

Computer Organization and Design

Representing Operands

Henry Fuchs

Slides adapted from Montek Singh, who adapted them
from Leonard McMillan and from Gary Bishop
Back to McMillan & Chris Terman, MIT 6.004 1999

Tuesday, Jan. 20, 2015

Representing Operands

* Characters

* Integers

- Positive numbers
- Negative numbers

* Non-Integers

- Fixed-Point Numbers
- Floating-Point Numbers

* Reading:

- Chapter 2.3-2.4
- Chapter 3.5 (only through pg. 249)

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 information?

- * We need standards of representations for

- Letters
- Numbers
- Colors/pixels
- Music
- 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?
 - now suppose you have 4 symbols to encode
 - ☰, ☱, ☲ and ☳
 - what would you do?
 - now suppose you have the following numbers to encode
 - 1, 3, 5 and 7
 - what would you do?

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 X is twice as likely as Y or Z
 - 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? $\lceil \log_2(M) \rceil$
- ex. Decimal digits $10 = \{0,1,2,3,4,5,6,7,8,9\}$
 - 4-bit binary code: 0000 to 1001 $\lceil \log_2(10) \rceil = \lceil 3.322 \rceil = 4$ 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) $\lceil \log_2(84) \rceil = \lceil 6.39 \rceil = 7$ 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]	01 [0000 0001]	02 [0000 0010]	03 [0000 0011]	04 [0000 0100]	05 [0000 0101]	06 [0000 0110]	07 [0000 0111]	08 [0000 1000]	09 [0000 1001]	10 [0000 1010]	11 [0000 1011]	12 [0000 1100]	13 [0000 1101]	14 [0000 1110]	15 [0000 1111]
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	□	Γ	⊥	└	⤻	☒	✓	݂	݁	݂	=	݁	݁	<	܂	܀
16 [0001 0000]	17 [0001 0001]	18 [0001 0010]	19 [0001 0011]	20 [0001 0100]	21 [0001 0101]	22 [0001 0110]	23 [0001 0111]	24 [0001 1000]	25 [0001 1001]	26 [0001 1010]	27 [0001 1011]	28 [0001 1100]	29 [0001 1101]	30 [0001 1110]	31 [0001 1111]	
DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US	
1	܃	܄	܅	܆	܇	܈	܉	܊	܋	܌	܍	܎	܏	ܐ	ܑ	ܒ
32 [0010 0000]	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]	
2	SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
48 [0011 0000]	49 [0011 0001]	50 [0011 0010]	51 [0011 0011]	52 [0011 0100]	53 [0011 0101]	54 [0011 0110]	55 [0011 0111]	56 [0011 1000]	57 [0011 1001]	58 [0011 1010]	59 [0011 1011]	60 [0011 1100]	61 [0011 1101]	62 [0011 1110]	63 [0011 1111]	
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64 [0100 0000]	65 [0100 0001]	66 [0100 0010]	67 [0100 0011]	68 [0100 0100]	69 [0100 0101]	70 [0100 0110]	71 [0100 0111]	72 [0100 1000]	73 [0100 1001]	74 [0100 1010]	75 [0100 1011]	76 [0100 1100]	77 [0100 1101]	78 [0100 1110]	79 [0100 1111]	
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	܀
80 [0101 0000]	81 [0101 0001]	82 [0101 0010]	83 [0101 0011]	84 [0101 0100]	85 [0101 0101]	86 [0101 0110]	87 [0101 0111]	88 [0101 1000]	89 [0101 1001]	90 [0101 1010]	91 [0101 1011]	92 [0101 1100]	93 [0101 1101]	94 [0101 1110]	95 [0101 1111]	
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
96 [0110 0000]	97 [0110 0001]	98 [0110 0010]	99 [0110 0011]	100 [0110 0100]	101 [0110 0101]	102 [0110 0110]	103 [0110 0111]	104 [0110 1000]	105 [0110 1001]	106 [0110 1010]	107 [0110 1011]	108 [0110 1100]	109 [0110 1101]	110 [0110 1110]	111 [0110 1111]	
6	ܔ	a	b	c	d	e	f	g	h	i	j	k	l	m	n	܀
112 [0111 0000]	113 [0111 0001]	114 [0111 0010]	115 [0111 0011]	116 [0111 0100]	117 [0111 0101]	118 [0111 0110]	119 [0111 0111]	120 [0111 1000]	121 [0111 1001]	122 [0111 1010]	123 [0111 1011]	124 [0111 1100]	125 [0111 1101]	126 [0111 1110]	127 [0111 1111]	
7	p	q	r	s	t	u	v	w	x	y	z	{	}	~	DEL	

