UMBC
AN HONORS UNIVERSITY

CMSC 411

Lecture 4: Representing Operands

- Characters
- Integers
  - Positive numbers
  - Negative numbers
- Non-Integers
  - Fixed-Point Numbers
  - Floating-Point Numbers

Ergun Simsek

Motivation

Computer use binary representation internally

- a wire is “hot” or “cold”
- a switch is “on” or “off”

How do we use bits to represent <u>information</u>?

We need standards of representations for

- Letters and Numbers
- Colors/pixels
- Music and video
- …

Information Encoding

Encoding = assign representation to information

Examples:

- suppose you have two "things" (symbols) to encode
  - one is ➚ and other ➘
  - what would you do?

What about Quantum Computing!!!

- now suppose you have 4 symbols to encode
  - ♒, ♎, ♌ and ⌘
- now suppose you have the following numbers to encode
  - 1, 3, 5 and 7

Encoding is an art

Choosing an appropriate and efficient encoding is a real engineering challenge (and an art)

Impacts design at many levels

- Complexity (how hard to encode/decode)
- Efficiency (#bits used, transmit energy)
- Reliability (what happens with noise?)
- Security (encryption)

Fixed-Length Encodings

What is fixed-length encoding?

- All symbols are encoded using the same number of bits

When to use it?

- If all symbols are equally likely (or we have no reason to expect otherwise)

When not to use it?

- When some symbols are more likely, while some are rare
- What to use then: Variable-length encoding
- Example:
  - Suppose probabilities using an getting an X, Y, or Z are 55%, 30%,and 15%
  - How would we encode them?

Fixed-Length Encodings

Length of a fixed-length code

- use as many bits as needed to unambiguously represent all symbols
  - 1 bit suffices for 2 symbols
  - 2 bits suffice for …?
  - n bits suffice for …?
  - how many bits needed for M symbols?

ceil

$$\lceil \log_2(M) \rceil$$

- ex. Decimal digits 10 = {0,1,2,3,4,5,6,7,8,9}
  - 4-bit binary code: 0000 to 1001

$$\lceil \log_2(10) \rceil = \lceil 3.322 \rceil = 4 \text{ bits}$$

- ex. ~84 English characters = {A-Z (26), a-z (26), 0-9 (10), punctuation (8), math (9), financial (5)}
  - 7-bit ASCII (American Standard Code for Information Interchange)

$$\lceil \log_2(84) \rceil = \lceil 6.39 \rceil = 7 \text{ bits}$$

Encoding Characters

ASCII Code: use 7 bits to encode 128 characters

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	00 0000 0000 NUL	01 0000 0001 SOH	02 0000 0010 STX	03 0000 0011 ETX	04 0000 0100 EOT	05 0000 0101 ENQ	06 0000 0110 ACK	07 0000 0111 BEL	08 0000 1000 BS	09 0000 1001 HT	10 0000 1010 LF	11 0000 1011 VT	12 0000 1100 FF	13 0000 1101 CR	14 0000 1110 SO	15 0000 1111 SI	8
1	16 0001 0000 DLE	17 0001 0001 DC1	18 0001 0010 DC2	19 0001 0011 DC3	20 0001 0100 DC4	21 0001 0101 NAK	22 0001 0110 SYN	23 0001 0111 ETB	24 0001 1000 CAN	25 0001 1001 EM	26 0001 1010 SUB	27 0001 1011 ESC	28 0001 1100 FS	29 0001 1101 GS	30 0001 1110 RS	31 0001 1111 US	9
2	32 0010 0000 SP	33 0010 0001 !	34 0010 0010 "	35 0010 0011 #	36 0010 0100 $	37 0010 0101 %	38 0010 0110 &	39 0010 0111 '	40 0010 1000 (	41 0010 1001)	42 0010 1010 *	43 0010 1011 +	44 0010 1100 ,	45 0010 1101 -	46 0010 1110 .	47 0010 1111 /	A
3	48 0011 0000 0	49 0011 0001 1	50 0011 0010 2	51 0011 0011 3	52 0011 0100 4	53 0011 0101 5	54 0011 0110 6	55 0011 0111 7	56 0011 1000 8	57 0011 1001 9	58 0011 1010 :	59 0011 1011 ;	60 0011 1100 <	61 0011 1101 =	62 0011 1110 >	63 0011 1111 ?	B
4	64 0100 0000 @	65 0100 0001 A	66 0100 0010 B	67 0100 0011 C	68 0100 0100 D	69 0100 0101 E	70 0100 0110 F	71 0100 0111 G	72 0100 1000 H	73 0100 1001 I	74 0100 1010 J	75 0100 1011 K	76 0100 1100 L	77 0100 1101 M	78 0100 1110 N	79 0100 1111 O	C
5	80 0101 0000 P	81 0101 0001 Q	82 0101 0010 R	83 0101 0011 S	84 0101 0100 T	85 0101 0101 U	86 0101 0110 V	87 0101 0111 W	88 0101 1000 X	89 0101 1001 Y	90 0101 1010 Z	91 0101 1011 [	92 0101 1100 \	93 0101 1101]	94 0101 1110 ^	95 0101 1111 _	D
6	96 0110 0000 `	97 0110 0001 a	98 0110 0010 b	99 0110 0011 c	100 0110 0100 d	101 0110 0101 e	102 0110 0110 f	103 0110 0111 g	104 0110 1000 h	105 0110 1001 i	106 0110 1010 j	107 0110 1011 k	108 0110 1100 l	109 0110 1101 m	110 0110 1110 n	111 0110 1111 o	E
7	112 0111 0000 p	113 0111 0001 q	114 0111 0010 r	115 0111 0011 s	116 0111 0100 t	117 0111 0101 u	118 0111 0110 v	119 0111 0111 w	120 0111 1000 x	121 0111 1001 y	122 0111 1010 z	123 0111 1011 {	124 0111 1100 |	125 0111 1101 }	126 0111 1110 ~	127 0111 1111 DEL	F

