inst.eecs.berkeley.edu/~cs61c

CS61C : Machine Structures

Lecture #2 – Number Representation

2010-08-29

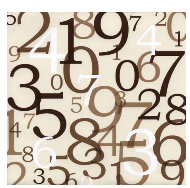
There is one handout today at the entrance!

Lecturer SOE Dan Garcia

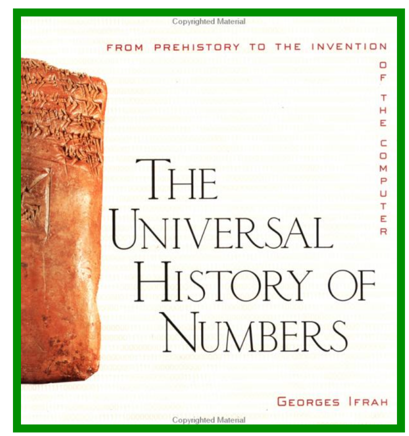
www.cs.berkeley.edu/~ddgarcia

Great book ⇒

The Universal History of Numbers



by Georges Ifrah



Review

- CS61C: Learn 6 great ideas in computer architecture to enable high performance programming via parallelism, not just learn C

  1. Layers of Representation/Interpretation
  2. Moore's Law
  3. Principle of Locality/Memory Hierarchy
  4. Parallelism
  5. Performance Measurement and Improvement
  6. Dependability via Redundancy

