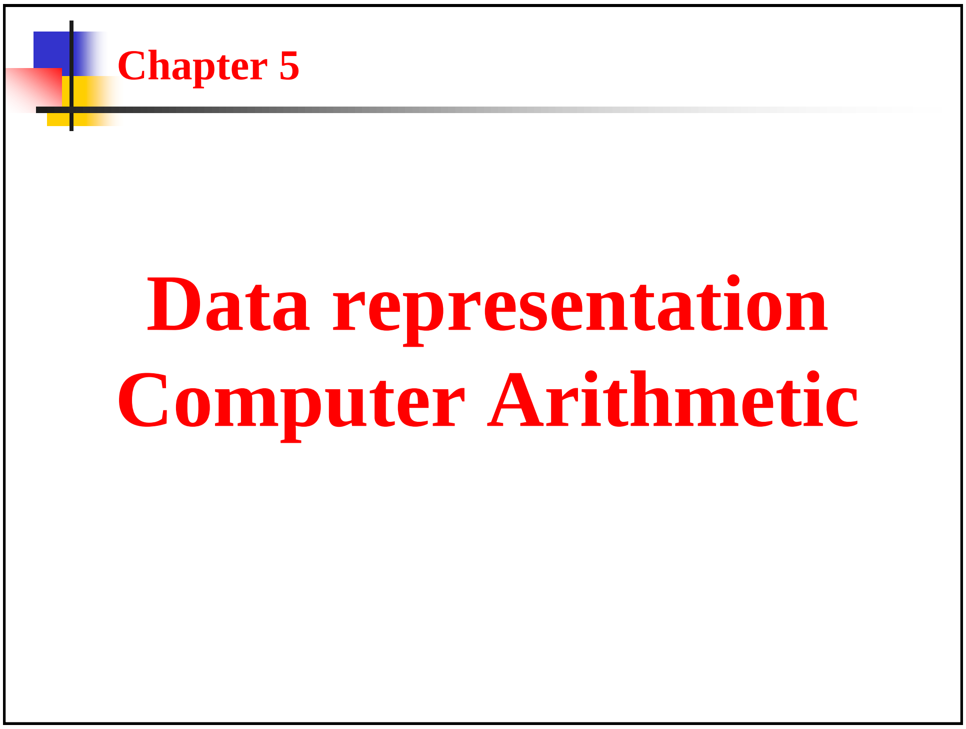
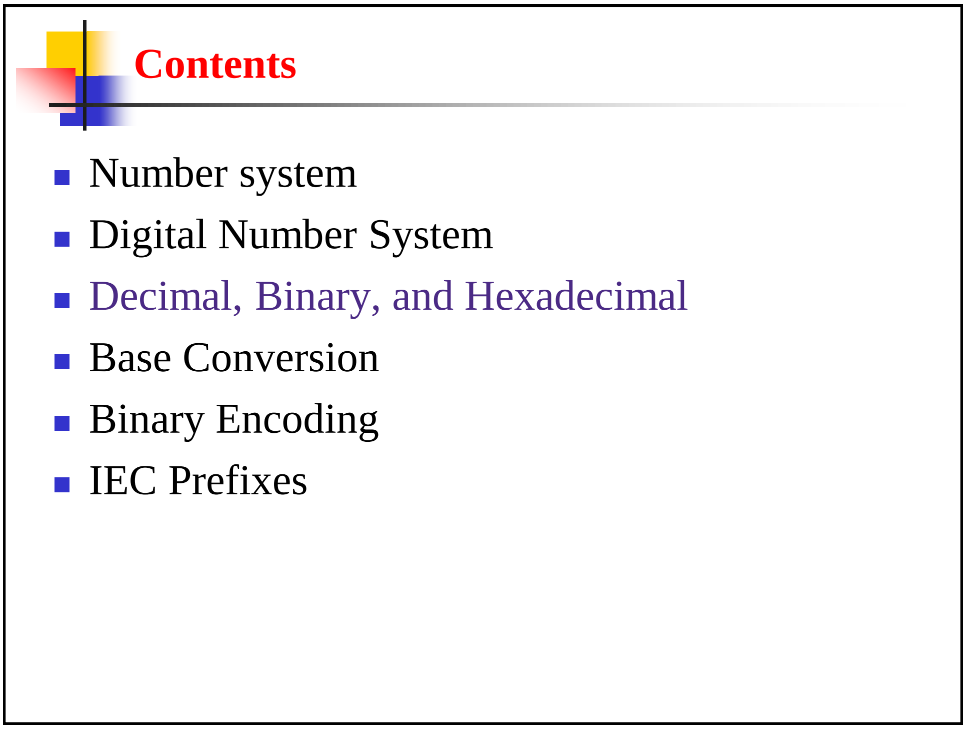


Chapter 5

Data representation Computer Arithmetic



Contents

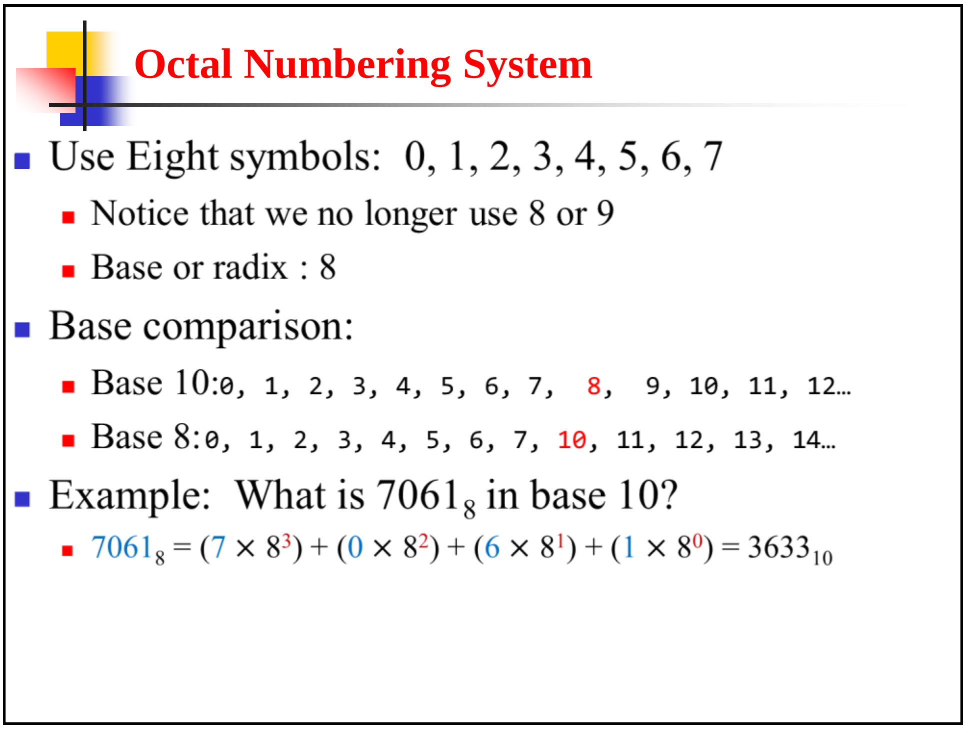
- Number system
- Digital Number System
- Decimal, Binary, and Hexadecimal
- Base Conversion
- Binary Encoding
- IEC Prefixes

