CSEE 3827: Fundamentals of Computer Systems, Spring 2022

Lecture 1

Prof. Dan Rubenstein (danr@cs.columbia.edu)

Quick Announcements

- Please fill in Office Hour Availability at:
- http://uribe.cs.columbia.edu/sched/table.php (due Tues Jan 25)
- Waitlisted students:
    - Section 1 & 2: please enroll in Section 3 for now (1 & 2 overloaded on waitlist)
    - Section 3: larger room forthcoming… will assign as soon as possible
    - EdStem bug for “Observers” - reported the bug, they’ll try to fix… in the meantime I’m adding folks manually

A note on lecture slides

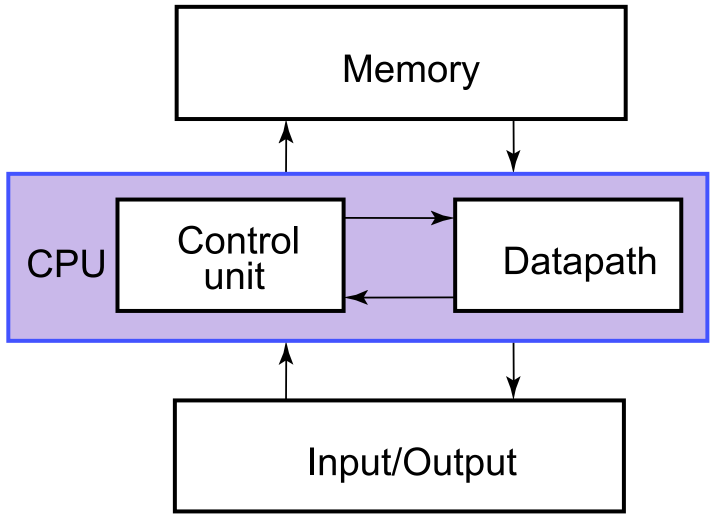
- There are 13 lecture slides over the course of the term
- They are grouped by topic
- Obviously, we won’t always complete a set of lecture slides in a single lecture. Some can take 2,3, or 4 class lectures

Agenda (M&K&M Ch 1, 3.11, 9.7)

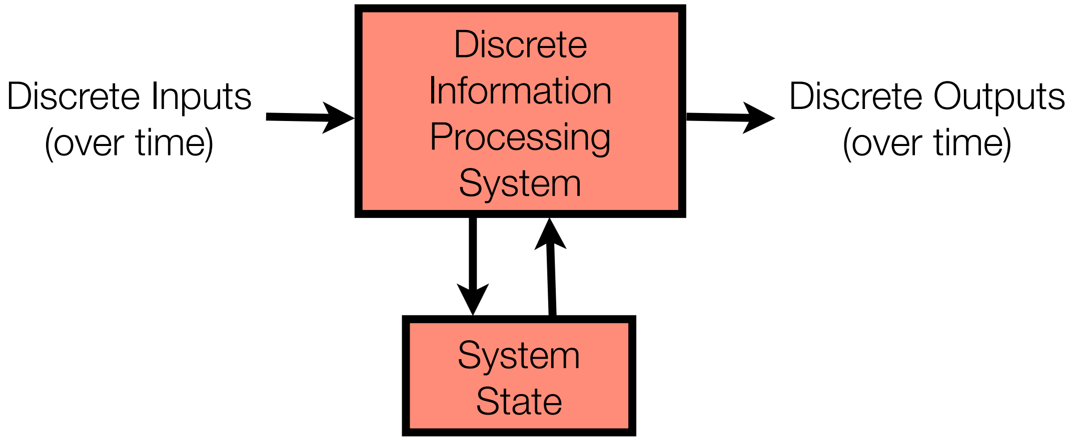
- Computer from a (very high level) Digital Perspective
- Digital/Binary
  - vs Decimal, Hexadecimal
- Terminology:
  - Bit / Byte / Word & Wordsize
  - Highest Order (most significant) Bit, Lowest Order (least significant) bit
- Negative Number Formats:
  - Signed Magnitude
  - 1's Complement
  - 2's Complement
- Floating Point via Binary (P&H 3.5 - skip FP in MIPS subsection)
  - Addition
  - Multiplication

Course Overview: Building a computer (digital perspective)

- CPU: the "brain" of a computer
  - Control unit does calculations on data in datapath
- Memory: stores data (for later use)
- Input/Output: interface to outside (disk, network, monitor, keyboard, mouse, etc.)

