Data Representation Cont.

Bit Patterns from N Bits

Number of Bits	Number of Patterns	Number of Patterns as Power of Two
1	2	2^1
2	4	2^2
3	8	2^3
4	16	2^4

- Number of possible patterns with N bits $= 2^N$
- How many patterns can be formed with
  - 10 bits? $= 2^{10} = 1024$
  - 20 bits? $= 2^{20} = 2^{10} * 2^{10} = 1048576$
  - 30 bits? $= 2^{30} = 2^{10} * 2^{20} = 1073741824$
  - 40 bits? $= 2^{40} = 2^{10} * 2^{30} = 1.0995116e+12$
  - 50 bits? $= 2^{50} = 2^{10} * 2^{40} = 1.1258999e+15$
  - 60 bits? $= 2^{60} = 2^{10} * 2^{50} = 1.1529215e+18$

Unsigned Integers Overview

- All bits represent magnitude

0	1	1	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	1	0	0	1	0	0	1	1	1	1	1	0	1

magnitude

- Can represent range $[0, 2^n - 1]$
- What range of values can be represented for a 8-bit unsigned integer?
  - $[0, 2^8-1]$
  - $[0, 255]$
- What ranges of values can be represented by an 32-bit unsigned int?
  - $[0, 2^{32}-1]$
  - $[0, 4294967296]$

Unsigned Integer to Decimal

- Convert unsigned integer to decimal
- Binary number written as $d_{n-1} \ldots d_2 d_1 d_0$ (where n = # of bits)
- The decimal value is $\sum_{i=0}^{n-1} d_i \times 2^i$
- Example:
  - 8-bit unsigned integer

Bits:	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**1**
Indexes:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

  - = $1(2^7) + 0(2^6) + 0(2^5) + 1(2^4) + 0(2^3) + 1(2^2) + 0(2^1) + 1(2^0)$
  - = $2^7 + 2^4 + 2^2 + 2^0$
  - = 128 + 16 + 4 + 1
  - = 149

Signed Integer Overview

- Use the leftmost bit for sign

0	1	1	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	1	0	0	1	0	0	1	1	1	1	1	0	1

sign

magnitude

- Use twos complement to represents negative numbers
  - Take the ones complement and add one
  - Essentially invert the bits and add one
- Can represent the range $[-2^{n-1}, 2^{n-1}-1]$
- What range of values can an 8-bit signed integer represent?
  - $[-2^{8-1}, 2^{8-1}-1]$
  - [-128, 127]
- What range of values can an 32-bit signed integer represent?
  - $[-2^{32-1}, 2^{32-1}-1]$
  - [-2147483648, 2147483647]

Signed Integer to Decimal

- Convert Signed Integer to Decimal
- Binary number written as $d_{n-1}d_{n-2} \ldots d_1 d_0$ (where n = # of bits)
- Decimal value is interpreted as $-d_{n-1}2^{n-1} + \sum_{i=0}^{n-2} d_i 2^i$
  - Works with both positive and negative numbers
- Example 1:
  - 8-bit signed integer

Bits:	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**1**
Indexes:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

  - $= -(1 \times 2^7) + 0(2^6) + 0(2^5) + 1(2^4) + 0(2^3) + 1(2^2) + 0(2^1) + 1(2^0)$
  - $= -(1 \times 2^7) + 1(2^4) + 1(2^2) + 1(2^0)$
  - $= -128 + 16 + 4 + 1$
  - $= -107$

Signed Integer to Decimal (Ex. Cont.)

- Let's confirm by taking taking the negative value of -107 and reevaluating decimal
- Negate -107 using twos complement
  - $-107_{10} = 10010101_2$
  - 01101010_2 (take complement)
  - 01101011_2 (add 1)
- Convert 01101011_2 to decimal
  - If right, it should be 107

Bits:	**0**	**1**	**1**	**0**	**1**	**0**	**1**	**1**
Indexes:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

- $= -(0 \times 2^7) + 1(2^6) + 1(2^5) + 0(2^4) + 1(2^3) + 0(2^2) + 1(2^1) + 1(2^0)$
- $= 2^6 + 2^5 + 2^3 + 2^1 + 2^0$
- $= 64 + 32 + 8 + 2 + 1$
- $= 107$ (correct!)