Encoding More Characters

9 / 35

- * ASCII is biased towards western languages, esp. English
- * In fact, many more than 256 chars in common use:
â, m, ö, ñ, è, ¥, 握, 敕, 訇, 力, 𣓃, 𣓄, 𣓅, 𣓆
- * 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:

0xxxxxx

16-bit Unicode

110yyyyx

10xxxxxx

24-bit Unicode

1110zzzz

10zyyyyx

10xxxxxx

32-bit Unicode

11110www

10wwzzzz

10zyyyyx

10xxxxxx

Encoding Positive Integers

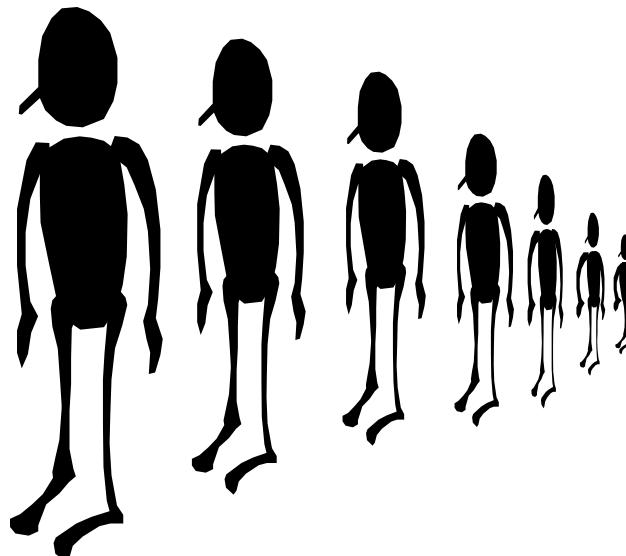
10 / 35

* 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



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

Some Bit Tricks

- * Get used to working in binary

- Specifically for Comp 411, but it will be helpful throughout your career as a computer scientist

- * Here are some helpful guides

1. Memorize the first 10 powers of 2

$$2^0 = 1$$

$$2^5 = 32$$

$$2^1 = 2$$

$$2^6 = 64$$

$$2^2 = 4$$

$$2^7 = 128$$

$$2^3 = 8$$

$$2^8 = 256$$

$$2^4 = 16$$

$$2^9 = 512$$

More Tricks with Bits

12 / 35

- * Get used to working in binary
 - * Here are some helpful guides
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 ($1024 * 1024 * 1024 * 1024 * 1024 * 1024$)

For fun:

<http://highscalability.com/blog/2012/9/11/how-big-is-a-petabyte-exabyte-zettabyte-or-a-yottabyte.html>

Even More Tricks with Bits

- * Get used to working in binary
- * Here are some helpful guides

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 (GIGA) (Often this is sufficient)
5. Compute the value of and add in each remaining 10-bit cluster

Other Helpful Clusterings

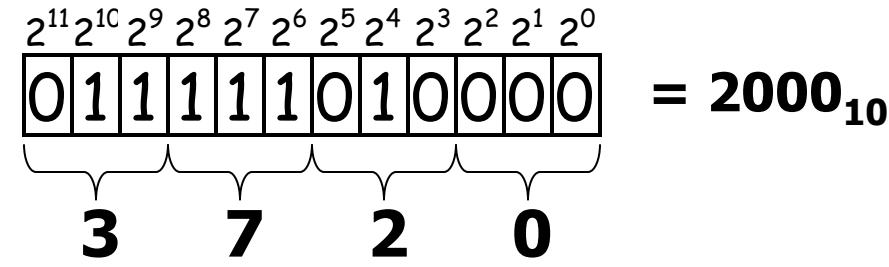
14 / 35

* 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**

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

03720



000 - 0	$0 * 8^0 = 0$
001 - 1	$+ 2 * 8^1 = 16$
010 - 2	$+ 7 * 8^2 = 448$
011 - 3	$+ 3 * 8^3 = \underline{1536}$
100 - 4	
101 - 5	
110 - 6	
111 - 7	

2000₁₀

One Last Clustering

* Base 16 is most common!