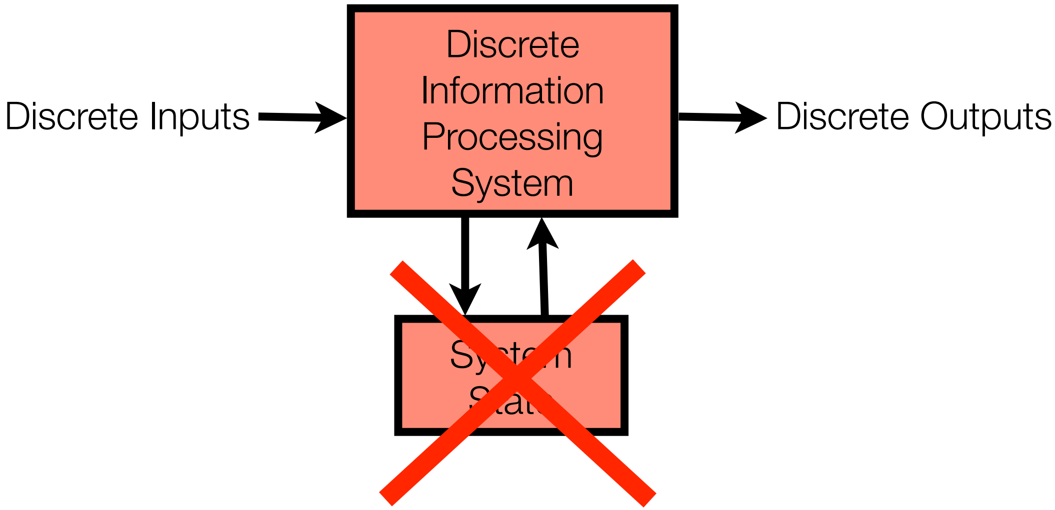


More simplistic view



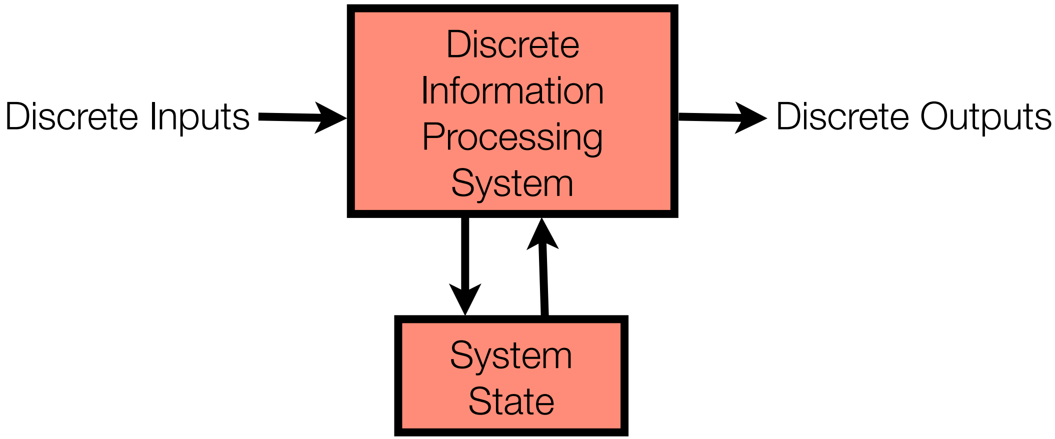
- Course: Starts with this more simple view (on next slide)

Course Overview: 1st quarter



- 1st quarter of course: really simple view: “computer” doesn’t maintain state
- Input → Compute → Output (just a math function)
- Feed same input, get same output

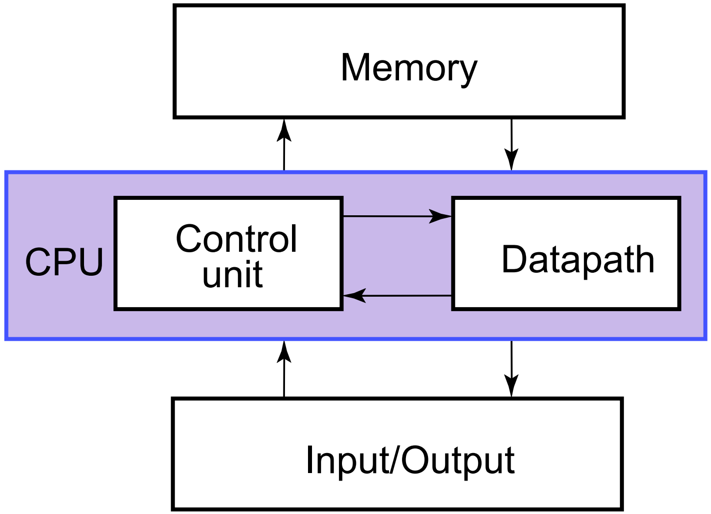
Course Overview: 2nd quarter



- 2nd quarter: “computer” has memory (system state)
    - Can use input & what it has stored in memory to determine output

Course Overview: 2nd Half

- Computer processes programs (stored in memory)
    - program made up of sequences of instructions
- Programs modify data also stored in memory



More on this later in term...

Addition in different bases

Number systems review: Base 10 (Decimal)

- How humans (usually) work with numbers
- 10 digits = {0,1,2,3,4,5,6,7,8,9}
- example: 4537.8 base 10 a.k.a. $(4537.8)_{10}$

$$\begin{array}{ccccccc} 4 & 5 & 3 & 7 & . & 8 \\ \times\ 10^{3} & \times\ 10^{2} & \times\ 10^{1} & \times\ 10^{0} & & \times\ 10^{-1} \end{array}$$

$$4000 + 500 + 30 + 7 + .8 = 4537.8$$

Number systems review: Base 10 (Decimal)

- How humans (usually) work with numbers
- 10 digits = {0,1,2,3,4,5,6,7,8,9}
- example: 4537.8 base 10 a.k.a. $(4537.8)_{10}$

$$
\begin{array}{ccccccccc}
4 & & 5 & & 3 & & 7 & . & 8 \\
\times\ 10^{3} & & \times\ 10^{2} & & \times\ 10^{1} & & \times\ 10^{0} & & \times\ 10^{-1} \\
\hline
4000 & + & 500 & + & 30 & + & 7 & + & .8 \quad = \quad 4537.8
\end{array}
$$

Shifting a column to the left multiplies a digit's value by 10

Number systems: Base 2 (Binary)

- How computers “think” about numbers
- 2 digits = {0,1}
- example: $(1011.1)_2$

$$
\begin{array}{ccccccccccc}
1 & & 0 & & 1 & & 1 & . & 1 & & \\
\times 2^{3} & & \times 2^{2} & & \times 2^{1} & & \times 2^{0} & & \times 2^{-1} & & \\
\hline
8 & + & 0 & + & 2 & + & 1 & + & .5 & = & (11.5)_{10}
\end{array}
$$

Number systems: Base 2 (Binary)

- How computers “think” about numbers
- 2 digits = {0,1}: we refer to binary digits as bits
- example: $(1011.1)_2$

$$
\begin{array}{ccccccccc}
1 & & 0 & & 1 & & 1 & . & 1 \\
\times\, 2^{3} & & \times\, 2^{2} & & \times\, 2^{1} & & \times\, 2^{0} & & \times\, 2^{-1} \\
\hline
8 & + & 0 & + & 2 & + & 1 & + & .5
\end{array}
= (11.5)_{10}
$$

Shifting a column to the left multiplies a digit’s value by 2

Number systems: Base 16 (Hexadecimal)

- How (nerdy) humans interact with computers
- 16 digits = {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}

Base 16 (Hexadecimal) Value	Base 2 (binary) value	Base 10 value
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Number systems: Base 16 (Hexadecimal)

- How (nerdy) humans interact with computers
- 16 digits = {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}
- example: (26BA) [alternate notation for hex: 0x26BA]

$$
\begin{array}{cccccccc}
2 & & 6 & & B & & A & \\
\times\ 16^{3} & & \times\ 16^{2} & & \times\ 16^{1} & & \times\ 16^{0} & \\
\hline
8192 & + & 1536 & + & 176 & + & 10 & = (9914)_{10}
\end{array}
$$

Number systems: Base 16 (Hexadecimal)

- How (nerdy) humans interact with computers
- 16 digits = {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}
- example: (26BA)   [alternate notation for hex: 0x26BA]

$$\begin{array}{cccc} 2 & 6 & B & A \\ \times\ 16^3 & \times\ 16^2 & \times\ 16^1 & \times\ 16^0 \\ \hline \end{array}$$

$$8192 + 1536 + 176 + 10 = (9914)_{10}$$

- Shifting a column to the left multiplies a digit's value by $(16)_{10}$
- Why Important: More concise than binary, but related (a power of 2)

Number ranges

- Cannot map infinite numbers in bounded memory: restrict to some finite range: must choose the representation for a computer

- How many numbers can I represent with ...