Floating Point Overview

- Most computers follow IEEE 754 standard
- Bits split up into three sections:

s	exp	mantissa

  - s: sign field determines if the number is negative (s=1 if negative)
  - exp: biased exponent
  - mantissa: fractional number in binary (base 2)
- Decimal Value = $(-1)^S \times 2^E \times F$
  - E : unbiased exponent in decimal
    - E = exp – bias (where k = number exp bits)
    - bias = $(2^{(k-1)}-1)$
    - The bias allows exp to be represented as an unsigned integer for comparison but represent negative exponents
  - F : binary scientific notation
    - F = 1.<mantissa> (or 0.<mantissa>, we'll see later on)

Converting Floating Point to Decimal

- Recall: Decimal Value = $(-1)^S \times 2^E \times F$
- Basic Steps for converting floating point to decimal
  1. Calculate Unbiased Exponent
     - Get E, where E = exp – bias and bias = $2^{(k-1)}-1$
  2. Get binary scientific notation with mantissa
     - Get F, where F = 1.<mantissa>
  3. Shift binary scientific notation ($2^E \times F$)
  4. Convert binary representation to decimal
  5. Tack on sign (multiply by $(-1)^S$)

Example

- Recall: Decimal Value = $(-1)^S \times 2^E \times F$
- Example: 8-bit floating point
  - 1 bit for sign, 3 bits for exponent, 4 bits for mantissa

sign	exp			mantissa			
1	1	1	0	1	0	1	1

1. Calculate unbiased exponent (E, where E = exp - bias)
   - E = exp – bias
   - $E = 110_2 - \text{bias} = 6_{10} - \text{bias}$ (evaluate exp)
   - $E = 6_{10} - (2^{(k-1)}-1) = 6_{10} - (2^{(3-1)}-1) = 6_{10} - 3_{10}$ (evaluate bias)
   - E = 3
2. Get binary scientific notation
   - F = 1.<mantissa> = 1.1011
3. Shift Binary Representation ($2^E \times F$)
   - $2^3 \times 1.1011_2 = 1101.1_2$
4. Evaluate Binary Result To Decimal
5. Tack on Sign (multiply by $(-1)^S$)

1	1	0	1	.	1
3	2	1	0		-1

$= 1(2^3) + 1(2^2) + 0(2^1) + 1(2^0) + 1(2^{-1})$

$= 2^3 + 2^2 + 2^0 + 2^{-1}$

$= 8 + 4 + 1 + 0.5$

$= 13.5$

$= -13.5$ ← Final Result

Other Values in Floating Point

- We just went over how normalized values are represented in floating point
- However two additional kinds of values are represented by floating point representation
  - How we interpret them is different than normalized values
- Denormal Values
  - When exp is all 0s
  - Represents numbers 0 or very close to zero
  - Difference from normalized values:
    - Different Unbiased Exponent (E) = 1 – bias or 1 - $(2^{(k-1)}-1)$
    - Different Binary Scientific Notation (F) = 0.<mantissa>
- Special Values
  - When exp all 1s
  - When mantissa is all 0's
    - Positive or negative Infinity ($\pm\infty$) depending on sign
  - When mantissa is not all 0's
    - NaN = Not a number

Denormal Value Example

- Recall: Decimal Value = $(-1)^S \times 2^E \times F$
- Example: 8-bit floating point
  - 1 bit for sign, 3 bits for exponent, 4 bits for mantissa

0	0	0	0	1	0	1	1
sign	exp			mantissa			

1. Calculate unbiased exponent (E, where E = 1 - bias)
   - $E = 1 - \text{bias}$
   - $E = 1 - (2^{(k-1)}-1) = 1 - (2^{(3-1)}-1) = 1 - 3$ (evaluate bias)
   - $E = -2$
2. Get binary scientific notation
   - F = 0.<mantissa> = 0.1011_2
3. Shift Binary Representation ($2^E \times F$)
   - $2^{-2} \times 0.1011_2 = 0.001011_2$
4. Evaluate Binary Result To Decimal
5. Tack on Sign (multiply by $(-1)^S$)