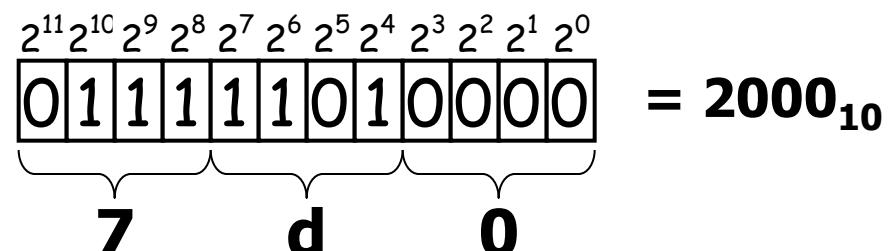
- called **hexadecimal** or **hex** → groups of 4 bits
- hex 'digits' ("hexits"): 0-9, and A-F
- each hexit position represents a power of 16
 - Convention: lead with **0x**

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

0x7d0

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



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

Signed-Number Representations

16 / 35

* What about signed 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					

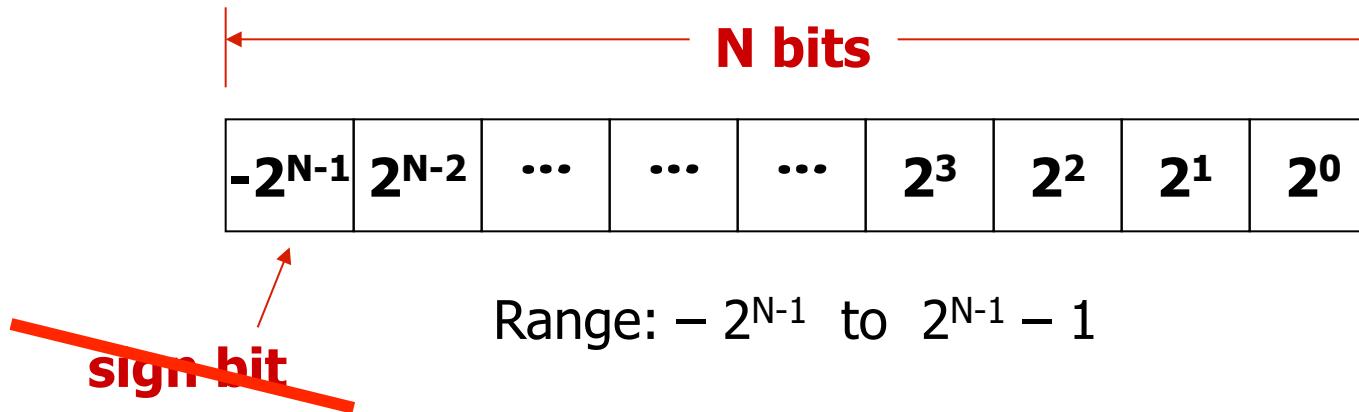
Signed-Number Representations

17 / 35

- * 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.

18 / 35



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**.

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

8-bit 2's complement example:

$$\begin{aligned}11010110 &= -2^7 + 2^6 + 2^4 + 2^2 + 2^1 \\&= -128 + 64 + 16 + 4 + 2 = -42\end{aligned}$$

Why 2's Complement?

19 / 35

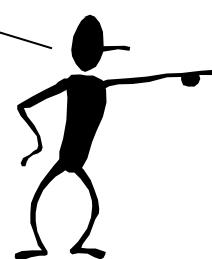
* Benefit: the same binary addition ($\text{mod } 2^n$) procedure will work for adding positive and negative numbers

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

Example:

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.

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


$$\begin{array}{r} 55_{10} = 00110111_2 \\ + -10_{10} = 11110110_2 \\ \hline 45_{10} = 100101101_2 \end{array}$$

2's Complement

20 / 35

* How to negate a 2's complement number

- First complement every bit (i.e. $1 \rightarrow 0, 0 \rightarrow 1$), then add 1 (memorize this!)
 - Example: $20 = 00010100, -20 = 11101011 + 1 = 11101100$
 - Example: $-20 = 11101100, 20 = 00010011 + 1 = 00010100$
- Same negation works for positive or negative numbers
- Remember: 2's complement is representation for both positive and negative numbers; can write in Hex, Octal,..

$$1 + 2 + 4 + 8 = 16 - 1$$

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

- Why does this work?
 - You figure it out! Hint:

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**

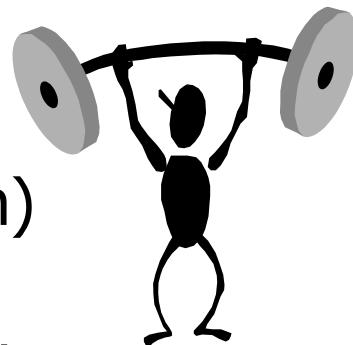


- Why does this work?