  ... 5 digits in decimal? 10^5 possible values $(100{,}000)_{10}$

  ... 8 binary digits? 2^8 possible values $(256)_{10}$

  ... 4 hexadecimal digits? 16^4 possible values $(65{,}536)_{10}$

Why Base 16 is useful

- Computers work in base 2 (binary)
- Writing base 2 numbers is confusing / laborious
- Easy to convert between base 2 and base 16, here's why:
    - To multiply a base-2 4-bit quantity by 16, shift over 4 columns
        - e.g., 1101 0000 is 16x larger than 1101
    - To multiply a base-16 1-digit quality by 16, shift over 1 column
        - e.g., D0 is 16x larger than D

Converting between bases 2 and 16

- 26BA
- = 2000 + 600 + B0 + A
- = 2 (shifted left 3) + 6 (shifted left 2) + B (shifted left 1) + A (hex digits)
- = 0010 (shifted left 12) + 0110 (shifted left 8) + 1011 (shifted left 4) + 1010 (binary digits)
- = 0010 0000 0000 0000 + 0110 0000 0000 + 1011 0000 + 1010
- = 0010 0110 1011 1010

2 6 B A

HEX

Binary

Using Base 16

- Useful for referring to long sequences of binary, e.g., 32-bit quantity:
- 0110 1010 1111 1010 0000 1100 0011 1111
- 6 A F A 0 C 3 F
- i.e., the 32-bit quantity can be written 6AFA0C3F, which is
- 01101010111110100000110000111111

Defs & Terminology

Computer from Digital Perspective

- (Digital) Information: just sequences of binary (0’s and 1’s)
  - True = 1, False = 0
  - Numbers: converted into binary form when “viewed” by computer
    - e.g., 19 = 10011 (16 (1) + 8 (0) + 4 (0) + 2 (1) + 1 (1)) in binary
  - Characters: assigned a specific numerical value (ASCII standard)
    - e.g., ‘A’ = 65 = 1000001, ‘a’ = 97 = 1100001
  - Text is a sequence of characters:
    - “Hi there” = 72, 105, 32, 116, 104, 101, 114, 101
      - = 1001000, 1101001, ...

Terminology: Bit, Byte, Word

- Bit: a single binary digit (a '0' or a '1')
- Byte: a grouping of 8 bits, e.g., 10110010. Q: how many distinct bytes exist?
- Word: a grouping of bits that is computer-architecture dependent
    - The number of bits that the computer architecture can process at once
    - e.g., 64-bit word architectures expect data to be passed in (and returns outputs back out) in 64-bit groupings
    - The number of bits used by the architecture is called its word size
    - OBSERVATION: computers have bounds on how much input they can handle at once
        - word size limits on the sizes of numbers they can deal with (in a single computation cycle)

Terminology 2: Highest / Lowest Order / significant bits

- Bit at the left is highest order (a.k.a. most significant) bit
- Bit at the right is lowest order (a.k.a. least significant) bit

most significant   least significant

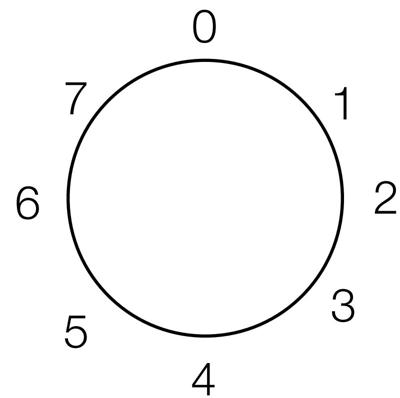
- e.g.,   1 0 1 0 1 1 1
- higher order bits represent “bigger” values when read as numbers
- e.g., above is 64 + 16 + 4 + 2 + 1 = 87 if read as unsigned binary
- Common reference notation for k-bit value: $b_{k-1}b_{k-2}b_{k-3}...b_1b_0$
  - or alternatively: $b_kb_{k-1}b_{k-2}...b_2b_1$

Modular Arithmetic

Modular Arithmetic

- Def: X mod Y = remainder(X/Y)   (i.e., a value between 0 and Y-1)

- e.g.,
  - 22 mod 5 = 2   (22 / 5 = 4 with remainder 2, i.e., 22 = 5 * 4 + 2)
  - 100 mod 9 = 1,
  - -10 mod 3 = -1 mod 3 = 2 mod 3   (-4 * 3 + 2)



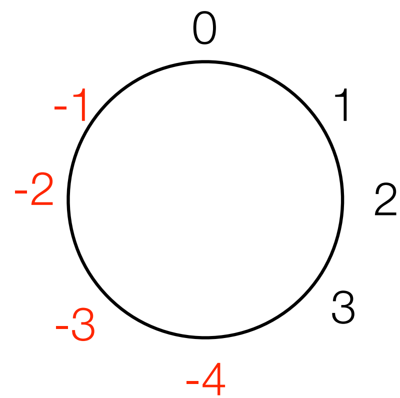
"Pinwheel" representation: move clockwise on the pinwheel for + numbers, counter-clockwise for - numbers

This pinwheel is for mod 8

Modular Arithmetic with negative representations

- For X mod Y, we usually define the remainder as a value between 0 and Y-1
- Alternative: define remainder between -Y/2 and Y/2-1
- e.g., mod 8 would use remainder values -4 through 3:
  - 4 mod 8 = -4 (4 = 8*1 + (-4))
  - 7 mod 8 = -1 (7 = 8*1 + (-1)
  - -5 mod 8 = 3 (-5 = 8 * -1 + 3)
  - 30 mod 8 = -2 (30 = 8*4 + (-2))
  - 30 mod 10 = 0 (30 = 10*3 + 0)

Important for 2's complement (discussed later)



"Pinwheel" representation: move clockwise on the pinwheel for + numbers, counter-clockwise for - numbers

This pinwheel is for mod 8 (range -4 to 3)

Integer # Formats (with word size restriction)

Suppose word size = k (e.g., k = 3)

- How many different bit-sequences are there with word size k?
  - 2^k (e.g., for k=3, $2^3 = 8$)

000
001
010
011
100
101
110
111

Suppose word size = k (e.g., k = 3)

- How many different bit-sequences are there with word size k?
  - 2^k (e.g., for k=3, $2^3 = 8$)