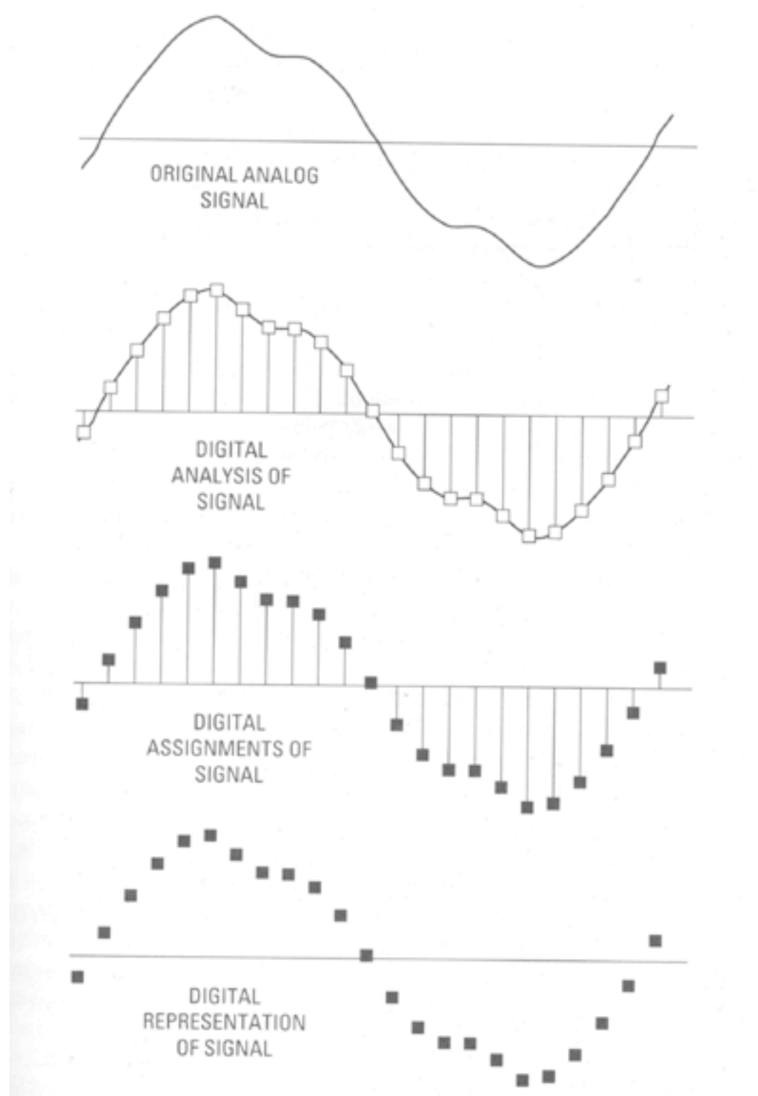
Putting it all in perspective…

"If the automobile had followed the same development cycle as the computer,

– Robert X. Cringely

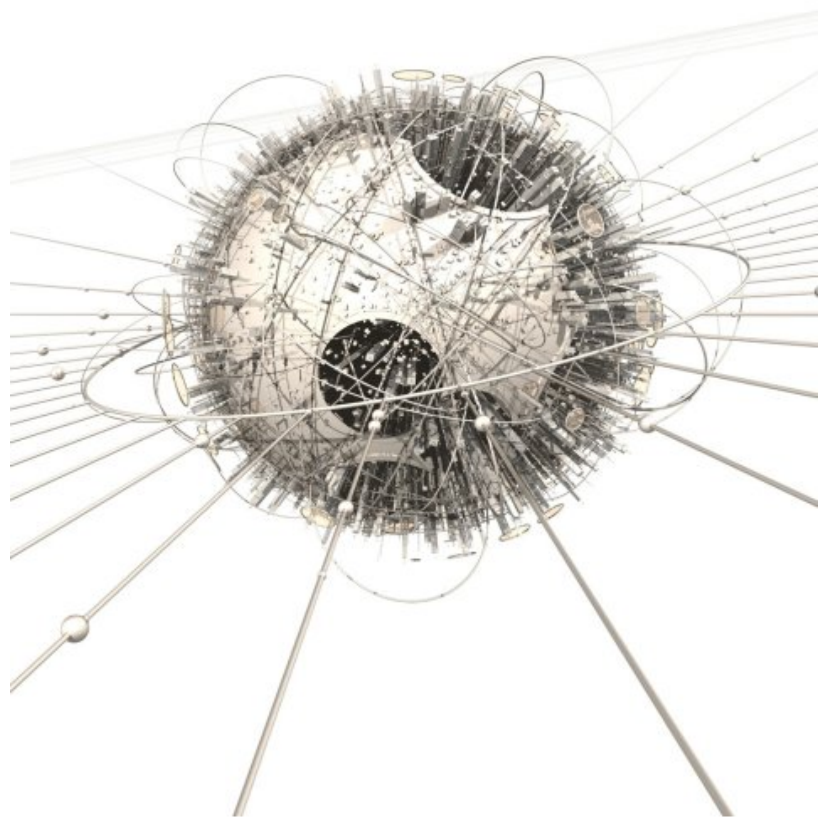
Data input: Analog → Digital

- **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.**



www.joshuadysart.com/journal/archives/digital_sampling.gif

Digital data not nec born Analog…



hof.povray.org

BIG IDEA: Bits can represent anything!!

- Characters?
  - 26 letters ⇒ 5 bits ($2^5 = 32$)
  - upper/lower case + punctuation ⇒ 7 bits (in 8) ("ASCII")
  - standard code to cover all the world's languages ⇒ 8,16,32 bits ("Unicode") `www.unicode.com`
- Logical values?
  - 0 ⇒ False, 1 ⇒ True
- colors ? Ex: *Red (00)* *Green (01)* *Blue (11)*
- locations / addresses? commands?
- MEMORIZE: N bits ⇔ at most 2^N things

How many bits to represent π ?

a) 1

b) 9 (π = 3.14, so that's 011 "." 001 100)

c) 64 (Since Macs are 64-bit machines)

d) Every bit the machine has!

e) ∞

What to do with representations of numbers?

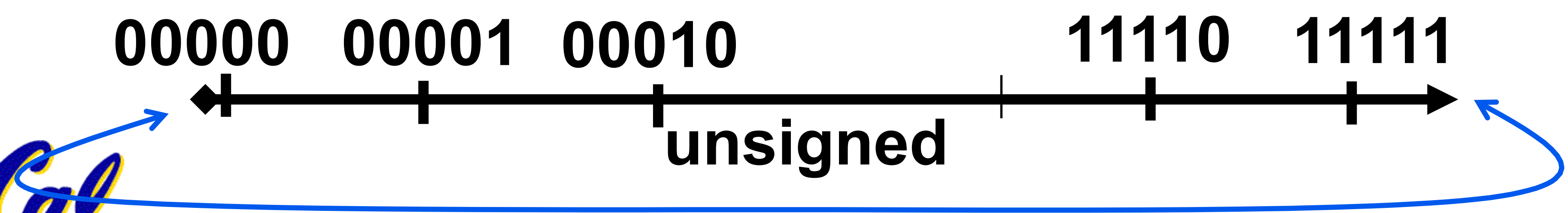
- **Just what we do with numbers!**
  - Add them
  - Subtract them
  - Multiply them
  - Divide them
  - Compare them

```
     1  1
     1  0  1  0
+    0  1  1  1
-----------------
  1  0  0  0  1
```

- **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 above are simply **representatives** of numbers. Abstraction! Strictly speaking they are called “numerals”.
- Numbers really have an ∞ 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, **overflow** is said to have occurred.