Base (radix) of a number system

- Is number of digits to present all values
- There are some number systems common
  - Binary systems
  - Decimal systems
  - Octal systems
  - Hexa systems
- No reason that we can't also use base 7 or 19 but they're obviously not very useful

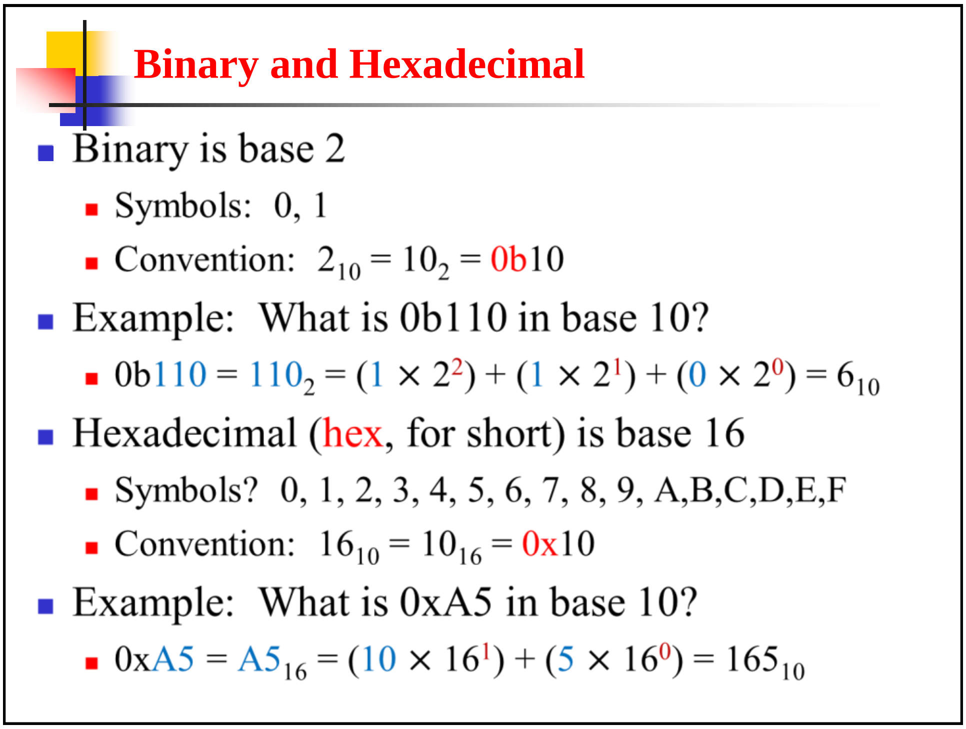
Decimal Numbering System

- Use Ten symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- Base or radix : 10
- Represent larger numbers as a sequence of digits
  - Each digit is one of the available symbols
- Example: 7061 in decimal (base 10)
  - $7061_{10} = (7 \times 10^3) + (0 \times 10^2) + (6 \times 10^1) + (1 \times 10^0)$
  - 7: MSD - Most significant digit
  - 1: LSD - Least significant digit



Octal Numbering System

- Use Eight symbols: 0, 1, 2, 3, 4, 5, 6, 7
  - Notice that we no longer use 8 or 9
  - Base or radix : 8
- Base comparison:
  - Base 10:`0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12…`
  - Base 8:`0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14…`
- Example: What is 7061_8 in base 10?
  - $7061_8 = (7 \times 8^3) + (0 \times 8^2) + (6 \times 8^1) + (1 \times 8^0) = 3633_{10}$



Binary and Hexadecimal

- Binary is base 2
  - Symbols: 0, 1
  - Convention: $2_{10} = 10_2 =$ 0b10
- Example: What is 0b110 in base 10?
  - 0b110 $= 110_2 = (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) = 6_{10}$
- Hexadecimal (hex, for short) is base 16
  - Symbols? 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A,B,C,D,E,F
  - Convention: $16_{10} = 10_{16} =$ 0x10
- Example: What is 0xA5 in base 10?
  - 0xA5 $= A5_{16} = (10 \times 16^1) + (5 \times 16^0) = 165_{10}$

Base Conversion

- Is converting from one base to another
- Any non-negative number can be written in any base
- Since most humans are used to the decimal system and most computers use the binary system it is important for people who work with computers to understand how to convert between binary and decimal

Conver from any base system to decimal

- Use formular :

$$\mathbf{N_S = C_n\, S^n + C_{n-1}S^{n-1} + C_{n-2}\, S^{n-2} + \ldots + C_0\, S^0 + C_{-1}\, S^{-1} + \ldots}$$

- Or

$$\mathbf{N_S = \sum C_i\, S^i}$$

- In which :

$$0 \le C_i \le S\text{-}1$$

i is position of ith digit, i=0 is the first digit in front of dot decimal

Examples : convert following numbers into D

- From B – D
  - Example : **1001_2**

    $1001_2 = 1x2^3 + 0x2^2 + 0x2^1 + 1x2^0 = 9$
- From O – D
  - Example : **162.43_8**

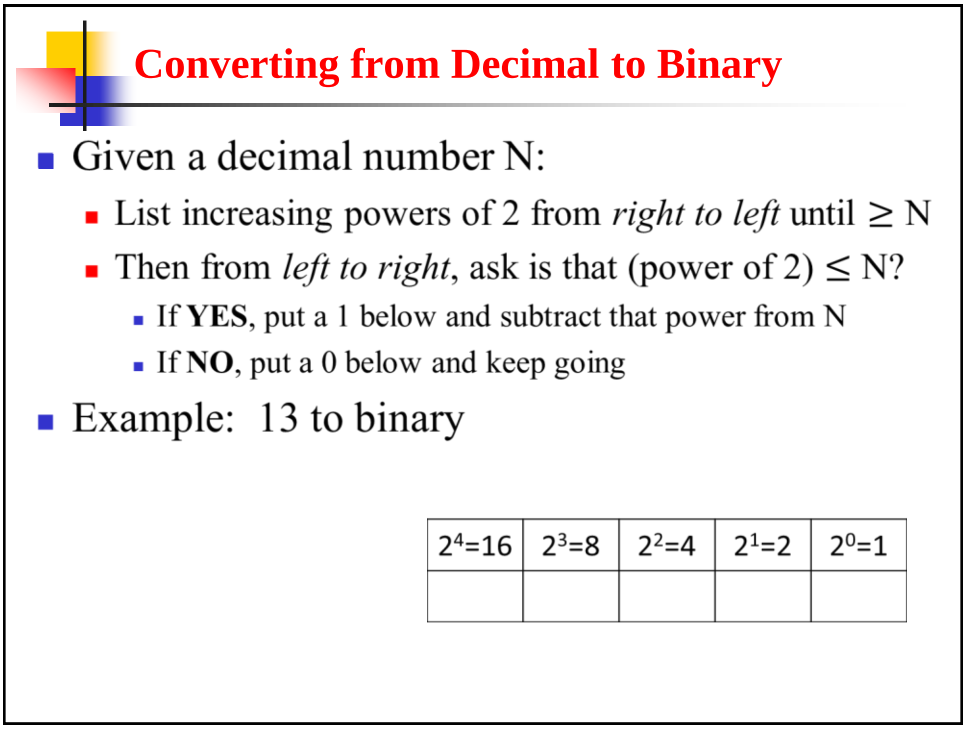
    $162.43_8 = 1x8^2+6x8^1+2x8^0+4x8^{-1}+3x8^{-2}$
- From H – D
  - Example : **$1E4A.6B_{16}$**