000
001
010
011
100
101
110
111

  - Can assign each number value to a distinct bit-sequence: can have up to 2^k different number values

A weird but feasible assignment:

word	val
000	15
001	-7
010	42
011	42
100	7
101	9
110	e^7
111	0.004

Unsigned binary numbers (with wordsize restriction)

- Binary numbers represent only non-negative (positive or 0) values
- e.g., wordsize = 4 (computer "thinks" using 4 bits at a time)
  - 0000 = 0
  - 0011 = 3
  - 1011 = 11
  - 1111 = 15
  - Can't represent 16 or larger as unsigned binary with wordsize of 4!!
  - Can't represent negative #'s as unsigned binary

BAA

Binary Addition Algorithm (of unsigned numbers)

- Like regular (base 10) addition algorithm, except:
  - 1+1 = 0 with a carry of 1,   1+1+1 = 1 with carry of 1
  - e.g., wordsize = 5 (all numerical representations restricted to 5 bits)
  - add 11110 and 10101 (30 + 21)

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

BAA

Binary Addition Algorithm (of unsigned numbers)

- Like regular (base 10) addition, except:
  - 1+1 = 0 with a carry of 1,   1+1+1 = 1 with carry of 1
  - e.g., wordsize = 5, add 11110 and 10101 (30 + 21)

$$
\begin{array}{lcccccccl}
\text{Overflow} \rightarrow & 1 & 1 & 1 & & & & & \\
 & & 1 & 1 & 1 & 1 & 0 & \leftarrow & 30 \\
+ & & 1 & 0 & 1 & 0 & 1 & \leftarrow & 21 \\
\hline
 & & 1 & 0 & 0 & 1 & 1 & \leftarrow & 19??
\end{array}
$$

- **Overflow:** when result cannot fit within the wordsize constraint
- e.g., above, the “correct” answer 110011 (=51) requires 6 bits: cannot be represented with only 5 bits in unsigned representation

Modular Arithmetic and Overflow

$$
\begin{array}{rl}
 & 1\ 1\ 1\ 1\ 0 \quad 30 \\
+ & 1\ 0\ 1\ 0\ 1 \quad 21 \\
\hline
 & 1\ 0\ 0\ 1\ 1 \quad 19??
\end{array}
$$

- Suppose we hadn't restricted wordsize, and allowed the solution to use an extra bit
- Solution would have been **1**10011 = 51
- The word size restriction prevented us from including the bit = 32
- The result (19) is off by 32, but is correct mod 32 = mod 2^{wordsize}
- i.e., 19 = 51 mod 2^5
- In other words, arithmetic within fixed wordsize computes the remainder

Negative Numbers

- Given: computer has a fixed wordsize (e.g., 4)
- Q: How to represent both positive and negative #'s when constrained by this fixed wordsize?
    - Note: we need an alternate mapping (from what is used for unsigned binary) of bit sequences to values so that some bit sequences represent negative numbers
- A: There are several ways to do this, some are easier for humans, some for computers…

Negative # Representations:

Signed magnitude

1's Complement

2's Complement

Negative Numbers: Signed Magnitude representation

- highest order bit (b_{k-1}) indicates sign: 0 = positive, 1 = negative
- remaining bits indicate magnitude
  - e.g., 0011 = 3
  - e.g., 1011 = -3
  - e.g., 1000 = 0000 = 0
- positive #'s have same form in both signed magnitude and unsigned
- Easy for humans to interpret, but not easiest form for computers to do addition/subtraction operations (as we will soon see…)

Negative Numbers: 1's Complement representation

- Again, highest order bit (b_{k-1}) indicates sign: 0 = positive, 1 = negative
- Non-negative #'s have same representation as unsigned (and signed-mag)
- To negate a #, flip all bits (not just highest-order as in signed-mag)
- Note flipping all bits will also flip the highest order (sign) bit
- e.g., wordsize = 4
    - 0010 = 2
    - 1101 = -2

Negative Numbers: 1's Complement representation

- Non-negative #'s have same representation as unsigned (and signed-mag)
- To negate a #, flip all bits (not just highest-order as in signed-mag)
- e.g., wordsize = 4
  - 0010 = 2
  - 1101 = -2
- Q: suppose wordsize is 8, what is the value of 11101011 when it represents a # in 1's Complement representation?
- A: Let X = 11101011 (it's a negative #)
  - Negate X by flipping all bits: -X = 00010100
  - -X = 20, so X = -20

Negative Numbers: 1's Complement representation

- Non-negative #'s have same representation as unsigned (and signed-mag)
- To negate a #, flip all bits (not just highest-order as in signed-mag)
- e.g., wordsize = 4
  - 0010 = 2
  - 1101 = -2
- Q: suppose wordsize is 8, what is the value of 11101011 when it represents a # in 1's Complement representation?
- A: Let X = 11101011 (it's a negative #)
  - Negate X by flipping all bits: -X = 00010100
  - -X = 20, so X = -20
- NOTE: 2 ways to represent 0 in 1's Complement: all 0's and all 1's
  - e.g., for wordsize of 8, 00000000 and 11111111 are both 0 in 1's complement

Problems with signed mag & 1's C

- Suppose we wanted to have a representation that includes negative numbers where we can apply the **binary addition algorithm** (BAA) to perform the addition:
- 1's complement and signed magnitude won't always work
- E.g., 4-bit word example:

$$\begin{array}{r} 0101 \\ +\ 1101 \\ \hline \end{array}$$

Using binary addition algorithm

Problems with signed mag & 1's C

- Suppose we wanted to have a representation that includes negative numbers where we can apply the **binary addition algorithm** (BAA) to perform the addition:

- 1's complement and signed magnitude won't always work

- E.g., 4-bit word example:

$$\begin{array}{r} {\color{red}1\ \ 1\ \ 1} \\ 0101 \\ +\ 1101 \\ \hline 0010 \end{array}$$

Using binary addition algorithm

Problems with signed mag & 1's C

- Suppose we wanted to have a representation that includes negative numbers where we can apply the **binary addition algorithm** (BAA) to perform the addition:

- 1's complement and signed magnitude won't always work

- E.g., 4-bit word example:

```
 1 1 1
  0101
+ 1101
  ----
  0010
```

Signed magnitude

$$\begin{array}{r} 5 \\ + \ -5 \end{array}$$

Result should be 0, not 2

Using binary addition algorithm (BAA)

Problems with signed mag & 1's C

- Suppose we wanted to have a representation that includes negative numbers where we can apply the binary addition algorithm to perform the addition:

