Digital Representation of Information
(unsigned floating-point numbers)

- **003DigitalInfoRepres…n02** sees decimal, binary and hex number systems as *positional* number systems for *unsigned whole numbers*
- These number systems also cover *unsigned floating-point numbers*
- Decimal example: radix point
  - $123.456 = 1\times10^{2} + 2\times10^{1} + 3\times10^{0} + 4\times10^{-1} + 5\times10^{-2} + 6\times10^{-3}$
  - Digits to the *right* of the *decimal point* have associated *negative* powers of **10**
- Binary example: radix point
  - $110.011b = 1\times2^{2} + 1\times2^{1} + 0\times2^{0} + 0\times2^{-1} + 1\times2^{-2} + 1\times2^{-3} = 6.375$
  - Digits to the *right* of the *binary point* have associated *negative* powers of **2**
- Hex example: radix point
  - $1AB.C8h = 1\times16^{2} + 10\times16^{1} + 11\times16^{0} + 12\times16^{-1} + 8\times16^{-2} = 417.\ 78125$
  - Digits to the *right* of the *hex point* have associated *negative* powers of **16**
- More generic term for *decimal/binary/hex* point → *radix* point

Digital Representation of Information
(unsigned floating-point numbers)

- *All but one* of the methods discussed in **003DigitalInfoRep…n02**
  - …for converting representations of unsigned whole numbers…
  - …from one system to another…
  - …remain applicable…
  - …for converting representations of unsigned floating-point numbers
- Before discussing the exceptional case, here's the caveat for another
  - When converting a representation from *binary to hex*…
  - …where the binary digits have to be separated into groups of 4…
  - …(and padding zeroes are added when necessary):
  - (1) The "grouping" task must be *divided into 2 (sub-tasks) at the radix point*
  - (2) Each sub-task *starts at the radix point*
  - (3) One sub-task moves *leftward* and the other *rightward*
  - (4) *Left*ward moving sub-task pads zeroes to the *left* end when necessary
  - (5) *Right*ward moving sub-task pads zeroes to the *right* end when necessary

Digital Representation of Information
(unsigned floating-point numbers)

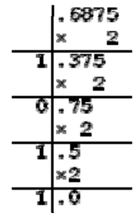
- (Caveat for binary to hex conversion) *Example 1*:
  10101011.11001101b = 1010 1011.1100 1101b = AB.CDh
- (Caveat for binary to hex conversion) *Example 2*:
  1.1b = 0001.1000b = 1.8h
- (Caveat for binary to hex conversion) *Example 3*:
  111010.10101b = 0011 1010.1010 1000b = 3A.A8h

Digital Representation of Information
(unsigned floating-point numbers)

- To illustrate the exceptional case, consider the conversion of a floating-point number from decimal to binary
  - *Example*: convert 16.6875 to binary
- The usual strategy: divide the decimal number into 2 parts
  - An *unsigned whole number* (16 for the above example)
  - An *unsigned fraction* (.6875 for the above example)
- We already know how to convert the *unsigned whole number* into binary
  - Using the repeated subtraction or repeated division method
- The difficulty lies in converting the *unsigned fraction*

Digital Representation of Information (unsigned floating-point numbers)

- One method (called the ***repeated multiplication method***) to convert an unsigned ***decimal fraction*** to its base-***b*** equivalent:
  - *Illustrative example:* the binary representation of .6875 is 0.1011b as shown on the right
  - Multiply the fractional part repeatedly by ***b*** until the fractional part of the product becomes **0** (may not happen – will soon see)
  - The *integer parts* are the digits from *left to right* of the base-*b* representation
  - (*Drills*) Convert into binary:
    - .5
    - .25
    - .625
    - .0625
    - .828125

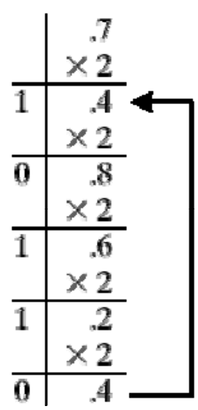


(basis for the method)

Decimal-to-Binary Conversion for .6875 Using Repeated Multiplication Method					
Multiplication by 2 = Outcome	Meaning of Outcome	Bit(s) Unraveled			
		2^{-1}	2^{-2}	2^{-3}	2^{-4}
.6875×2 = 1.375	.6875 = 1(2^{-1}) + .375(2^{-1})	1			
.375×2 = 0.75	.375(2^{-1}) = 0(2^{-2}) + .75(2^{-2})		0		
.75×2 = 1.5	.75(2^{-2}) = 1(2^{-3}) + .5(2^{-3})			1	
.5×2 = 1.0	.5(2^{-3}) = 1(2^{-4}) + .0(2^{-4})				1

Digital Representation of Information (unsigned floating-point numbers)

- Although the decimal representation of a fraction may terminate, its representation in another base may not terminate
  - For example, the binary representation of 0.7 is 0.10110011001100110...b where the block of digits 0110 is repeated indefinitely (this is commonly written as $0.1\overline{0110}$b)
  - (*Drills*) Convert into binary:
    - .3
    - .6
    - .05
    - .33333...