  $1x16^3+Ex16^2+4x16^1+Ax16^0+6x16^{-1}+Bx16^{-2}$

  $1x16^3+14x16^2+4x16^1+10x16^0+6x16^{-1}+11x16^{-2}$

Examples :

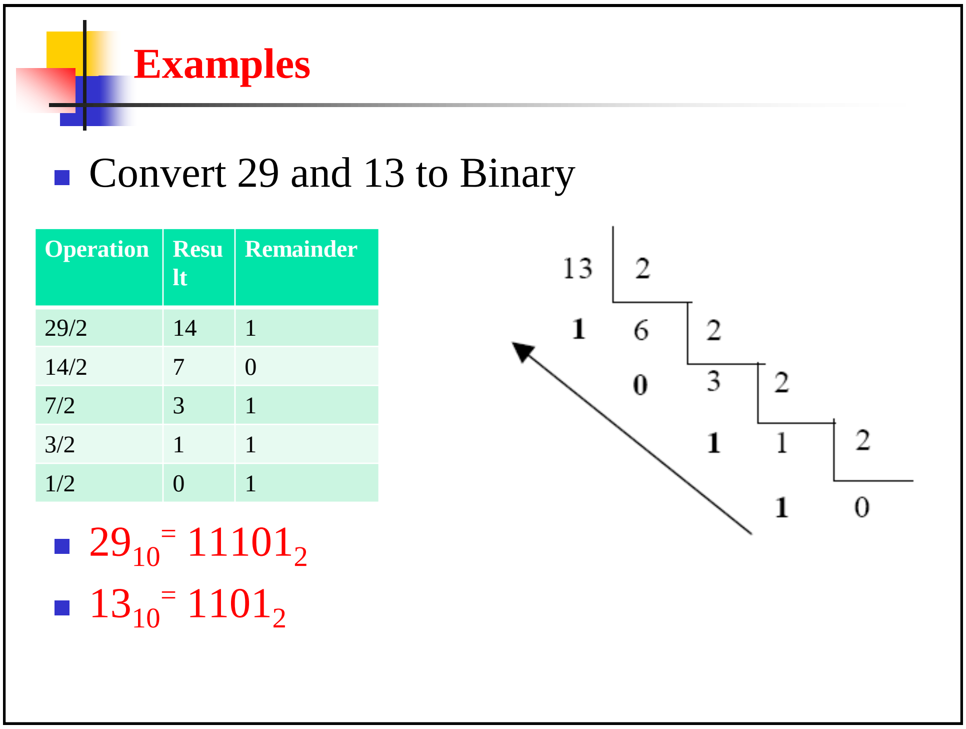
- Can convert from any base *to* base 10
  - $0b110 = 110_2 = (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) = 6_{10}$
  - $0xA5 = A5_{16} = (10 \times 16^1) + (5 \times 16^0) = 165_{10}$
- Comments : MSB and LSB
  - Convert to decimal
  - $1101.11 = 1x2^3 + 1x2^2 + 0x2^1 + 1x2^0 + 2^{-1} + 2^{-2} = 13.75$
  - 1: MSB - Most significant bit (Left most bit)
  - 1: LSB - Least significant digit (Right most bit)



Converting from Decimal to Binary

- Given a decimal number N:
  - List increasing powers of 2 from *right to left* until ≥ N
  - Then from *left to right*, ask is that (power of 2) ≤ N?
    - If **YES**, put a 1 below and subtract that power from N
    - If **NO**, put a 0 below and keep going
- Example: 13 to binary