- 1's complement and signed magnitude won't always work

- e.g., 4 bit example:

```
 1 1 1
  0101
+ 1101
  ----
  0010
```

Signed magnitude

$$\begin{array}{r} 5 \\ + \;\; -5 \end{array}$$

Result should be 0, not 2

1's complement

$$\begin{array}{r} 5 \\ + \;\; -2 \end{array}$$

Result should be 3, not 2

Using binary addition algorithm (BAA)

Negative Numbers: 2's complement representation

- Non-negative #'s have same form as unsigned (and signed-mag)
- To negate a #, flip all bits and then (using binary add algorithm) add 1
  - Ignore overflow (will only occur when negating 0)
- e.g., wordsize = 4
  - 0010 = 2 so 1101 + 0001 = 1110 = -2
  - 0011 = 3 so 1100 + 0001 = 1101 = -3
  - 1101 = -3 so 0010 + 0001 = 0011 = 3 (can negate negatives too!)
  - 0000 = 0 so 1111 + 0001 = 0000 = 0 (0 is unique in 2's complement)
- Note: negation works both ways (going + to - or - to +)

  - The exception: 1 followed by all 0's, e.g., 1000
    - value is -2^{k-1} for wordsize of k, e.g., k=4, value is -8
    - Note: the positive value of 2^{k-1} not expressible with k bits in 2's complement form

Number encodings

	Unsigned	Sign&Mag.	1s comp.	2s comp.
0 0 0	0	+0	+0	+0
0 0 1	1	+1	+1	+1
0 1 0	2	+2	+2	+2
0 1 1	3	+3	+3	+3
1 0 0	4	0	-3	-4
1 0 1	5	-1	-2	-3
1 1 0	6	-2	-1	-2
1 1 1	7	-3	0	-1
	8 values	*7 values, 2 zeroes*	*7 values, 2 zeroes*	*8 values, 1 zero*

Number encodings

Signed Mag and 1's complement waste a bit-pattern: 2 reps for 0

	Unsigned	Sign&Mag.	1s comp.	2s comp.
0 0 0	0	+0	+0	+0
0 0 1	1	+1	+1	+1
0 1 0	2	+2	+2	+2
0 1 1	3	+3	+3	+3
1 0 0	4	-0	-3	-4
1 0 1	5	-1	-2	-3
1 1 0	6	-2	-1	-2
1 1 1	7	-3	-0	-1
	8 values	*7 values, 2 zeroes*	*7 values, 2 zeroes*	*8 values, 1 zero*

Number encodings

2’s complement: smallest negative # has no positive counterpart in representation
The flip-bits-and-add-1 process doesn’t successfully negate that number

	Unsigned	Sign&Mag.	1s comp.	2s comp.
0 0 0	0	+0	+0	+0
0 0 1	1	+1	+1	+1
0 1 0	2	+2	+2	+2
0 1 1	3	+3	+3	+3
1 0 0	4	0	-3	-4
1 0 1	5	-1	-2	-3
1 1 0	6	-2	-1	-2
1 1 1	7	-3	0	-1
	8 values	*7 values, 2 zeroes*	*7 values, 2 zeroes*	*8 values, 1 zero*

In 2's complement, the binary addition algorithm always works (unless there is overflow)!

- e.g.,

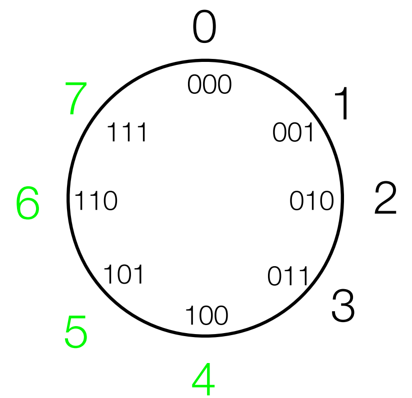
$$
\begin{array}{rrr}
 & {\color{red}1\ 1\ 1} & \\
 & 0101 & 5 \\
+ & 1101 & -3 \\
\hline
 & 0010 & 2
\end{array}
$$

Using binary addition algorithm

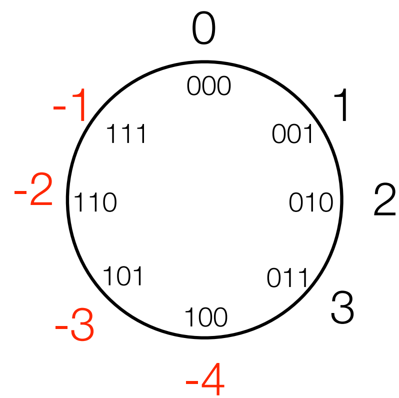
What's special about 2's complement

- Like unsigned, when adding using BAA, results are correct mod 2^k (k is word size)
  - Pinwheel perspective: when incrementing, rotate right, when decrementing, rotate left
  - In either unsigned or 2's C form, will land on value correct Mod 2^k but actual value not correct if not on pinwheel

Unsigned



2's C

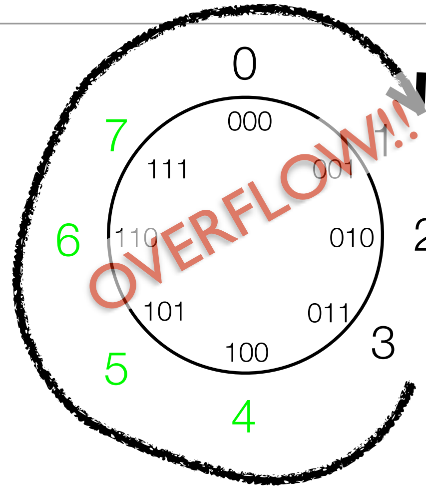


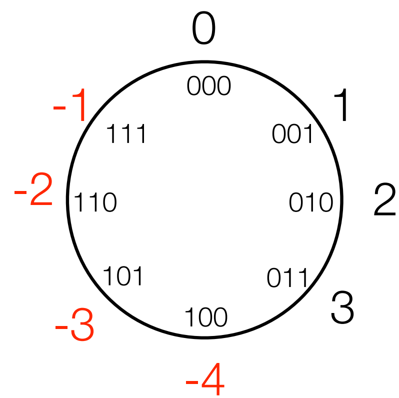
What's special about 2's complement

- Think about modular arithmetic (e.g., mod 8)
  - If we choose "unsigned" mod 8 = (0,1,2,3,4,5,6,7)
    - 3 + 6 mod 8 = 1   (i.e., rem (9/8))
    - 3 - 6 mod 8 = 5