0	.	0	0	1	0	1	1
0		-1	-2	-3	-4	-5	-6

$= 0(2^{-1}) + 0(2^{-2}) + 1(2^{-3}) + 0(2^{-4}) + 1(2^{-5}) + 1(2^{-6})$

$= 2^{-3} + 2^{-5} + 2^{-6}$

$= 0.125 + 0.03125 + 0.015625$

$= 0.171875$

$= +0.171875$ Final Result

Special Value Examples

- Example 1:

sign	exp			mantissa			
0	1	1	1	0	0	0	0

  - exp is all 1s so it must be a special value
  - mantissa is all 0s and the sign is 0 so positive
  - special value + 0 mantissa + positive value = $+\infty$

- Example 2:

sign	exp			mantissa			
0	1	1	1	0	1	0	0

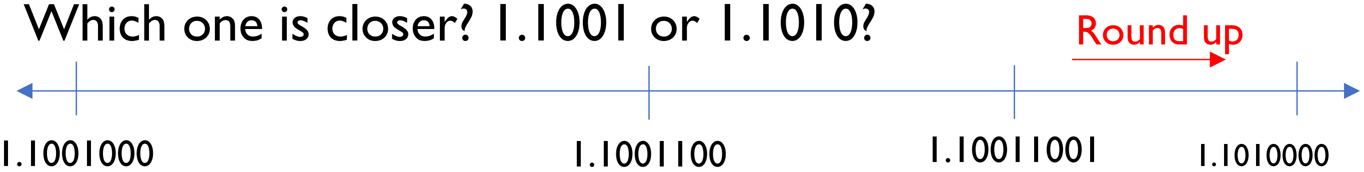
  - exp is all 1s so it must be a special value
  - mantissa is not all zeros
  - special value + non-zero mantissa = NaN

Floating Point Summary

- Three different cases
- Normalized values
  - When exp is not all 0s or not all 1s
  - $E = exp - (2^{(k-1)}-1)$
  - F = 1.<mantissa>
- Denormalized Values
  - When exp is 0
  - $E = 1 - (2^{(k-1)}-1)$ -> (e.g for 32-bit float: 1- 127 = -126)
  - F = 0.<mantissa>
  - Represents 0 and values very close to 0
- Special Values
  - When exp all 1's
  - When mantissa is all 0's
    - Positive or negative Infinity ($\pm\infty$) depending on sign
  - Else when mantissa is not all 0's
    - NaN = Not a number

Rounding in Floating Point

- Round to the nearest number
- Example:
  - Assume 4 bit mantissa
  - 1.10011001
  - Need to trauncate to 4 mantissa bits
  - Which one is closer? 1.1001 or 1.1010?



  - Round up to 1.1010 because it's closer
- What happens if tie?
  - Round to even binary number (where last digit is 0)
- Example:
  - 1.10011
    - If we round down we get an odd number 1.1001
    - So round up to even number 1.1010
  - 1.10001
    - If we round up we get 1.1001 which is not even
    - Round down to even number 1.1000

ASCII

- American Standard for Computer Information Interchange
  - Defines what character is represents by a sequence of bits
- According to ASCII standard, 1 character is stores with 1 byte (8 bits)
- Based on the English Alphabet
- Originally only encoded 128 character using 7 bits
  - One bit could be used for error detection
- Subsequently extended to use all 256 values

ASCII Table

	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	p
1	SOH	DC1 XON	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3 XOFF	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	del

Character value stored in 1 byte

Value of character in Hex

- ‘1’ = 0x31
- ‘3’ = 0x33
- ‘9’ = 0x39
- ‘a’ = 0x61
- ‘A’= 0x41

ASCII Character Representing Integer

- Supppose user types a 4 character sequence "123\n"
- Conversion from character representation to the desired two's complement integer representation
  - Integer desired = ASCII representation - 48