2^4=16	2^3=8	2^2=4	2^1=2	2^0=1



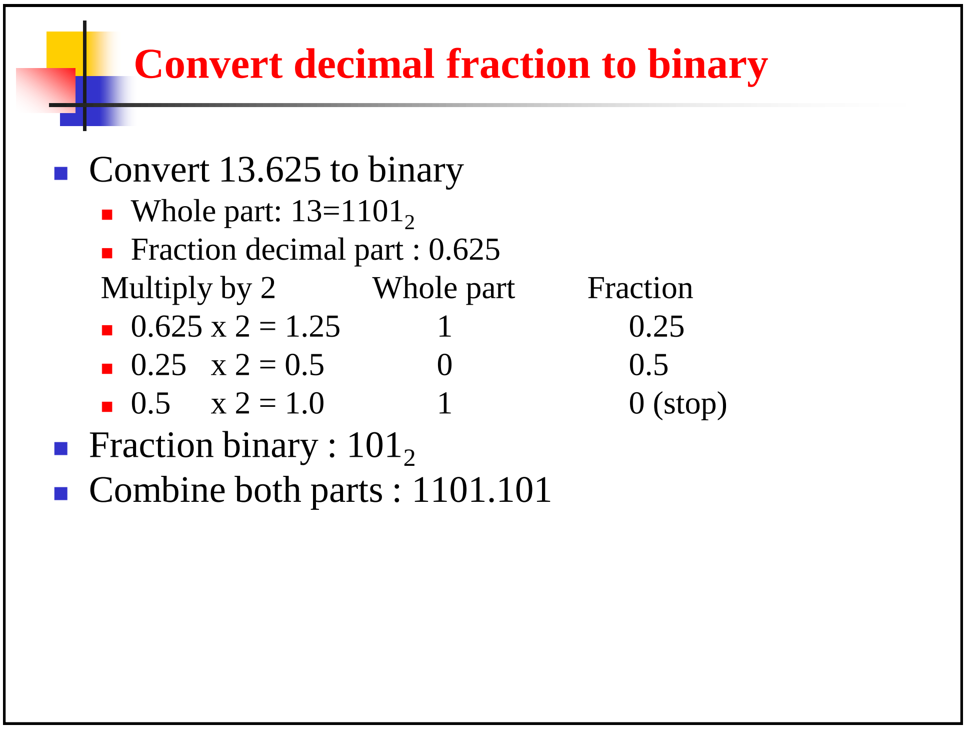
Examples

- Convert 29 and 13 to Binary

Operation	Resu lt	Remainder
29/2	14	1
14/2	7	0
7/2	3	1
3/2	1	1
1/2	0	1

13
2
1
6
2
0
3
2
1
1
2
1
0

- $29_{10} = 11101_2$
- $13_{10} = 1101_2$

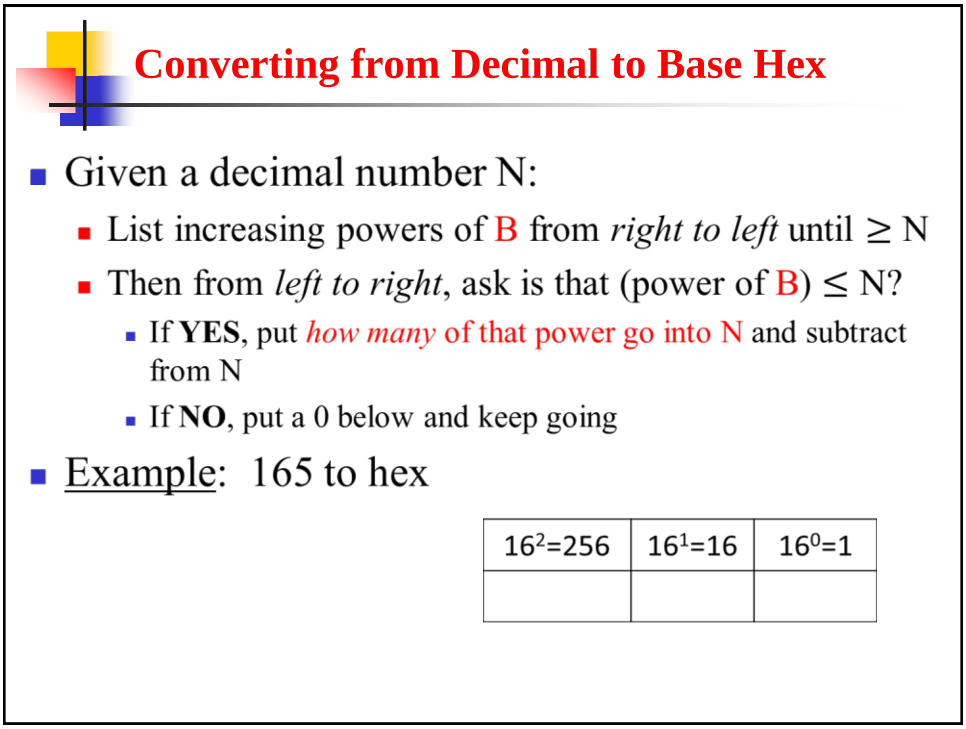


Convert decimal fraction to binary

- Convert 13.625 to binary
  - Whole part: $13=1101_2$
  - Fraction decimal part : 0.625

Multiply by 2	Whole part	Fraction
0.625 x 2 = 1.25	1	0.25
0.25 x 2 = 0.5	0	0.5
0.5 x 2 = 1.0	1	0 (stop)

- Fraction binary : 101_2
- Combine both parts : 1101.101

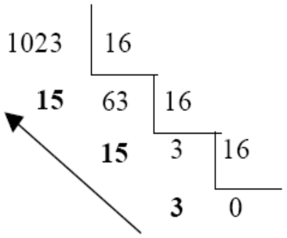


Converting from Decimal to Base Hex

- Given a decimal number N:
  - List increasing powers of B from *right to left* until ≥ N
  - Then from *left to right*, ask is that (power of B) ≤ N?
    - If **YES**, put *how many* of that power go into N and subtract from N
    - If **NO**, put a 0 below and keep going
- Example: 165 to hex

$16^2=256$	$16^1=16$	$16^0=1$

- Examples : convert from Decimal to Hex
- 1023 = 3FFH



1023 | 16

15 63 | 16

15 3 | 16

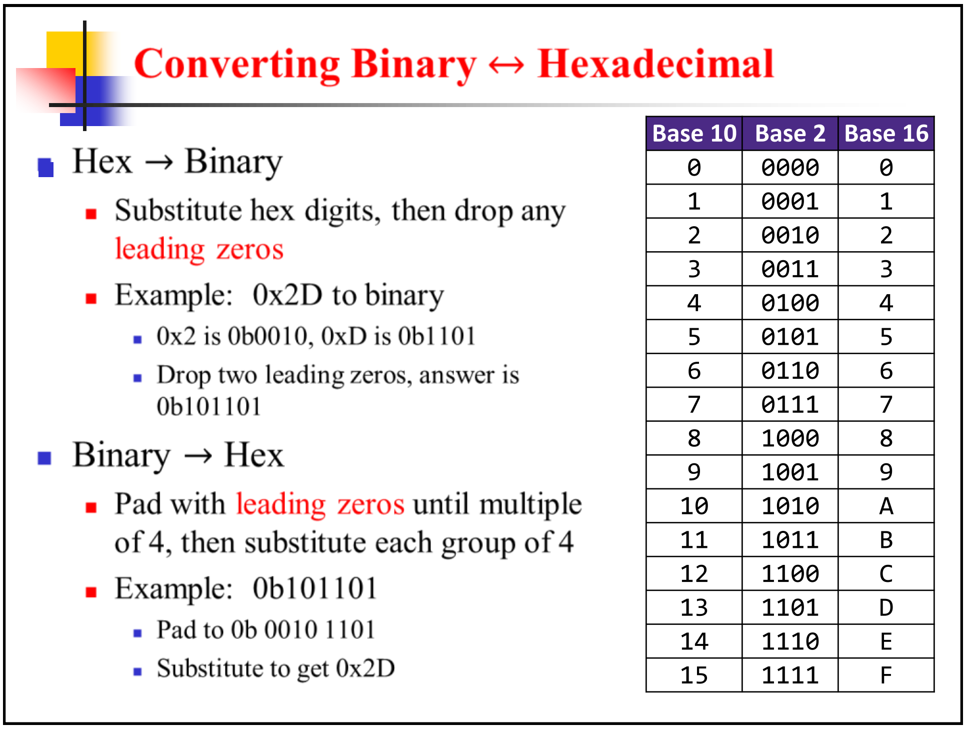
3 0

Convert from Decimal to Octal

- Example : 153.513 to octal
  - Whole part: 153=231_8
  - Fraction decimal part : 0.513

Multiply by 8	Whole part	Fraction
0.513x 8 = 4.104	4	0.104
0.104x 8 = 0.832	0	0.832
0.832x 8 = 6.656	6	0.656
0.656x8 = 5.248	5	0.28
0.248x8 = 1.984	1	0.984…

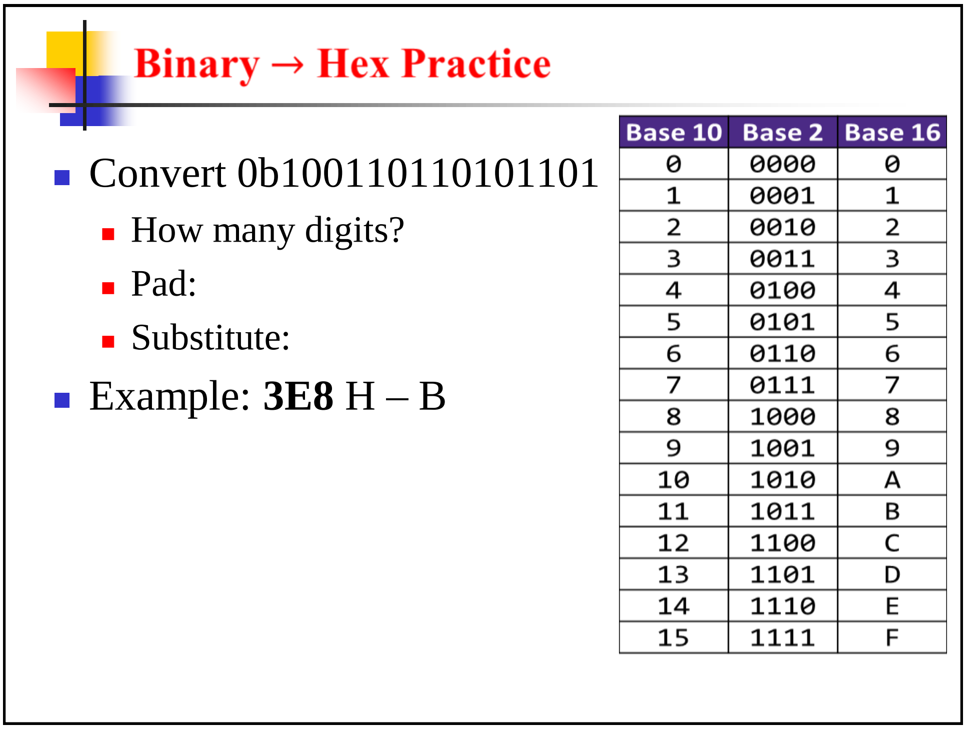
- Result : 40651_8
- Final : 231.40651_8