Digital Representation of Information
(unsigned floating-point numbers)

- To simplify hardware, floating-point numbers are usually stored in a format that uses the scientific notation
  - But in binary (using powers of two instead of 10)
  - *Example:*
    23.85 or **10111.11011001100110…b**
    would be stored as
    1.0111111011001100110… × **2**$^{\mathbf{100}}$
    where the exponent (**100**) is in binary
- A *normalized* floating-point number has the form
  1.ssssssssssssssss × 2$^{\text{eeeeeeee}}$
  where 1.ssssssssssssssss is called the *significand* (also referred to as *fraction* or *mantissa*) and **eeeeeeee** is the *exponent*

Digital Representation of Information
(signed floating-point numbers)

- Up to the late 1970s, signed floating-point numbers were represented in binary differently by different computer makers
  - This made many programs incompatible for different machines
- In 1985, IEEE (Institute of Electrical and Electronic Engineers) standardized the representation
  - Published *IEEE 754-1985 standard*
  - Almost all software and hardware companies, including IBM, Intel and Microsoft now abide by the standard
  - Often the standard is supported by the hardware of the computer itself (e.g., Intel's math coprocessors, which are built into all its CPU's since the Pentium)

Digital Representation of Information
(signed floating-point numbers)

- IEEE 754-1985 defines two *binary floating-point formats* with different precisions
  - *Single precision*
  - *Double precision*
- NOTE:
  - IEEE 754-1985 also specifies other extended precision formats
  - IEEE 754-1985 is now replaced by IEEE 754-2008
    - Binary floating-point formats of IEEE 754-1985 preserved
  - (We will not discuss extended precision formats and other formats included in the IEEE 754-2008)

Digital Representation of Information
(signed floating-point numbers)

- *IEEE 754-1985 single precision* uses 32 bits for encoding
  - Range (in decimal and for both negative and positive)
    1.2×10^{-38} to $3.4 \times 10^{+38}$
  - Precision: 7 significant decimal digits
- Assignment of the 32 bits

Bit #	Bit(s) Representing
31	*sign bit* (0 for positive, 1 for negative)
23-30	*biased exponent*
0-22	*fraction* (also called *significand* or *mantissa*)

where biased exponent = exponent + 7Fh (+ 127 decimal)

• always non-negative
• easier hardware design
• less transistor consuming

• in normalized form
• leading one not stored (why?)
• (hidden one representation)

Digital Representation of Information
(signed floating-point numbers)

- *IEEE 754-1985 double precision* uses 64 bits for encoding
  - Range (in decimal and for both negative and positive):
    2.3×10^{-308} to $1.7 \times 10^{+308}$
  - Precision: 15 significant decimal digits
- Assignment of the 64 bits

Bit #	Bit(s) Representing
63	*sign bit* (0 for positive, 1 for negative)
52-62	*biased exponent*
0-51	*fraction* (also called *significand* or *mantissa*)

see notes on previous slide

where biased exponent = exponent + 3FFh (+ 1023 decimal)

Digital Representation of Information
(signed floating-point numbers)

- *IEEE 754-1985 floating-point formats* summary
  - What we've discussed + a little more

IEEE 754 FLOATING-POINT STANDARD

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

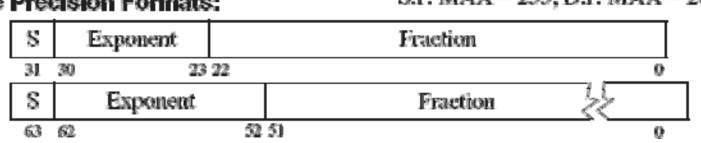
where Single Precision Bias = 127,
Double Precision Bias = 1023.

IEEE 754 Symbols

Exponent	Fraction	Object
0	0	± 0
0	≠0	± Denorm
1 to MAX - 1	anything	± Fl. Pt. Num.
MAX	0	±∞
MAX	≠0	NaN

S.P. MAX = 255, D.P. MAX = 2047

IEEE Single Precision and Double Precision Formats:



Digital Representation of Information
(signed floating-point numbers)

- Converting a given real number to *IEEE single precision*:
  - Convert real number (ignoring the sign) into *binary* number
  - Express converted binary number in *normalized, scientific form*
    $1.ssss \times 2^{eeee}$
    (ssss is the *fraction* or *significand* or *mantissa*)
  - Calculate biased exponent = eeee + 7Fh
  - Place the bits according to format:

Bit #	Bit(s) Representing
31	*sign bit* (0 for positive, 1 for negative)
23-30	*biased exponent*
0-22	*fraction* (or *significand* or *mantissa*)

- Steps are similar for converting to *IEEE double precision*

Digital Representation of Information
(signed floating-point numbers)