Encoding More Characters

ASCII is biased towards western languages, esp. English

In fact, many more than 256 chars in common use:

â, ğ, ö, ñ, è, ¥, 揗, 敇, 횝, カ, א, λ, ж, ค

Unicode is a worldwide standard that supports all languages, special characters, classic, and arcane

- Several encoding variants, e.g. 16-bit (UTF-8)

ASCII equiv range:

0	x	x	x	x	x	x	x

16-bit Unicode

1	1	0	y	y	y	y	x	1	0	x	x	x	x	x	x

24-bit Unicode

1	1	1	0	z	z	z	z	1	0	z	y	y	y	y	x	1	0	x	x	x	x	x	x

32-bit Unicode

1	1	1	1	0	w	w	w	1	0	w	w	z	z	z	z	1	0	z	y	y	y	y	x	1	0	x	x	x	x	x	x

Encoding Positive Integers

How to encode positive numbers in binary?

- Each number is a sequence of 0s and 1s
- Each bit is assigned a weight
- Weights are increasing powers of 2, right to left
- The value of an n-bit number is

$$v = \sum_{i=0}^{n-1} 2^i b_i$$

2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	1	1	1	1	0	1	0	0	0	0

$$\begin{array}{rcr} 2^4 & = & 16 \\ + \; 2^6 & = & 64 \\ + \; 2^7 & = & 128 \\ + \; 2^8 & = & 256 \\ + \; 2^9 & = & 512 \\ + \; 2^{10} & = & 1024 \\ \hline & & 2000_{\text{ten}} \end{array}$$

Some Bit Tricks

1. Memorize the first 10 powers of 2

$2^0 = 1$

$2^1 = 2$

$2^2 = 4$

$2^3 = 8$

$2^4 = 16$

$2^5 = 32$

$2^6 = 64$

$2^7 = 128$

$2^8 = 256$

$2^9 = 512$

2. Memorize the prefixes for powers of 2 that are multiples of 10

2^{10} = Kilo (1024)

2^{20} = Mega (1024*1024)

2^{30} = Giga (1024*1024*1024)

2^{40} = Tera (1024*1024*1024*1024)

2^{50} = Peta (1024*1024*1024*1024*1024)

2^{60} = Exa (1024*1024*1024*1024*1024*1024)

Some Bit Tricks (Cont…)

01	0000000011	0000001100	0000101000

3. When you convert a binary number to decimal, first break it down into clusters of 10 bits.
4. Then compute the value of the leftmost remaining bits (1) find the appropriate prefix (e.g. above it should be GIGA) (why? Remember it is kilo, mega, giga, etc.)
5. Compute the value of and add in each remaining 10-bit cluster

Other Helpful Clusterings: Octal

Sometimes convenient to use other number “bases”

