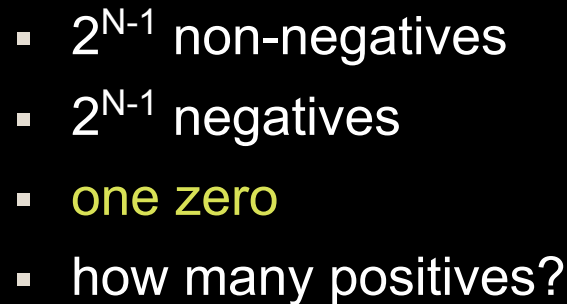
$$= -8 + 5$$

$$= -3_{\text{ten}}$$

**Example: -3 to +3 to -3 (again, in a nibble):**

<b>x</b>	:	1101	
<b>x'</b>	:	0010	<sub>two</sub>
<b>+1</b>	:	0011	<sub>two</sub>
<b>( )'</b>	:	1100	<sub>two</sub>
<b>+1</b>	:	1101	<sub>two</sub>

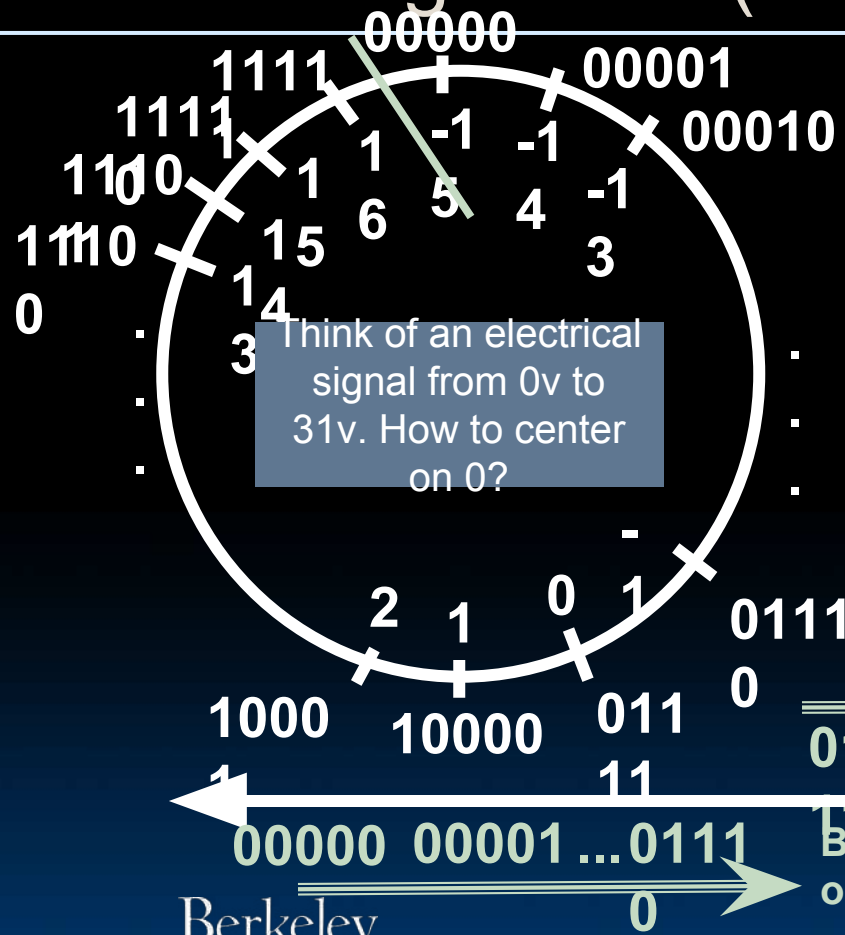




# Binary odometer

## Number Representation (26)

# Bias Encoding: $N = 5$ (bias = -15)



- # = unsigned + bias
- Bias for  $N$  bits chosen as  $-(2^{N-1}-1)$
- **one zero**
- how many positives?





# And in summary...

META: We often make design decisions to make HW simple

- We represent “things” in computers as particular bit patterns:  $N$  bits  $\Rightarrow 2^N$  things
- These 5 integer encodings have different benefits; 1s complement and sign/mag have most problems.
- **unsigned** (C18's `uintN_t`) :



- **Two's complement** (C99's `intN_t`) universal, learn!