- Same hint:

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

10's-complement Arithmetic (You'll never need to borrow again)



Helpful Table of the
9's complement for
each digit

0 → 9
1 → 8
2 → 7
3 → 6
4 → 5
5 → 4
6 → 3
7 → 2
8 → 1
9 → 0

- Step 1) Write down two 3-digit numbers (you want to subtract the second from the first)
- Step 2) Form the 9's-complement of each digit in the second number (the subtrahend)
- Step 3) Add 1 to it (the second number, the subtrahend)
- Step 4) Add this number to the first
- Step 5) If your result was less than 1000,
form the 9's complement again and add 1
and remember your result is negative
else
subtract 1000

What did you get? Why weren't you taught to subtract this way?

* 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

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

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

➤ observe: every term is even except first

- this determines b_0
- divide both sides by 2

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

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

* Decimal to Binary

- Problem: given v , find b_i (inverse problem)

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

- Algorithm:

➤ Repeat

- divide v by 2
- remainder becomes the next bit, b_i
- quotient becomes the next v

➤ Until v equals 0

* Note: Same algorithm applies to other number bases

- just replace divide-by-2 by divide-by- n for base n

Non-Integral Numbers

* How about non-integers?

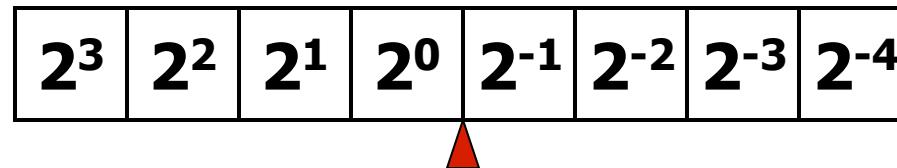
- examples
 - 1.234
 - -567.34
 - 0.00001
 - 0.000000000000012
- fixed-point representation
- floating-point representation

Fixed-Point Representation

26 / 35

* 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



$$\begin{aligned}1101.0110 &= 2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3} \\&= 8 + 4 + 1 + 0.25 + 0.125 \\&= 13.375\end{aligned}$$

Or

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

Fixed-Point Base Conversion

27 / 35

* Binary to Decimal

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

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

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

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

* Examples

- $0.1_2 = \frac{1}{2} = 0.5_{\text{ten}}$
- $0.0011_2 = 1/8 + 1/16 = 0.1875_{\text{ten}}$
- $0.001100110011_2 = 1/8 + 1/16 + 1/128 + 1/256 + 1/2048 + 1/4096 = 0.19995117187_{\text{ten}}$ (getting close to 0.2)
- $0.\overline{0011}_2$ (repeats) = 0.2_{ten}

* Decimal to Binary

- Problem: given v , find b_i (inverse problem)

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

- Hint: this time, try multiplying by 2

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

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

- Algorithm:

- Repeat
 - multiply v by 2
 - whole part becomes the next bit, b_i
 - remaining fractional part becomes the next v

- Until (v equals 0) or (desired accuracy is achieved)

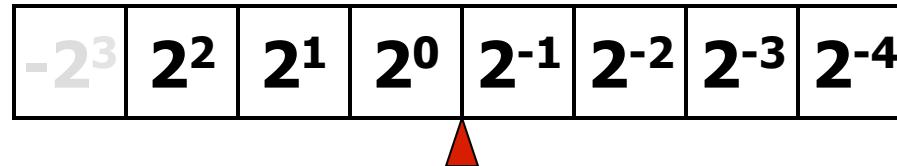
Repeated Binary Fractions

- * Not all fractions have a finite representation
 - e.g., in decimal, $1/3 = 0.333333\dots$ (unending)
- * In binary, many of the fractions you are used to have an infinite representation!
 - Examples
 - $1/10 = 0.1_{10} = 0.000110011\dots_2 = 0.\overline{0011}_2$
 - $1/5 = 0.2_{10} = 0.\overline{0011}_2 = 0.333\dots_{16}$
- * 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



$$\begin{aligned}
 1101.0110 &= -2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3} \\
 &= -8 + 4 + 1 + 0.25 + 0.125 = -2.625
 \end{aligned}$$