- often bases are powers of 2: e.g., 8, 16
  - allows bits to be clustered into groups
- base 8 is called octal ➔ groups of 3 bits
  - Convention: lead the number with a **0**

$$v = \sum_{i=0}^{n-1} 8^i d_i$$

$$\begin{array}{cccccccccccc} 2^{11} & 2^{10} & 2^{9} & 2^{8} & 2^{7} & 2^{6} & 2^{5} & 2^{4} & 2^{3} & 2^{2} & 2^{1} & 2^{0} \\ \end{array}$$

$$\underbrace{0\,1\,1}_{\mathbf{3}}\;\underbrace{1\,1\,1}_{\mathbf{7}}\;\underbrace{0\,1\,0}_{\mathbf{2}}\;\underbrace{0\,0\,0}_{\mathbf{0}} \quad = \mathbf{2000_{10}}$$

Octal - base 8

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

$$\begin{aligned} 0*8^0 &= 0 \\ +\ 2*8^1 &= 16 \\ +\ 7*8^2 &= 448 \\ +\ 3*8^3 &= \underline{1536} \\ & \quad 2000_{10} \end{aligned}$$

$$\mathbf{2000_{10} = 03720_8}$$

One Last Clustering: HEX

Base 16 is most common!

- called hexadecimal or hex ➔ groups of 4 bits
- hex 'digits' ("hexits"): 0-9, and A-F
- each hexit position represents a power of 16
  - ➢ Convention: lead with **0x**

$$v = \sum_{i=0}^{n-1} 16^i d_i$$

Hexadecimal - base 16

0000 - 0	1000 - 8
0001 - 1	1001 - 9
0010 - 2	1010 - a
0011 - 3	1011 - b
0100 - 4	1100 - c
0101 - 5	1101 - d
0110 - 6	1110 - e
0111 - 7	1111 - f

2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	1	1	1	1	0	1	0	0	0	0

$= \mathbf{2000_{10}}$ $= \mathbf{0x7d0_{16}}$

$$\underbrace{0111}_{\mathbf{7}}\ \underbrace{1101}_{\mathbf{d}}\ \underbrace{0000}_{\mathbf{0}}$$

$$\begin{array}{lcr} 0*16^0 & = & 0 \\ +\ 13*16^1 & = & 208 \\ +\ 7*16^2 & = & \underline{1792} \\ & & 2000_{10} \end{array}$$

Signed-Number Representations

What about <u>signed</u> numbers?

- one obvious idea: use an extra bit to encode the sign
  - convention: the most significant bit (leftmost) is used for the sign
  - called the SIGNED MAGNITUDE representation

$$v = -1^S \sum_{i=0}^{n-2} 2^i b_i$$

S	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	1	1	1	1	0	1	0	0	0	0

2000 **-2000**

Signed-Number Representations

The Good: Easy to negate, find absolute value

The Bad:

- add/subtract is complicated
  - depends on the signs
  - 4 different cases!
- two different ways of representing a 0
- it is not used that frequently in practice
  - except in floating-point numbers

Alternative: 2's Complement Rep.

N bits

-2^{N-1}	2^{N-2}	•••	•••	•••	2^3	2^2	2^1	2^0

sign bit

Range: $-\,2^{N-1}$ to $2^{N-1} - 1$

The 2's complement representation for signed integers is the most commonly used signed-integer representation. It is a simple modification of unsigned integers where the most significant bit is considered negative.

$$v = -2^{n-1} b_{n-1} + \sum_{i=0}^{n-2} 2^i b_i$$

chute

ladders

8-bit 2's complement example:

$$11010110 = -2^7 + 2^6 + 2^4 + 2^2 + 2^1$$
$$= -\,128 + 64 + 16 + 4 + 2 = -\,42$$

Why 2's Complement?

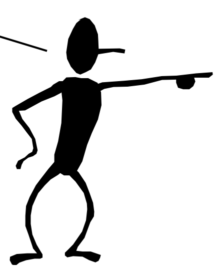
Benefit: the same binary addition (mod 2^n) procedure will work for adding positive and negative numbers

- Don't need separate subtraction rules!
- The same procedure will also handle unsigned numbers!
- NOTE: We typically ignore the leftmost carry

Example:

$$
\begin{array}{rcr}
55_{10} & = & 00110111_2 \\
+\ 10_{10} & = & 00001010_2 \\
\hline
65_{10} & = & 01000001_2
\end{array}
$$

When using signed magnitude representations, adding a negative value really means to subtract a positive value. However, in 2's complement, adding is adding regardless of sign. In fact, you NEVER need to subtract when you use a 2's complement representation.



$$
\begin{array}{rcr}
55_{10} & = & 00110111_2 \\
+\ -10_{10} & = & 11110110_2 \\
\hline
45_{10} & = & \cancel{1}00101101_2
\end{array}
$$

2's Complement

How to negate a number?