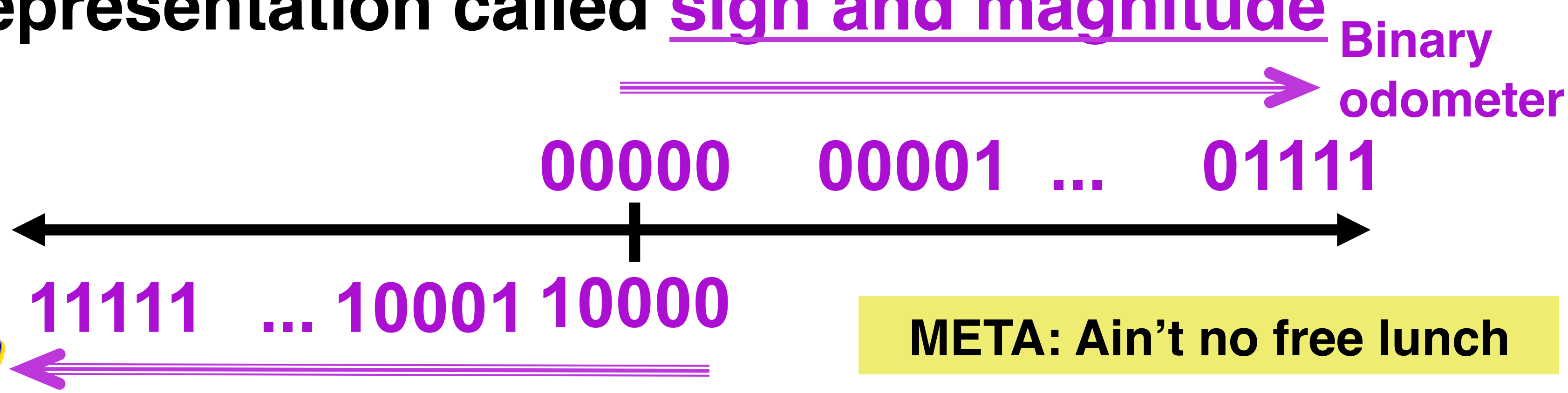


How to Represent Negative Numbers?

(C's `unsigned int`, C99's `uintN_t`)

- So far, unsigned numbers

- Obvious solution: define leftmost bit to be sign!
  - 0 ➔ + 1 ➔ –
  - Rest of bits can be numerical value of number
- Representation called sign and magnitude



META: Ain't no free lunch

Shortcomings of sign and magnitude?

- **Arithmetic circuit complicated**
  - Special steps depending whether signs are the same or not
- **Also, two zeros**
  - 0x00000000 = $+0_{ten}$
  - 0x80000000 = -0_{ten}
  - What would two 0s mean for programming?
- **Also, incrementing "binary odometer", sometimes increases values, and sometimes decreases!**
- **Therefore sign and magnitude abandoned**

Administrivia

- **Upcoming lectures**
  - Next few lectures: Introduction to C
- **Lab overcrowding**
  - Remember, you can go to ANY discussion (none, or one that doesn't match with lab, or even more than one if you want)
  - Overcrowded labs - consider finishing at home and getting checkoffs in lab, or bringing laptop to lab
  - If you're checked off in 1^{st} hour, you get an extra point on the labs!
  - TAs get 24x7 cardkey access (and will announce after-hours times)
- **Enrollment**
  - It will work out, don't worry
- **Soda locks doors @ 6:30pm & on weekends**
- **Look at class website, piazza often!**