3+6

011
+110





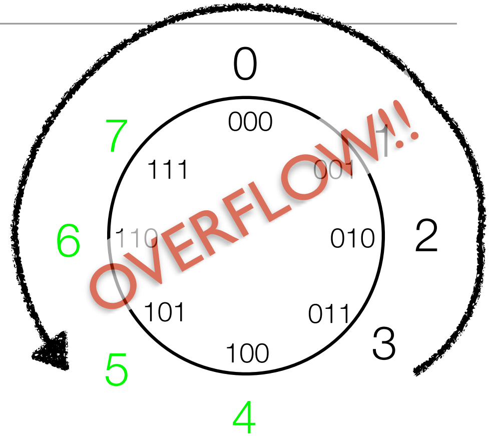
What’s special about 2’s complement

- Think about modular arithmetic (e.g., mod 8)
  - If we choose “unsigned” mod 8 = (0,1,2,3,4,5,6,7)
    - 3 + 6 mod 8 = 1   (i.e., rem (9/8))
    - 3 - 6 mod 8 = 5   (also overflow)

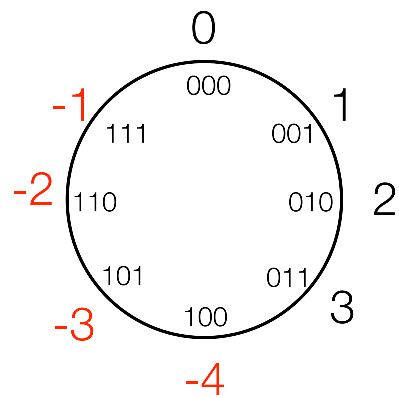
3-6

011

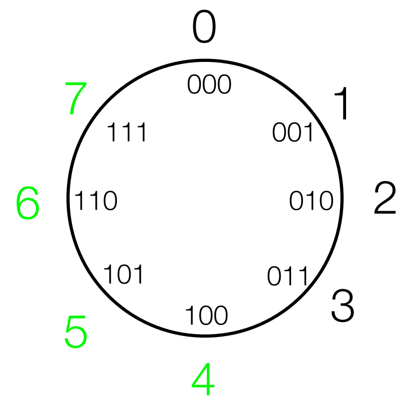
- 110



.

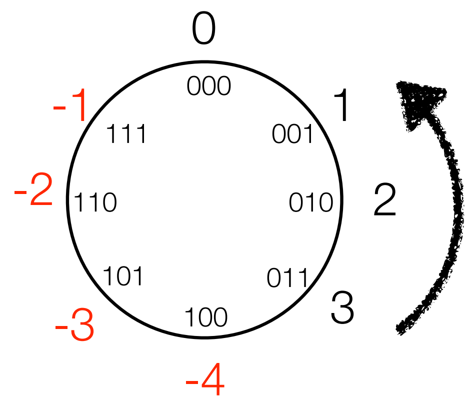


What’s special about 2’s complement

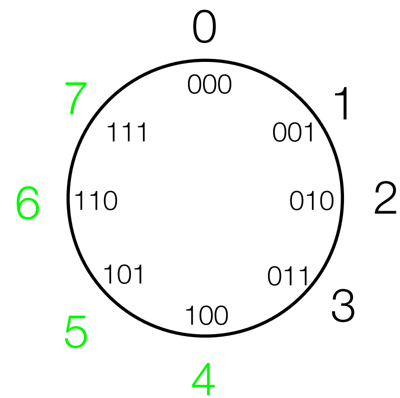


- But could also choose “2’s C” mod 8 = (-4,-3,-2,-1,0,1,2,3)
  - 6 (i.e., 110) would become -2
    - 3 + (-2) mod 8 = 3 - 2 mod 8 = 1
    - 3 - (-2) mod 8 = 3 + 2 mod 8 = -3

011
+110

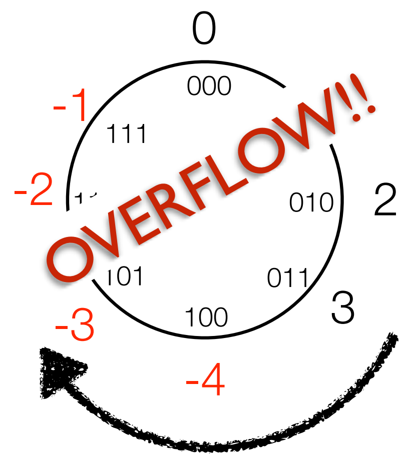


What's special about 2's complement



- But could also choose "2's C" mod 8 = (-4,-3,-2,-1,0,1,2,3)
    - 6 (i.e., 110) would become -2
        - 3 + (-2) mod 8 = 3 - 2 mod 8 = 1
        - 3 - (-2) mod 8 = 3 + 2 mod 8 = -3

011
\- 110



Why Flip all bits +1 for 2C works: intuitive "proof"

- Start with any k-bit word X (other than 100…000) and look at it in 2's complement representation
- Let Y = X with all bits flipped
- X+Y = 11…1111   (using binary add algorithm)
  - Thus, X+Y = -1 in 2's complement rep. (rotated 1 back from 00..0000 on pinwheel)
- Then X + Y + 1 = 00…0000 (using binary add algorithm)
- So Y+1 = -X (and recall Y is X with all bits flipped)

Representation v. Operation

k-bit Words & Ranges of various representations

- Given a k-bit word, the range of numbers can be represented is:
  - unsigned: 0 to $2^k - 1$ (e.g., k=8, 0 to 255)
  - signed mag: $-2^{k-1} + 1$ to $2^{k-1} - 1$ (e.g., k=8, -127 to 127 [2 vals for 0])
  - 1's complement: same as signed mag (but negative numbers are represented differently)
  - 2's complement: -2^{k-1} to $2^{k-1} - 1$ (e.g., k=8, -128 to 127 [1 val for 0])

Getting representation

- Q: Given an 8-bit wordsize, what is the value of 10001011?