- OR:
 - first ignore the binary point, use 2's complement, put the point back
- $$\begin{aligned}
 1101.0110 &= (-128 + 64 + 16 + 4 + 2) * 2^{-4} \\
 &= -42/16 = -2.625
 \end{aligned}$$

Signed fixed-point numbers

31 / 35

* 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

$$\begin{aligned}1101.0110 &= -2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3} \\&= -8 + 4 + 1 + 0.25 + 0.125 = -2.625\end{aligned}$$

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

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)

Why? Monotonicity

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

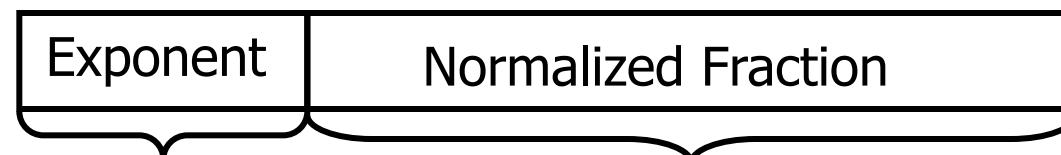
Floating-Point Representation

33 / 35

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

- This format can be used to represent numbers with fractions (3.90×10^{-4}), very small numbers (1.60×10^{-19}), and large numbers (6.02×10^{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 Fraction $\times 2^{\text{Exponent}}$



“dynamic range”

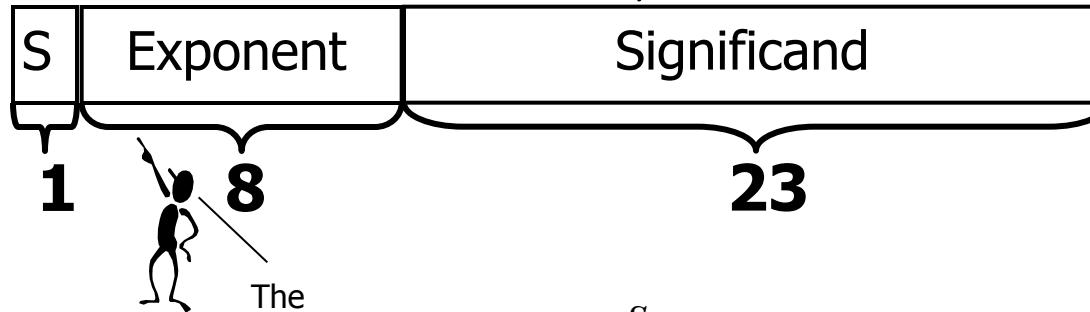
“bits of accuracy”

IEEE 754 Format

Single-precision format



This is effectively a signed magnitude fixed-point number with a “hidden” 1.

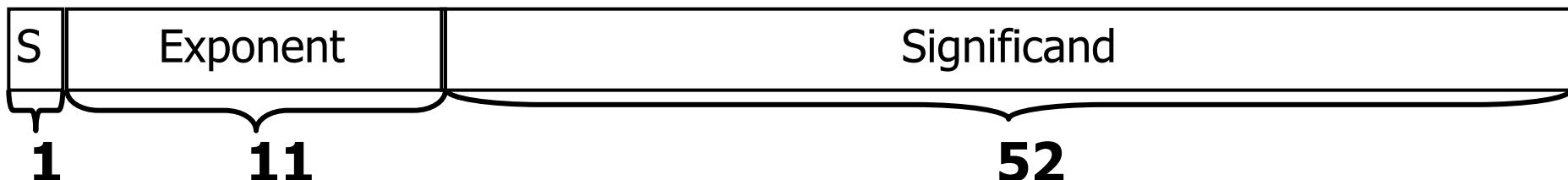


The exponent is represented in bias 127 notation.
Why?



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

Double-precision format



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

The 1 is hidden because it provides no information after the number is “normalized”

- * Selecting encoding scheme has implementation 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