  `http://inst.eecs.berkeley.edu/~cs61c/`
  `piazza.com`

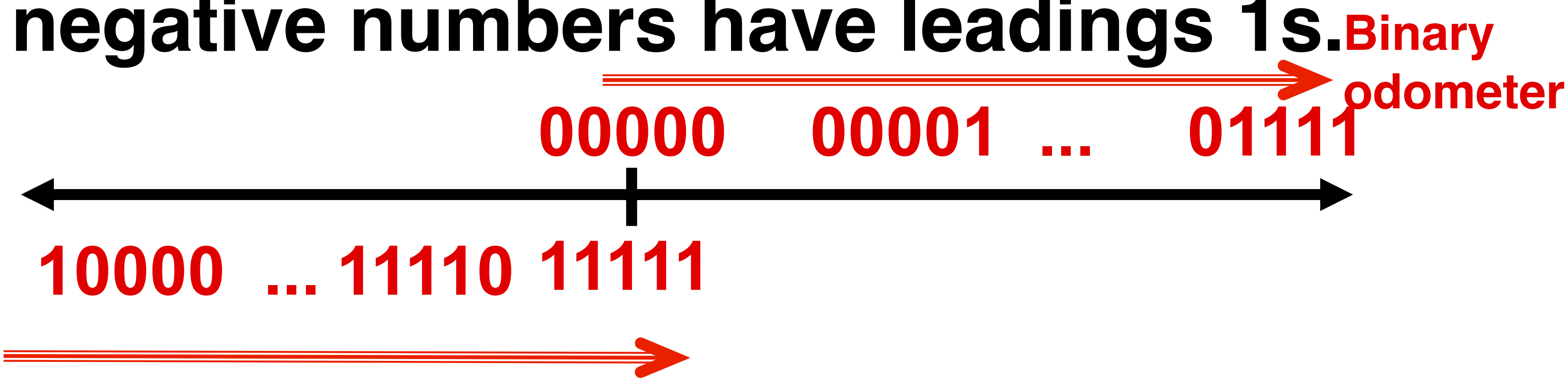
Iclickerskinz.com

Great DeCal courses I supervise

- **UCBUGG (3 units, P/NP)**
  - UC Berkeley Undergraduate Graphics Group
  - TuTh 5-7pm in 200 Sutardja Dai
  - Learn to create a short 3D animation
  - No prereqs (but they might have too many students, so admission not guaranteed)
  - `http://ucbugg.berkeley.edu`
- **MS-DOS X (2 units, P/NP)**
  - Macintosh Software Developers for OS X
  - TuTh 7-9pm in 200 Sutardja Dai
  - Learn to program iOS devices!
  - No prereqs (other than interest)
  - `http://msdosx.berkeley.edu`

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 leadings 1s.



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

Shortcomings of One's complement?

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

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 ≥ 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`, aka a "signed integer")

(Also C's `short`, `long long`, …, C99's `int`*`N`*`_t`)