ASCII Character	Hex Value	Decimal Value	Binary	Desired Integer	Two's Complement
'1'	0x31	49	00110001	1	00000001
'2'	0x32	50	00110010	2	00000010
'3'	0x33	51	00110011	3	00000011
'\n'	0x01	10	00001010	(NA)	(NA)

Unicode and UTF-8

- What about characters for other languages?
  - ASCII only allows for a small number of characters
- Unicode is a standard that defines more than 107,000 characters across 90 scripts (and more)
- Most Common: UTF-8
  - Variable length encoding of Unicode: 1-4 bytes for each character
  - 1-byte form is reserved for ASCII backward compatibility

Addressing

- All information is represented in binary form but require different sizes
- Pointer sizes are different depending on the architecture:
  - 32-bit machine: 32-bit pointer = 4 bytes
  - 64-bit machine: 64-bit pointer = 8 bytes
- How many different addresses can a pointer have?
  - 32-bits = 2^{32} bytes = 2^{2}x2^{30} bytes = 4 Gigabytes
  - 64-bits = 2^{64} bytes = 2^{4}x2^{60} bytes = 16 Exabytes
- This is what known as the "Address Space" or space of all memory address

Big Endian vs. Little Endian

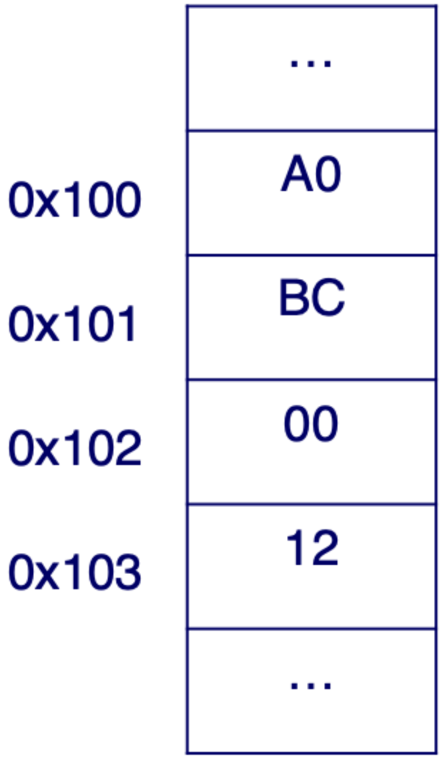
- How to determine value when you have a binary number spread across multiple bytes?

A0	**BC**	**00**	**12**

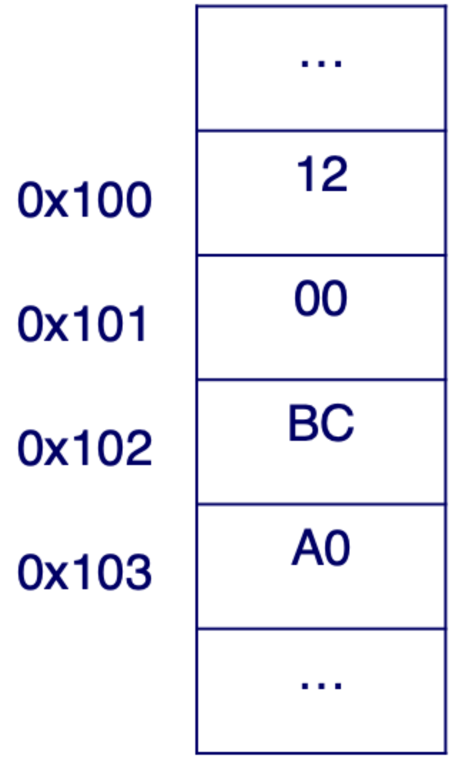
- Is it A0BC0012 or 1200BCA0?
- Big Endian
  - Most significant byte first
  - A0BC0012 in example above
- Little Endian
  - Least significant byte first
  - 1200BCA0 in example above
- Why care?
  - Interpret machine code and values
  - Different computers use different endianness
  - Need to convert into standard form before transmitting

Data in Memory

Integer: 0xA0BC0012



Big Endian



Little Endian