- First complement every bit (i.e. $1 \rightarrow 0$, $0 \rightarrow 1$), then add 1
  - 4-bit example (Convert + number to a – and vice versa)

    +5 = 0101 ➔ -5 = 1010 + 1 = 1011 = 1+2-8

    -5 = 1011 ➔ +5 = 0100 + 1 = 0101 = 1+4

  - 8-bit example (Convert + number to a – and vice versa)

    +20 = 00010100 ➔ -20 = 11101011 + 1 = 11101100

    -20 = 11101100 ➔ +20 = 00010011 + 1 = 00010100

- Why does this work?
  - Proof on board. Hint: $1111_2 = 1 + 2 + 4 + 8 = 16 - 1 = 2^4 - 1$

$$\sum_{i=0}^{n-1} 2^i b_i = 2^n - 1 \qquad \text{for } b_i = 1$$

2’s Complement

How to negate a number?

- Method:
  - Complement every bit
  - Add 1 to LSB

- Shortcut
  - Keep the rightmost “1” and any following “0”s as they are
  - Complement all remaining bits
  - Example: 1001000 → 0111000

2's Complement

Sign-Extension

- suppose you have an 8-bit number that needs to be "extended" to 16 bits
  - Why? Maybe because we are adding it to a 16-bit number…

- Examples
  - 16-bit version of 42 = 0000 0000 0010 1010
  - 8-bit version of -2 = 1111 1111 1111 1110

- Why does this work?
  - Same hint:
  - Proof on board

$$\sum_{i=0}^{n-1} 2^i b_i = 2^n - 1$$

Tutorial on Base Conversion (+ve ints)

Binary to Decimal

- multiply each bit by its positional power of 2
- add them together

$$v = \sum_{i=0}^{n-1} 2^i b_i$$

Decimal to Binary

- Problem: given ***v***, find b_i (inverse problem)
- Hint: expand series

$$v = \sum_{i=0}^{n-1} 2^i b_i$$

$$v = b_0 + 2b_1 + 4b_2 + 8b_3 + \ldots + 2^{n-1} b_{n-1}$$

- observe: every term is even except first
  - this determines b_0
  - divide both sides by 2

$$v \;\text{div}\; 2 = b_1 + 2b_2 + 4b_3 + \ldots + 2^{n-2} b_{n-1} \quad \text{(quotient)}$$

$$v \;\text{mod}\; 2 = b_0 \quad \text{(remainder)}$$

Tutorial on Base Conversion (+ve ints)

Decimal to Binary

- Problem: given $\boldsymbol{v}$, find $\boldsymbol{b_i}$ (inverse problem)

$$v = b_0 + 2b_1 + 4b_2 + 8b_3 + \ldots + 2^{n-1} b_{n-1}$$

- Algorithm:
  - Repeat
    - divide $\boldsymbol{v}$ by 2
    - remainder becomes the next bit, $\boldsymbol{b_i}$
    - quotient becomes the next $\boldsymbol{v}$
  - Until $\boldsymbol{v}$ equals 0

Note: Same algorithm applies to other number bases

- just replace divide-by-2 by divide-by-$\boldsymbol{n}$ for base $\boldsymbol{n}$

Non-Integral Numbers

How about non-integers?