Two's Complement Formula

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

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

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

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

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

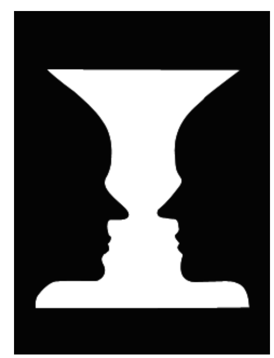
$$= -8 + 4 + 0 + 1$$

$$= -8 + 5$$

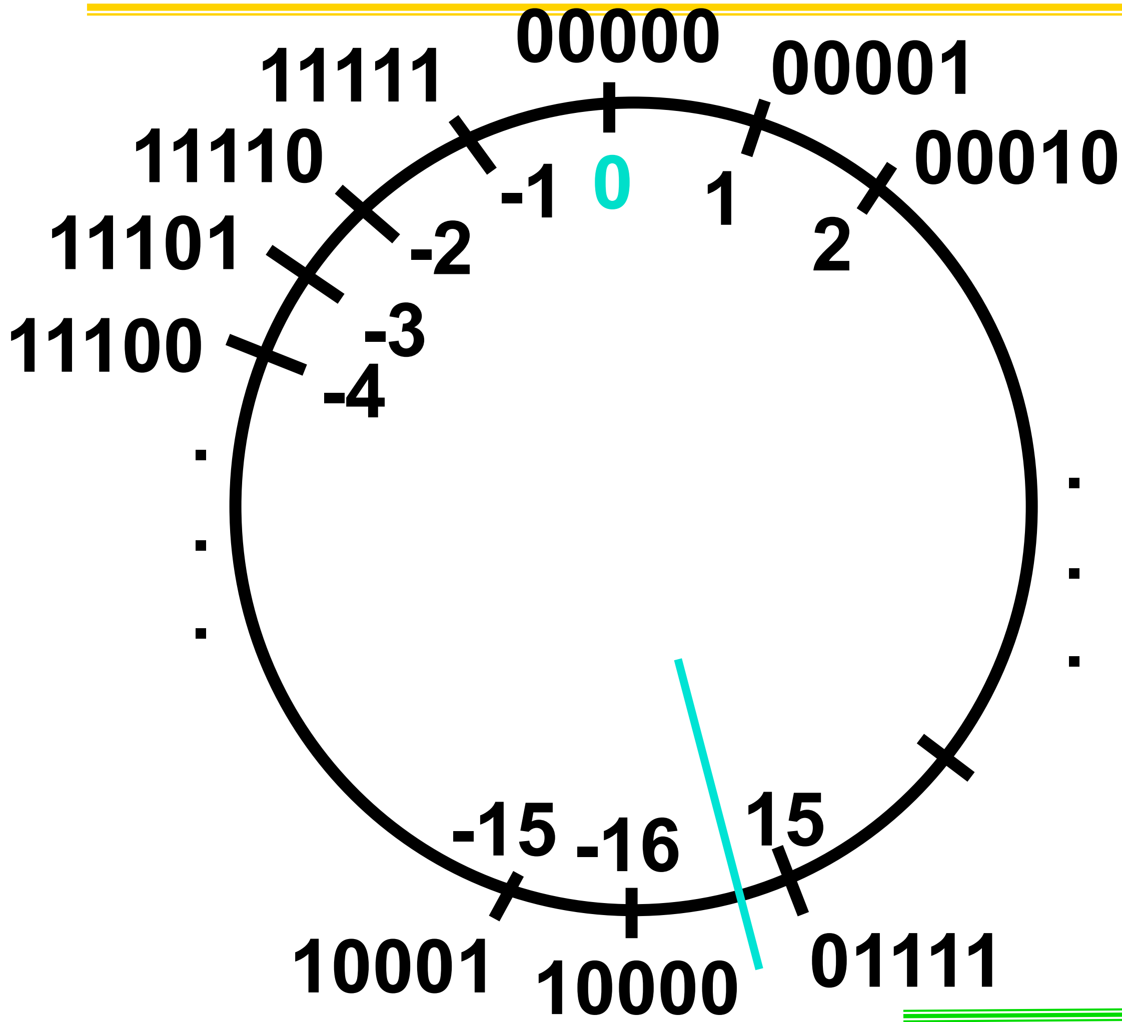
$$= -3_{ten}$$

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

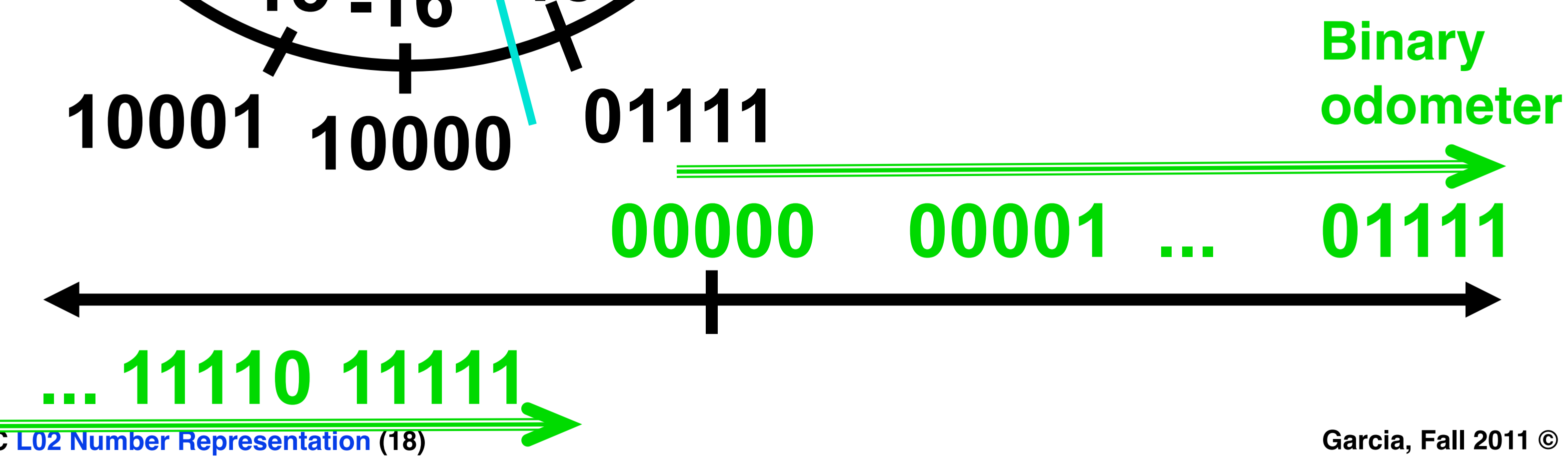
x	:	1101_{two}
x'	:	0010_{two}
+1	:	0011_{two}
()'	:	1100_{two}
+1	:	1101_{two}