Converting Binary ↔ Hexadecimal

- Hex → Binary
  - Substitute hex digits, then drop any leading zeros
  - Example: 0x2D to binary
    - 0x2 is 0b0010, 0xD is 0b1101
    - Drop two leading zeros, answer is 0b101101
- Binary → Hex
  - Pad with leading zeros until multiple of 4, then substitute each group of 4
  - Example: 0b101101
    - Pad to 0b 0010 1101
    - Substitute to get 0x2D

Base 10	Base 2	Base 16
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
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F



Binary → Hex Practice

- Convert 0b100110110101101
  - How many digits?
  - Pad:
  - Substitute:
- Example: **3E8** H – B

Base 10	Base 2	Base 16
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
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Converting Binary to Octal

- Using groups of hextets, the binary number 11010100011011_2 (= 13595_{10}) in hexadecimal is:

0011	0101	0001	1011
3	5	1	B

- Octal (base 8) values are derived from binary by using groups of three bits ($8 = 2^3$):

011	010	100	011	011
3	2	4	3	3

- **Octal was very useful when computers used six-bit words.**

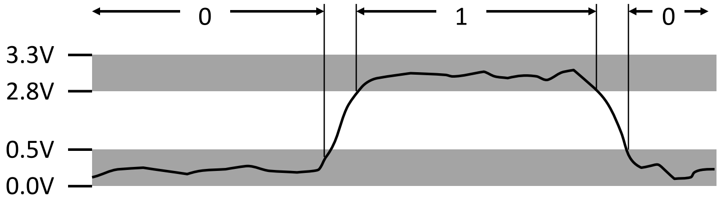
Base Comparison

- Why does all of this matter?
  - *Humans* think about numbers in **base 10**, but *computers* “think” about numbers in **base 2**
  - Binary encoding is what allows computers to do all of the amazing things that they do!
- You should have this table memorized by the end of the class
  - Might as well start now!

Base 10	Base 2	Base 16
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
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Aside: Why Base 2?

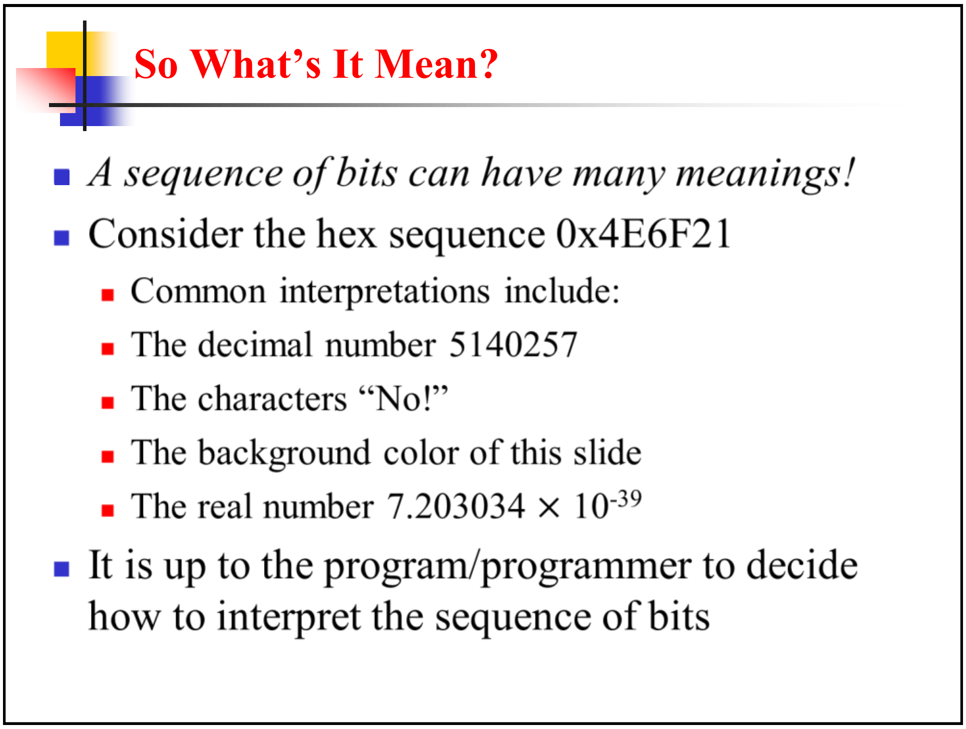
- Electronic implementation
  - Easy to store with bi-stable elements
  - Reliably transmitted on noisy and inaccurate wires



- Other bases possible, but not yet viable:
  - DNA data storage (base 4: A, C, G, T) is a hot topic
  - Quantum computing