Getting representation

- Q: Given an 8-bit wordsize, what is the value of 10001011?
- A: Is it to be represented in Unsigned, Signed Magnitude, 1's complement or 2's complement?
  - Unsigned: 128 + 8 + 2 + 1 = 139
  - Signed Mag: -1 * (8 + 2 + 1) = -11
  - 1's Complement: the negation of 01110100 = -116
  - 2's Complement: the negation of 01110101 = -117
    - Note: for a given set of bits, when # is negative, 2's complement is 1 less than 1's complement

Representation v. Operation

- We have discussed various representations for expressing integers
  - unsigned, signed magnitude, 1's-complement, 2's-complement
- There are also bit-oriented operations that go by the same names
  - 1's-complement: flip all bits
  - 2's-complement: flip all bits and (binary) add 1
- Operation can be performed on a number, regardless of representation
  - e.g., let 10111 be a number in signed-magnitude form (value is -7)
  - 2's complement (operation) on 10111 = 01001 (value is 9 in signed-mag form)
- Observe:
  - 2's-complement operation negates a number when number uses 2's-complement representation
  - 1's-complement operation negates a number when number uses 1's-complement representation

Automating Subtraction

- Q: Why are we interested in 2's-complement when it seems so less intuitive?
- A: much easier to automate subtraction (i.e., add #'s of opposite sign)
  - e.g., wordsize 6, perform 14 - 21 using signed magnitude representation

$$\begin{array}{r} 001110 \\ -\ 010101 \\ \hline \end{array}$$

Automating Subtraction

- Q: Why are we interested in 2‘s-complement when it seems so less intuitive?
- A: much easier to automate subtraction (i.e., add #’s of opposite sign)
    - e.g., wordsize 6, perform 14 - 21 using signed magnitude representation

$$\begin{array}{r} 001110 \\ -\ 010101 \\ \hline \end{array} \longrightarrow - \begin{array}{r} 0\,1\,10 \\ 0\cancel{1}\cancel{0}\cancel{1}01 \\ -\ 001110 \\ \hline 000111 \end{array}$$

- Signed magnitude has lots of potential “gruntwork”
    - e.g., flip top & bottom, “borrow” from higher order bits, etc.

2’s-complement subtraction: make use of BAA

- Just negate subtrahend (bottom # in subtract) and add
- e.g, wordsize 6, perform 14 - 21 using 2’s complement representation

$$
\begin{array}{r}
001110 \\
-\ \underline{010101}
\end{array}
$$

2’s-complement subtraction: make use of BAA

- Just negate subtrahend (bottom # in subtract) and add

- e.g, wordsize 6, perform 14 - 21 using 2’s complement representation

$$\begin{array}{rl} & 001110 \quad (14) \\ - & 010101 \quad (21) \\ \hline \end{array} \xrightarrow[\text{2's complement operation}]{\text{negate subtrahend}} \begin{array}{rl} & 001110 \quad (14) \\ + & 101011 \quad (-21) \\ \hline \end{array}$$

2’s-complement subtraction: make use of BAA

- Just negate subtrahend (bottom # in subtract) and add
- e.g, wordsize 6, perform 14 - 21 using 2’s complement representation

$$
\begin{array}{rrl}
 & 001110 & (14) \\
- & 010101 & (21) \\
\hline
\end{array}
\xrightarrow[\text{2's complement operation}]{\text{negate subtrahend}}
\begin{array}{rrl}
 & \scriptstyle 1\,1\;\;1\;\;\;\; & \\
 & 001110 & (14) \\
+ & 101011 & (-21) \\
\hline
 & 111001 & (-7)
\end{array}
$$

Apply binary addition algorithm

$X = 111001$, $-X = 000111 = 7$, $X=-7$

Detecting Overflow

- Q: How to tell if the result of a computation overflows (i.e., result cannot be expressed within the wordsize constraint)
  - e.g., 4-bit words, unsigned: 1110 + 1010 (14 + 10)
    - result is 24, cannot be expressed in 4-bit unsigned (only values 0-15)
    - Hence, overflows
  - Detection in unsigned easy:
    - Addition: if final carry = 1, then overflow, else not
    - Subtraction: subtrahend is bigger # (result negative), then overflow

Indication of overflow for unsigned

$$\begin{array}{r} 111 \\ 1110 \\ 1010 \\ \hline 1000 \end{array}$$

Overflow detection in 2's-complement is easy

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

MSB
0
1

- If final two carries match, then no overflow
- If differ, then overflow
- e.g., wordsize = 4

5+1=6	-5 + -3 = -8	-2 + 7 = 5	-2 + -7 = -9
0 0	1 1	1 1	1 0
0101	1011	1110	1110
+ 0001	+ 1101	+ 0111	+ 1001
0110	1000	0101	0111
			overflow

7+7=14

0 1
0111
+ 0111
1110

overflow

Remainder of Course: We use unsigned and 2's complement: It's what computers usually use for their representation

Where have you used 2’s complement (w/o knowing it?)

```
int main() {
  int x = 47;
  int y = -50;
  unsigned int z = 50;
  printf(“%d %d %u\n”, x, y, z);
}
```

Where have you used 2’s complement (w/o knowing it?)

```
int main() {
  int x = 47;
  int y = -50;
  unsigned int z = 50;
  printf(“%d %d %u\n”, x, y, z);
}
```

- The computer represents
  - x and y as signed 2’s complement
  - z as an unsigned

Final thoughts on Overflow

- Can't stress enough that overflow means the result cannot be represented within the word size (not necessarily that the result is too big)!

- Example: suppose I choose a (weird) way to map 2-bit words to values as follows:

Two-bit word	Associated Value
00	1
01	3
10	6
11	7

-

Final thoughts on Overflow

- Can't stress enough that overflow means the result cannot be represented within the word size (not necessarily that the result is too big)!

- Example: suppose I choose a (weird) way to map 2-bit words to values as follows:

Two-bit word	Associated Value
00	1
01	3
10	6
11	2

00 + 00 = 11 (i.e., 1 + 1 = 2)

00 + 01 = ??
1 + 3 should equal 4, but no 2-bit combo represents 4

•

Final thoughts on Overflow

- Can't stress enough that overflow means the result cannot be represented within the word size (not necessarily that the result is too big)!

- Example: suppose I choose a (weird) way to map 2-bit words to values as follows:

Two-bit word	Associated Value
00	1
01	3
10	6
11	2

00 + 00 = 11 (i.e., 1 + 1 = 2)

00 + 01 = ??
1 + 3 should equal 4, but no 2-bit combo represents 4

Thus, for this representation, 00+01 causes overflow

Floating Point

Need a bigger range? Floating Point Representation

- Change the encoding.
- Floating point (used to represent very large numbers in a compact way)
  - A lot like scientific notation:

exponent

$$-7.776 \times 10^{3} = -7776 = -6^{5}$$

mantissa
(a.k.a. fraction)

Note: mantissa always in form X.XX...
(one digit before the '.')