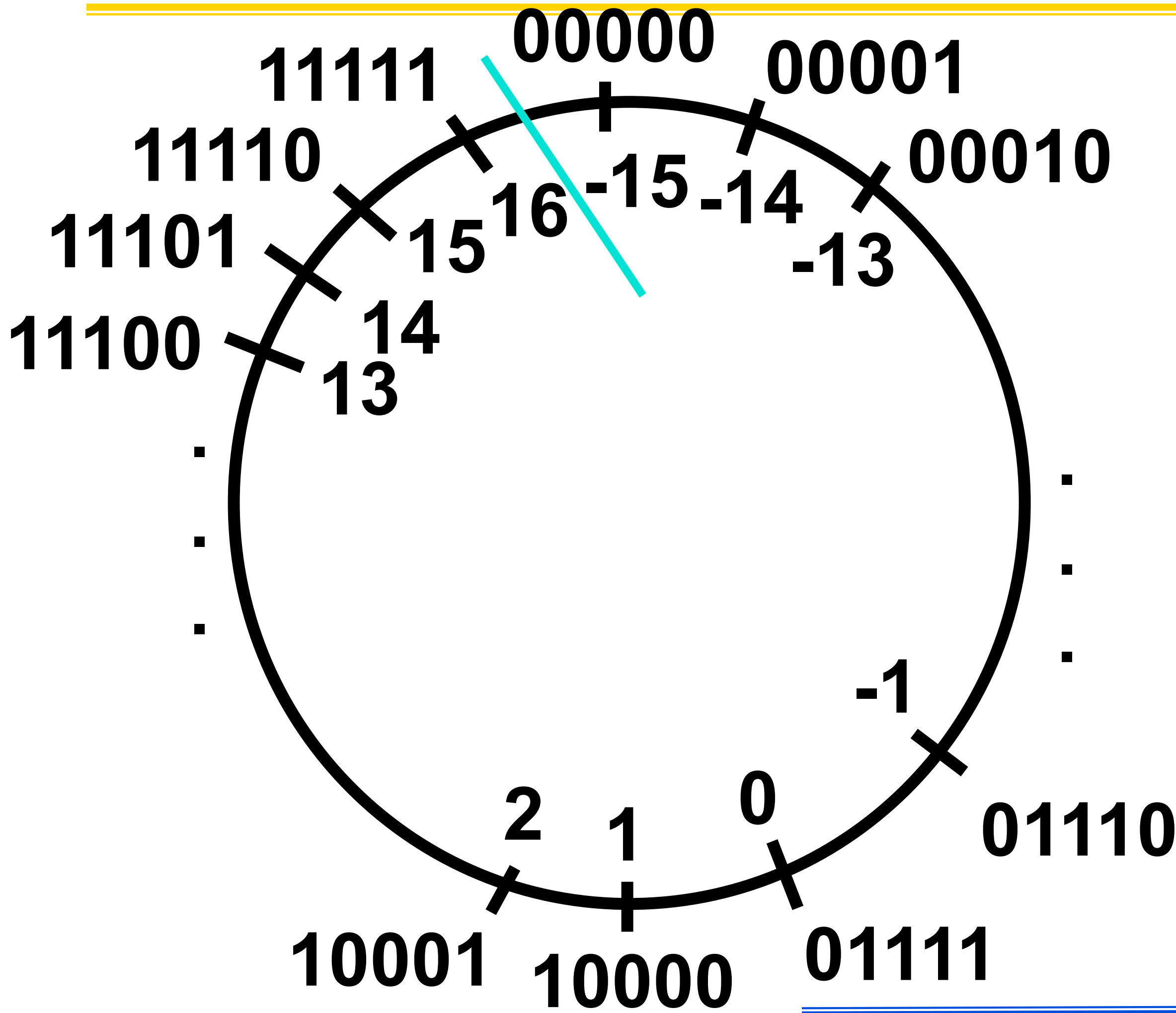
2’s Complement Number “line”: N = 5



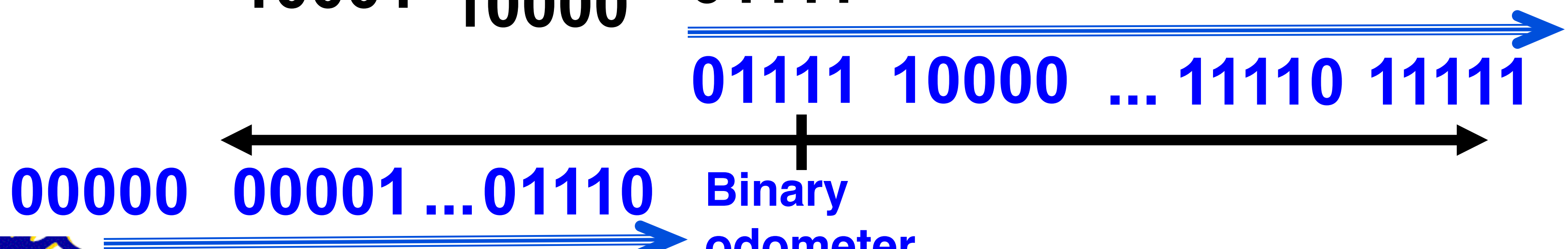
- 2^{N-1} non-negatives
- 2^{N-1} negatives
- one zero
- how many positives?



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



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



How best to represent -12.75?

a) 2s Complement (but shift binary pt)

b) Bias (but shift binary pt)

c) Combination of 2 encodings

d) Combination of 3 encodings

e) We can’t

Shifting binary point means “divide number by some power of 2. E.g., $11_{10} = 1011.0_2 \rightarrow 10.110_2 = (11/4)_{10} = 2.75_{10}$

And in summary…

META: We often make design decisions to make HW simple

- We represent "things" in computers as particular bit patterns: N bits ⇒ 2^N things
- These 5 integer encodings have different benefits; 1s complement and sign/mag have most problems.
- unsigned (C99's `uintN_t`) :

00000 00001 … 01111 10000 … 11111

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

00000 00001 … 01111

10000 … 11110 11111

- Overflow: numbers ∞; computers finite,errors!

META: Ain't no free lunch

REFERENCE: 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 look 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:
  - $32_{ten} == 32_{10} == 0x20 == 100000_2 == 0b100000$
  - Use subscripts “ten”, “hex”, “two” in book, slides when might be confusing