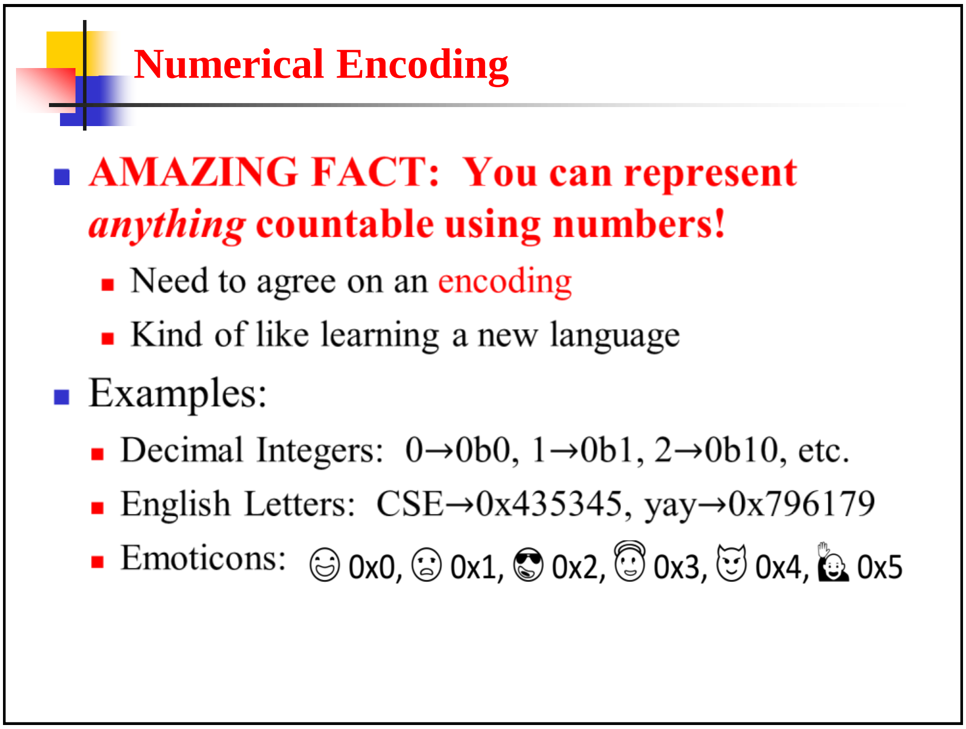
Binary Encoding

- With N binary digits, how many “things” can you represent?
  - Need N binary digits to represent n things, where $2^N \geq n$
  - Example: 5 binary digits for alphabet because $2^5 = 32 > 26$
- A binary digit is known as a bit
- A group of 4 bits (1 hex digit) is called a nibble
- A group of 8 bits (2 hex digits) is called a byte
  - 1 bit → 2 things, 1 nibble → 16 things, 1 byte → 256 things



So What's It Mean?

- *A sequence of bits can have many meanings!*
- Consider the hex sequence 0x4E6F21
  - Common interpretations include:
  - The decimal number 5140257
  - The characters "No!"
  - The background color of this slide
  - The real number 7.203034×10^{-39}
- It is up to the program/programmer to decide how to interpret the sequence of bits



Numerical Encoding

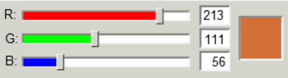
- **AMAZING FACT: You can represent *anything* countable using numbers!**
  - Need to agree on an encoding
  - Kind of like learning a new language
- Examples:
  - Decimal Integers: 0→0b0, 1→0b1, 2→0b10, etc.
  - English Letters: CSE→0x435345, yay→0x796179
  - Emoticons: 😊 0x0, ☹ 0x1, 😎 0x2, 😇 0x3, 😈 0x4, 🙋 0x5

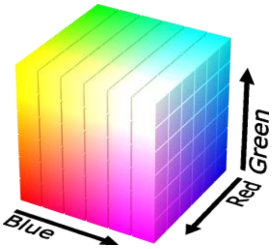
Binary Encoding

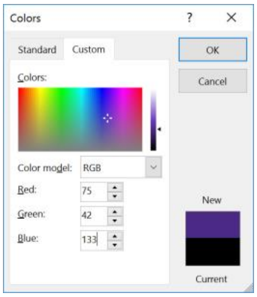
- With N binary digits, how many “things” can you represent?
  - Need N binary digits to represent n things, where $2^N \geq n$
  - Example: 5 binary digits for alphabet because $2^5 = 32 > 26$
- A binary digit is known as a bit
- A group of 4 bits (1 hex digit) is called a nibble
- A group of 8 bits (2 hex digits) is called a byte
  - 1 bit → 2 things, 1 nibble → 16 things, 1 byte → 256 things

Binary Encoding – Colors

- RGB – Red, Green, Blue
  - Additive color model (light): byte (8 bits) for each color
  - Commonly seen in hex (in HTML, photo editing, etc.)
  - Examples: **Blue**→0x0000FF, **Gold**→0xFFD700, **White**→0xFFFFFF, **Deep Pink**→0xFF1493


R: 213
G: 111
B: 56

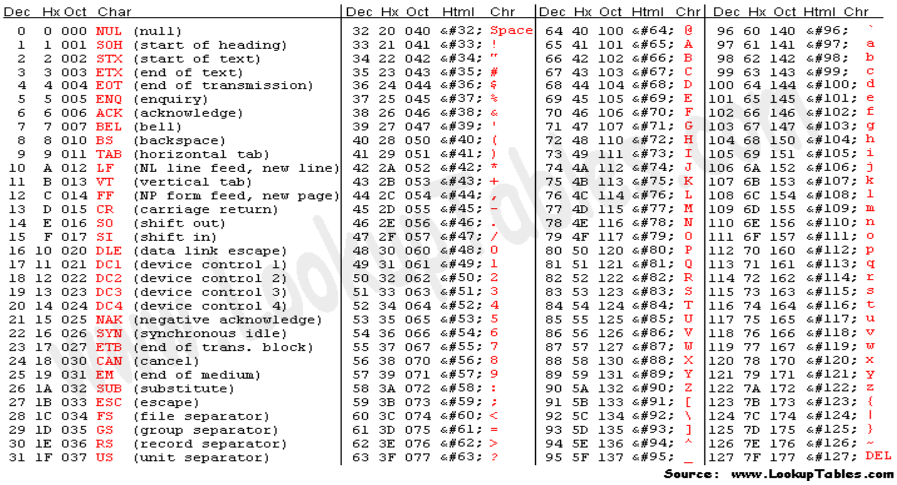




Color model: RGB
Red: 75
Green: 42
Blue: 133

Binary Encoding – Characters/Text

- ASCII Encoding (www.asciitable.com)
  - American Standard Code for Information Interchange

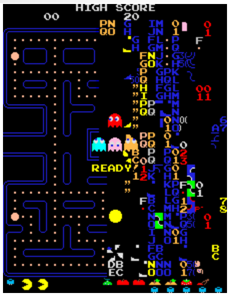


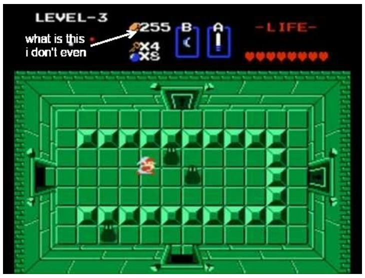
Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT (end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1 (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2 (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3 (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4 (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

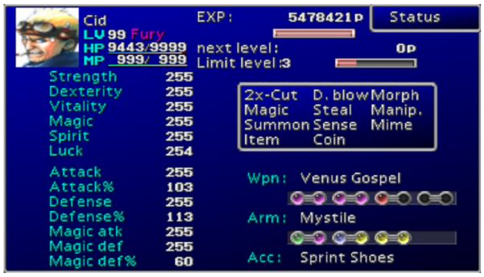
Source: www.LookupTables.com

Binary Encoding – Video Games

- As programs run, in-game data is stored somewhere
- In many old games, stats would go to a maximum of 255
- Pacman "kill screen"
  - http://www.numberphile.com/videos/255.html







Cid
LV 99 Fury
HP 9443/9999
MP 999/ 999

EXP: 5478421p
next level: 0p
Limit level:3

Status

Strength	255
Dexterity	255
Vitality	255
Magic	255
Spirit	255
Luck	254
Attack	255
Attack%	103
Defense	255
Defense%	113
Magic atk	255
Magic def	255
Magic def%	60