- examples
  - 1.234
  - -567.34
  - 0.00001
  - 0.0000000000000012
- fixed-point representation
- floating-point representation

Fixed-Point Representation

Set a definite position for the "binary" point

- everything to its left is the integral part of the number
- everything to its right is the fractional part of the number

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

1101.0110 $= 2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3}$

$= 8 + 4 + 1 + 0.25 + 0.125$

$= 13.375$

Or

1101.0110 $= 214 * 2^{-4} = 214/16 = 13.375$

Fixed-Point Base Conversion

Binary to Decimal

$$v = \sum_{i=-1}^{-m} 2^i b_i$$

- multiply each bit by its positional power of 2
- just that the powers of 2 are now negative
- for m fractional bits

$$v = \frac{b_{-1}}{2} + \frac{b_{-2}}{4} + \frac{b_{-3}}{8} + \frac{b_{-4}}{16} + \ldots + \frac{b_{-m}}{2^m}$$

$$v = 2^{-1} b_{-1} + 2^{-2} b_{-2} + 2^{-3} b_{-3} + 2^{-4} b_{-4} + \ldots + 2^{-m} b_{-m}$$

Examples

- 0.1_2 = ½ = 0.5_{ten}
- 0.0011_2 = 1/8 + 1/16 = 0.1875_{ten}
- 0.001100110011_2= 1/8+1/16+1/128+1/256+1/2048+1/4096 = 0.19995117187_{ten} (getting close to 0.2)
- 0.0011_2 (repeats) = 0.2_{ten}

Fixed-Point Base Conversion

Decimal to Binary

- Problem: given $\boldsymbol{v}$, find $\boldsymbol{b_i}$ (inverse problem)

$$v = 2^{-1}b_{-1} + 2^{-2}b_{-2} + 2^{-3}b_{-3} + 2^{-4}b_{-4} + \ldots + 2^{-m}b_{-m}$$

- Hint: this time, try multiplying by 2

$$2v = b_{-1} + 2^{-1}b_{-2} + 2^{-2}b_{-3} + 2^{-3}b_{-4} + \ldots + 2^{-m+1}b_{-m}$$

  - whole number part is $\boldsymbol{b_{-1}}$
  - remaining fractional part is the rest

- Algorithm:
  - Repeat
    - multiply $\boldsymbol{v}$ by 2
    - whole part becomes the next bit, $\boldsymbol{b_i}$
    - remaining fractional part becomes the next $\boldsymbol{v}$
  - Until ($\boldsymbol{v}$ equals 0) or (desired accuracy is achieved)

Repeated Binary Fractions

Not all fractions have a finite representation

- e.g., in decimal, 1/3 = 0.3333333… (unending)

In binary, many of the fractions you are used to have an infinite representation!

- Examples
  - $1/10 = 0.1_{10} = 0.000110011\ldots_2 = 0.0\underline{\mathit{0011}}_2$
  - $1/5 = 0.2_{10} = 0.\underline{\mathit{0011}}_2 = 0.333\ldots_{16}$

Question

- In Decimal: When do fractions repeat?
  - when the denominator is mutually prime w.r.t. 5 and 2
- In Binary: When do fractions repeat?
  - when the denominator is mutually prime w.r.t. 2
  - i.e., when denominator is anything other than a power of 2

Signed fixed-point numbers

How do you incorporate a sign?

- use sign magnitude representation
  - an extra bit (leftmost) stores the sign
  - just as in negative integers
- 2's complement
  - leftmost bit has a negative coefficient

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

$$1101.0110 = -2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3}$$
$$= -8 + 4 + 1 + 0.25 + 0.125 = -2.625$$

  - OR:
    - first ignore the binary point, use 2's complement, put the point back

$$1101.0110 = (-128 + 64 + 16 + 4 + 2) * 2^{-4}$$
$$= -42/16 = -2.625$$

Signed fixed-point numbers

How to negate in 2’s complement representation

- Same idea: flip all the bits, and add “1” to the rightmost bit
  - **not the bit to the left of the binary point**

- Example

$$1101.0110 = -2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3}$$
$$= -8 + 4 + 1 + 0.25 + 0.125 = -2.625$$

$$1101.0110 \rightarrow 0010.1001 + 0.0001 = 0010.1010$$
$$0010.1010 = 2^1 + 2^{-1} + 2^{-3}$$
$$= 2 + 0.5 + 0.125 = 2.625$$

Bias Notation

Idea: add a large number to everything, to make everything look positive!