Two's Complement for N=32

$0000 \ldots 0000\ 0000\ 0000\ 0000_{two} =$	0_{ten}
$0000 \ldots 0000\ 0000\ 0000\ 0001_{two} =$	1_{ten}
$0000 \ldots 0000\ 0000\ 0000\ 0010_{two} =$	2_{ten}
. . .	
$0111 \ldots 1111\ 1111\ 1111\ 1101_{two} =$	$2,147,483,645_{ten}$
$0111 \ldots 1111\ 1111\ 1111\ 1110_{two} =$	$2,147,483,646_{ten}$
$0111 \ldots 1111\ 1111\ 1111\ 1111_{two} =$	$2,147,483,647_{ten}$
$1000 \ldots 0000\ 0000\ 0000\ 0000_{two} =$	$-2,147,483,648_{ten}$
$1000 \ldots 0000\ 0000\ 0000\ 0001_{two} =$	$-2,147,483,647_{ten}$
$1000 \ldots 0000\ 0000\ 0000\ 0010_{two} =$	$-2,147,483,646_{ten}$
. . .	
$1111 \ldots 1111\ 1111\ 1111\ 1101_{two} =$	-3_{ten}
$1111 \ldots 1111\ 1111\ 1111\ 1110_{two} =$	-2_{ten}
$1111 \ldots 1111\ 1111\ 1111\ 1111_{two} =$	-1_{ten}

- One zero; 1st bit called sign bit
- 1 "extra" negative: no positive $2,147,483,648_{ten}$

Two's comp. shortcut: Sign extension

- **Convert 2's complement number rep. using n bits to more than n bits**
- **Simply replicate the most significant bit (sign bit) of smaller to fill new bits**
  - 2's comp. positive number has infinite 0s
  - 2's comp. negative number has infinite 1s
  - Binary representation hides leading bits; sign extension restores some of them
  - 16-bit -4_{ten} to 32-bit:

$$1111\ 1111\ 1111\ 1100_{two}$$

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_{two}$$