2x-Cut D. blow Morph
Magic Steal Manip.
Summon Sense Mime
Item Coin

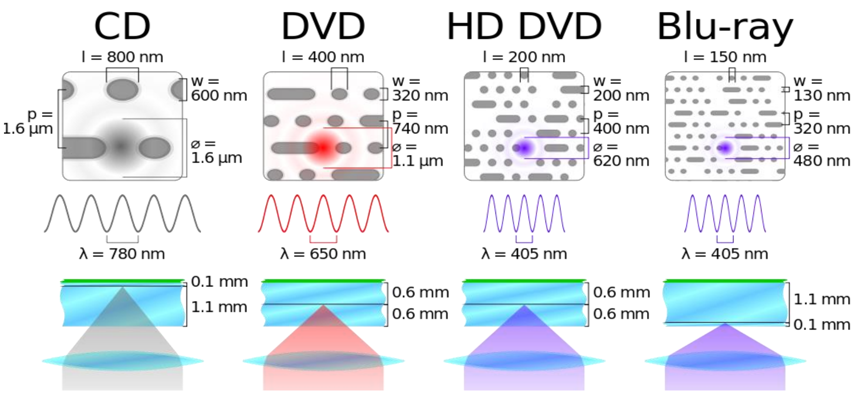
Wpn: Venus Gospel
Arm: Mystile
Acc: Sprint Shoes

Binary Encoding – Files and Programs

- At the lowest level, all digital data is stored as bits!
- Layers of abstraction keep everything comprehensible
  - Data/files are groups of bits interpreted by program
  - Program is actually groups of bits being interpreted by your CPU
- Computer Memory Demo
  - From vim: `%!xxd`
  - From emacs: `M-x hexl-mode`

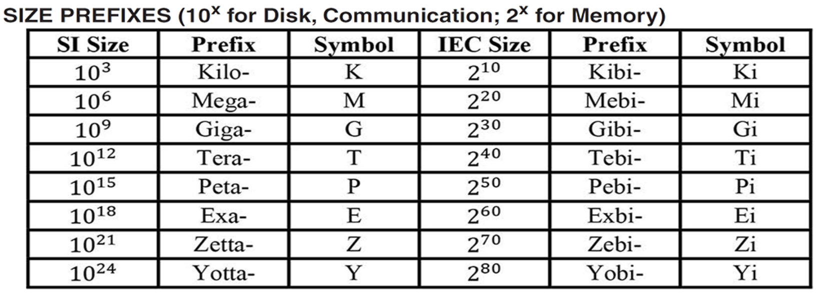
Binary Encoding – Optical Disk Storage

- Data stored using tiny indentations in a spiral pattern
  - Not a direct translation between 0/1 and bump/no bump
  - https://en.wikipedia.org/wiki/Compact_disc#Physical_details



Units and Prefixes

- Here focusing on large numbers (exponents > 0)
- Note that $10^3 \approx 2^{10}$
- SI prefixes are *ambiguous* if base 10 or 2
- IEC prefixes are *unambiguously* base 2

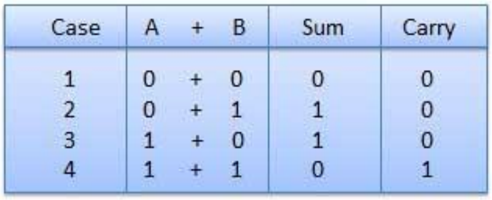


SIZE PREFIXES (10^x for Disk, Communication; 2^x for Memory)

SI Size	Prefix	Symbol	IEC Size	Prefix	Symbol
10^3	Kilo-	K	2^{10}	Kibi-	Ki
10^6	Mega-	M	2^{20}	Mebi-	Mi
10^9	Giga-	G	2^{30}	Gibi-	Gi
10^{12}	Tera-	T	2^{40}	Tebi-	Ti
10^{15}	Peta-	P	2^{50}	Pebi-	Pi
10^{18}	Exa-	E	2^{60}	Exbi-	Ei
10^{21}	Zetta-	Z	2^{70}	Zebi-	Zi
10^{24}	Yotta-	Y	2^{80}	Yobi-	Yi

Arithemetic on Binary system

- Addition : four cases
  - 0 + 0 = 0
  - 0 + 1 = 1
  - 0 + 1 = 1
  - 1 + 1 = 10



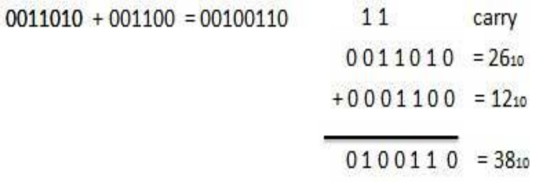
Case	A	+	B	Sum	Carry
1	0	+	0	0	0
2	0	+	1	1	0
3	1	+	0	1	0
4	1	+	1	0	1

- Examples :

1 1. 0 1 1 (3.375)

<u>1 0. 1 1 0 (2.750)</u>

1 1 0. 0 0 1 (6.125)



```
0011010 + 001100 = 00100110       1 1          carry
                                  0 0 1 1 0 1 0   = 26₁₀
                                 +0 0 0 1 1 0 0   = 12₁₀
                                 ______________
                                  0 1 0 0 1 1 0   = 38₁₀
```

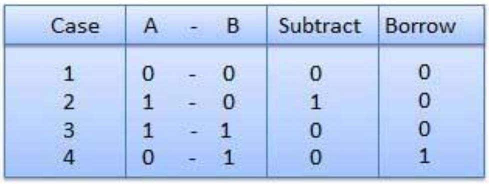
- Substraction : has four cases

0 – 0 = 0

1 – 0 = 1

1 – 1 = 0

0 – 1 = 1 Borrow 1



Case	A - B	Subtract	Borrow
1	0 - 0	0	0
2	1 - 0	1	0
3	1 - 1	0	0
4	0 - 1	0	1

- Examples:

Borrow	**11**	**111**
Subtrahend	**100**	**111001**
Minus	**011**	**1011**
Result	**001**	**101110**

0011010 - 001100 = 00001110

```
   1 1       borrow
 0 0 ~~1 1~~ 0 1 0   = 26₁₀
-0 0 0 1 1 0 0   = 12₁₀
______________
 0 0 0 1 1 1 0   = 14₁₀
```

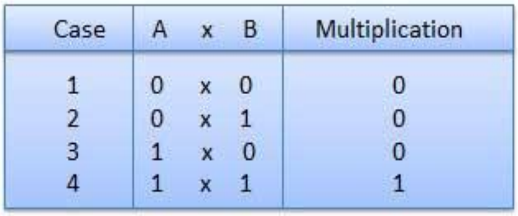
- Multiply :

0 x 0 = 0

0 x 1 = 0

1 x 0 = 0

1 x 1 = 1



Case	A x B	Multiplication
1	0 x 0	0
2	0 x 1	0
3	1 x 0	0
4	1 x 1	1

- Examples:

```
  7      →    0111
x 5      →  x 0101
 ---         ------
 35           0111
             0000
            0111
           0000
         ----------
           0100011
```

$= 1.2^5 + 1.2^1 + 1.2^0 = 35_{(10)}$

Signed Integer Representation

- The conversions we have so far presented have involved only positive numbers.
- To represent negative values, computer systems allocate the high-order bit to indicate the sign of a value.
  - The high-order bit is the leftmost bit in a byte. It is also called the most significant bit.
- The remaining bits contain the value of the number.