- must subtract this "bias" from every representation
- This representation is called "Bias Notation".

$$v = \sum_{i=0}^{n-1} 2^i b_i - Bias$$

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	0	1	1	0

Ex: (Bias = 127)

$$\begin{array}{rcr} 6 * 1 & = & 6 \\ 13 * 16 & = & 208 \\ & & -\ 127 \\ \hline & & 87 \end{array}$$

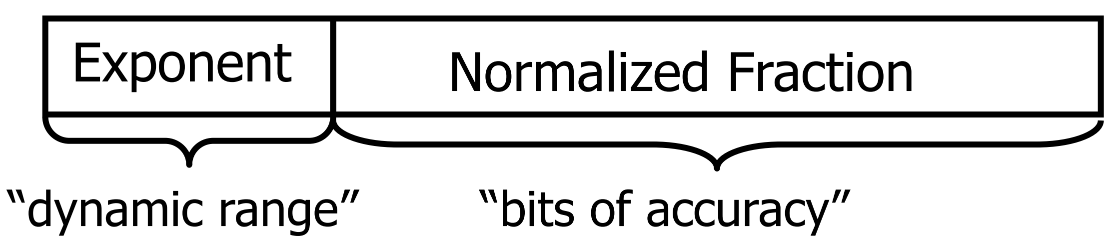
Why? Monotonicity

Floating-Point Representation

Another way to represent numbers is to use a notation similar to Scientific Notation.

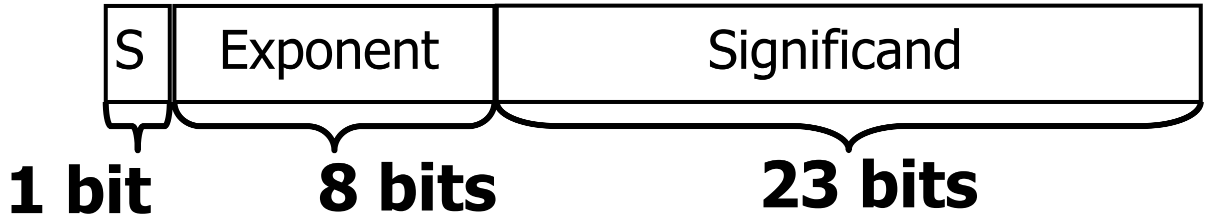
- This format can be used to represent numbers with fractions (3.90 x 10^{-4}), very small numbers (1.60 x 10^{-19}), and large numbers (6.02 x 10^{23}).
- This notation uses two fields to represent each number. The first part represents a normalized fraction (called the significand), and the second part represents the exponent (i.e. the position of the "floating" binary point).

$$Normalized \quad Fraction \times 2^{Exponent}$$



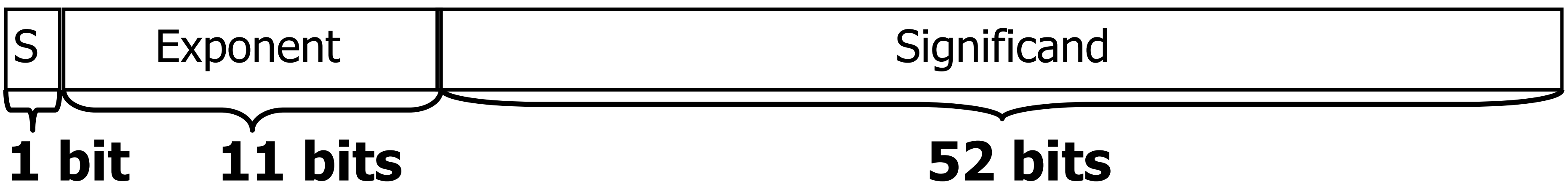
IEEE 754 Floating-Point Formats

Single-precision format



$$v = -1^{S} \times 1.Significand \times 2^{Exponent-127}$$

Double-precision format



$$v = -1^{S} \times 1.Significand \times 2^{Exponent-1023}$$

IEEE 754 Single-precision Example

$0.1_{10} = 0.00011001100110011001100110011001100_2$

$0.00011001100110011001100110011001100 = 1.10011001100110011001100110011001100 \times 2^{-4}$

Exponent = -4 + 127 = 123

Sign bit = 0

Mantissa = 10011001100110011001100

S (1 bit)	Exponent (8 bits)	Mantissa (23 bits)
0	01111011	10011001100110011001100

In Closing

Selecting encoding scheme has imp. implications

- how this information can be processed
- how much space it requires.

Computer arithmetic is constrained by finite encoding

- Advantage: it allows for complement arithmetic
- Disadvantage: it allows for overflows, numbers too big or small to be represented

Bit patterns can be interpreted in an endless number of ways, however important standards do exist

- Two's complement
- IEEE 754 floating point

Next Class

- von Neumann model of a computer
- Instruction set architecture
- MIPS instruction formats
- Some MIPS instructions