- E.g.: converting 9.75 to *IEEE single precision*
  - Convert real number (ignoring the sign) into *binary* number
    9.75 = 1001.11b
  - Express converted binary number in *normalized, scientific form*
    $1001.11b = 1.00111 \times 2^3$
  - Calculate biased exponent = 3 + 7Fh = 82h
  - Place the bits according to format:

Bit #	Bit(s) Representing
31	0 (= positive)
23-30	10000010 (= 82h)
0-22	00111000000000000000000

Digital Representation of Information
(signed floating-point numbers)

- E.g.: converting 0.078125 to *IEEE single precision*
  - Convert real number (ignoring the sign) into *binary* number
    0.078125 = 0.000101b
  - Express converted binary number in *normalized, scientific form*
    0.000101b = 1.**01** $\times$ 2^{-4}
  - Calculate biased exponent = -4 + 7Fh = **7B**h
  - Place the bits according to format:

Bit #	Bit(s) Representing
31	0 (= positive)
23-30	01111011 (= 7Bh)
0-22	01000000000000000000000

Digital Representation of Information
(signed floating-point numbers)

- E.g.: converting –96.27 to *IEEE single precision*
  - Convert real number (ignoring the sign) into *binary* number
    96.27 = 1100000.01000101000111101b
  - Express converted binary number in *normalized, scientific form*
    1100000.01000101000111101b =
    1.**10000001000101000111101**b $\times$ 2^6
  - Calculate biased exponent = 6 + 7Fh = **85**h
  - Place the bits according to format:

Bit #	Bit(s) Representing
31	1 (= negative)
23-30	10000101 (= 85h)
0-22	10000001000101000111101

Digital Representation of Information
(signed floating-point numbers)

- E.g.: converting 152.1875 to *IEEE double precision*
  - Convert real number (ignoring the sign) into *binary* number
    152.1875 = 10011000.0011b
  - Express converted binary number in *normalized, scientific form*
    10011000.0011b = 1.00110000011b $\times$ 2^7
  - Calculate biased exponent = 7 + 3FFh = 406h
  - Place the bits according to format:

Bit #	Bit(s) Representing
63	0 (= positive)
52-62	10000000110 (= 406h)
0-51	00110000011000...000

Digital Representation of Information
(signed floating-point numbers)

- E.g.: get decimal value of *IEEE single precision* number below:
  - 11000000111100000000000000000000

S	Exponent	Fraction
1	1000 0001	111 0000 0000 0000 0000 0000

$(-1)^{S} \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-127)}$

$(-1)^{1} \times (1 + .111)\text{b} \times 2^{(129-127)}$

$-1 \times 1.111\text{b} \times 2^{(2)}$

-111.1b

-7.5

Digital Representation of Information
(ASCII Character Set)

- Slide 21 of **002DigitalInfoRep…n01** indicates that any type of data can at least be represented the following way:
  - Describe the data in some language (that uses characters like letters, digits, punctuations marks, etc.)
  - Encode the characters that are used by the language as sequence of binary bits
  - Now the data can be represented in binary
- Until a few years ago, character sets use only 8 bits
  - The 16-bit *Unicode* character set was created to cater for the great diversity of languages around the world
- When running in character mode, personal computers invariably use the *ASCII character set*
  - ASCII = **A**merican **S**tandard **C**ode for **I**nformation **E**xchange

Digital Representation of Information
(ASCII Character Set)

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

- **CR = “carriage return” (Windows: move to beginning of line)**
- **LF = “line feed” (Windows: move directly one line below)**
- **SPC = “blank space”**

Digital Representation of Information
(ASCII Character Set)

- It is normally unnecessary to know the actual values of ASCII characters
  - The form `'x'` should be used wherever possible
- But certain relationships among the characters are helpful and should be learned
  - Some of these are listed in the next two slides

Digital Representation of Information
(ASCII Character Set)

- Characters represented by numbers **0** through **31** are *non-printing characters*
  - Used for standard control purposes
  - Common examples are
    - `13` = carriage return
    - `10` = line feed
    - `9` = tab
- Character **127** (DEL) is also *non-printing characters*
- Characters corresponding to **0** through **31** and **127** can be represented in assembly language *only* by their numeric values
- Characters corresponding to **1** through **26** in addition to having various standard interpretations are also called *control characters*

Digital Representation of Information
(ASCII Character Set)

- The ***space*** character (`' '`) corresponds to **32**
  - It is considered a printing character
- The ***digit*** characters run *continuously*
  - Digit **`'0'`** = **48**, digit **`'1'`** = **49**, and so on
  - Useful formula to know:
    `digit_character = '0' + digit_value`
- The letters of the alphabet are in *two continuous ranges*
  - One for uppercase and one for lowercase
  - Useful formula to know (for any single particular letter):
    `lowercase = corresponding_uppercase + ('a' - 'A')`
  - Binary representations for a lowercase letter and its corresponding uppercase letter differ by only 1 bit (*which one?*)