  - But for this course, think binary:

$$-1.10 \times 2^{0111} \ (= -1.5 * 2^{7})$$

$$1.01 \times 2^{-0111} \ (= 1.25 * 2^{-7})$$

Note: in proper form, for binary, mantissa always 1.XX...
(one digit before the '.' and digit always a 1)
The only exception: $0 = 0.0 \times 2^{0}$

Standard Forms for Floating Point Numbers

- How to represent a floating point # within the confines of a 32-bit word

- The bits of the word are separated into different fields

31 | 30 29 28 27 26 25 24 23 | 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

Sign	exponent	fraction (mantissa)

- IEEE 754 standard specifies
  - which bits represent which fields (bit 31 is sign, bits 30-23 are 8-bit exponent, bits 22-00 are 23-bit fraction)
  - how to interpret each field

1 1 0 1 0 . 1 1 1 0

2^4 2^3 2^2 2^1 2^0 2^{-1} 2^{-2} 2^{-3}

9.876...

10^{-1} 10^{-2} 10^{-3}

5a) 11010.1110 → 1.10101110 · 2^4

4+127=131

128+2+1

IEEE 754 Floating Point description

31	30 29 28 27 26 25 24 23	22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
Sign	exponent	fraction (mantissa)

- Sign: 0 = positive #, 1 = negative # (like signed magnitude)
- Exponent: unsigned with a bias of 127
  - exponent value = unsigned binary rep of 8 bits - ~~127~~ +127
- Fraction: recall fraction always of form 1.XXXXXX
  - leave off the '1', just represent the "XXXXXXX" part

+127

31	30 29 28 27 26 25 24 23	22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
1	00001101	10100000000000000000000

0 1 0 0 0 0 0 1 1 0 1 0 1 1 1 0 - - — - - 00

- $= (-1) \times 1.101_{(2)} \times 2^{13-127} = -1.625_{(10)} \times 2^{-114} = -7.82409 \times 10^{-35}$

Approximation of 0

- Because representation always of form $\pm 1.XXXX \times 2^{yyyy}$, cannot represent true 0

- Note: All bits set to 0 equals 1.0×2^{-127}, a very very small number, effectively 0

Bias and Comparing floats

- Bias allows exponents between -127 (very small) and 128 (very large)
- Q: Why bias instead of 2's-complement or unsigned magnitude?
- A: Easy comparison between two floats, A & B, for which is larger
  - **Step 1**: Check signs. A+ and B-, return $A>B$. A- and B+, return $A<B$
  - **Step 2** (A and B same sign): Check exponents
    - (A+ and $A.exp > B.exp$) or (A- and $A.exp < B.exp$), return $A>B$
    - (A- and $A.exp > B.exp$) or (A+ and $A.exp < B.exp$), return $A<B$
  - **Step 3** (A and B same sign, same exponents): check fractions
    - (A+ and $A.frac > B.frac$) or (A- and $A.frac < B.frac$), return $A>B$
    - (A- and $A.frac > B.frac$) or (A+ and $A.frac < B.frac$), return $A<B$
  - **Step 4** (A and B same sign, same exponents, same fraction): return $A=B$

- Observation: when bits are ordered sign, exponent, magnitude, process same as comparing 2 signed-magnitude numbers

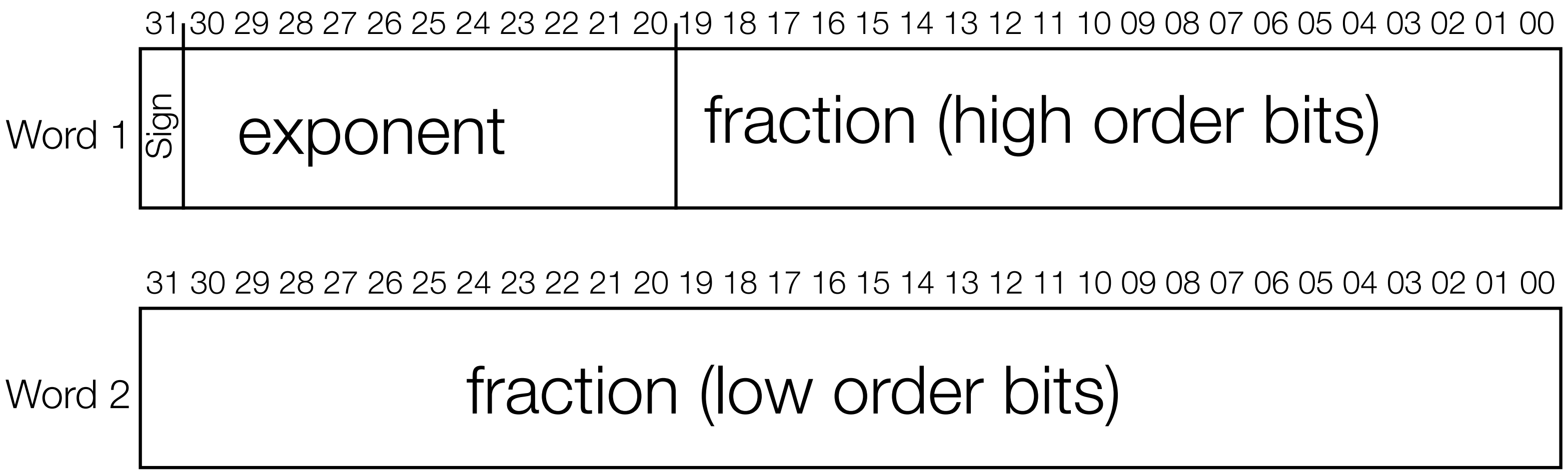
Which IEEE 754 FP # is biggest?

	Bit 31 (sign)	Bits 30–23 (exponent)	Bits 22–00 (fraction)
A	0	00001101	10100000000000000000000
B	0	00101101	00000000000000000000101
C	0	00101101	00000000000000000000011

- All + #s, B > A (larger exponent), B > C (same exponent, larger fraction)
- What if #'s were simply viewed as signed-mag?
  - Same result: B>C>A

IEEE 754 64-bit (double) precision

- Described within 32-bit architecture as two 32-bit words



- 11-bit exponent with bias of 1023
- fraction is 52 bits long (20 higher order bits in Word 1, remaining 32 lower order bits in Word 2

Underflow

- When the magnitude of the value is too small to be described by the representation
- e.g., in IEEE 754 floating point:
  - $X = 1 \times 2^{-100}$ can be represented by the standard (what is the set of bits in the exponent field?)
  - However, X^2, which equals 2^{-200} is too small to be represented
    - smallest possible exponent is -127
  - Hence, attempts to compute X*X results in an underflow

End of Lecture

- Important take-aways for rest of course:
  - High-level view of computer (digital perspective): notion of word size
  - Binary #’s: negative representation (2’s complement)
    - adding, subtracting, overflow, overflow detection