- There are three ways in which signed binary numbers may be expressed:
  - Signed magnitude,
  - One’s complement and
  - Two’s complement.
- In an 8-bit word, signed magnitude representation places the absolute value of the number in the 7 bits to the right of the sign bit.

Signed magnitude

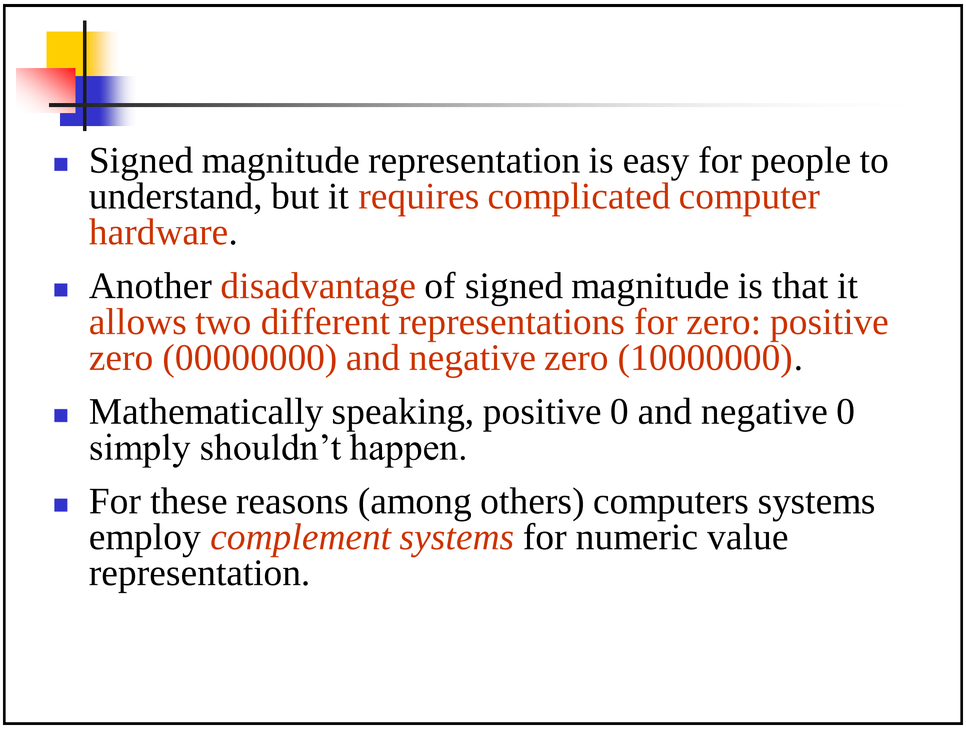
- For example, in 8-bit signed magnitude,
  - Positive 3 is: 00000011
  - Negative 3 is: 10000011
- Computers perform arithmetic operations on signed magnitude numbers in much the same way as humans carry out pencil and paper arithmetic.
  - Humans often ignore the signs of the operands while performing a calculation, applying the appropriate sign after the calculation is complete.

Comments

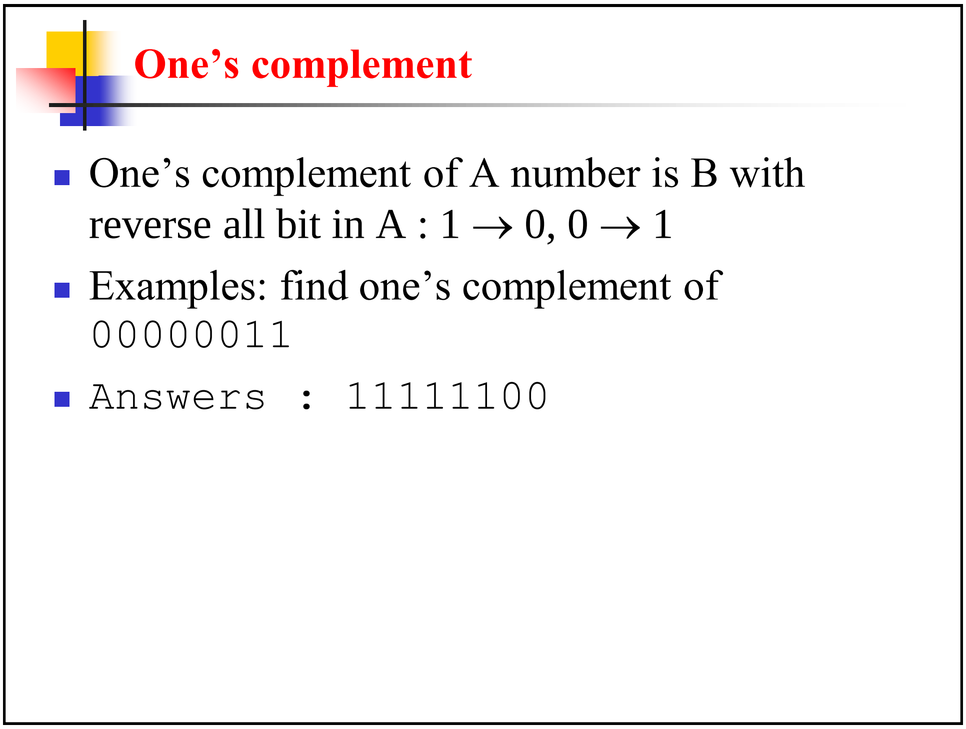
- Zero number has 2 ways present
  - 000000 (0)
  - 100000 (-0)
- Calculations are almost wrong result
- Examples : 75 + 46
- Examples : 107+46 =25

```
            1 1 1
0       1 0 0 1 0 1 1
0   +   0 1 0 1 1 1 0
        -------------
0       1 1 1 1 0 0 1
```

```
1
          1   1 1 1
0         1 1 0 1 0 1 1
0   +     0 1 0 1 1 1 0
          -------------
0         0 0 1 1 0 0 1
```



- Signed magnitude representation is easy for people to understand, but it requires complicated computer hardware.
- Another disadvantage of signed magnitude is that it allows two different representations for zero: positive zero (00000000) and negative zero (10000000).
- Mathematically speaking, positive 0 and negative 0 simply shouldn’t happen.
- For these reasons (among others) computers systems employ *complement systems* for numeric value representation.



One’s complement

- One’s complement of A number is B with reverse all bit in A : 1 → 0, 0 → 1
- Examples: find one’s complement of `00000011`
- `Answers : 11111100`

Two’s complement

- Two’s complement equal one’s complement add 1
- Example: find two’s complement of 00000011
- One’s complement : 11111100
- Add 1 + 1

```
                 11111100
```

Signed Representation using one’s complement

- As following Rules :
  - MSB is sign bit : 0 – Positive, 1 – Negative
  - Other bits present value of postitive number or one’s complement of negative number
  - With n bit, values can be present –(2^{n-1} – 1) to (2^{n-1} – 1)
- Example : using 6 bits

17 : 010001	26 : 011010
-17 : 101110	-26 : 100101

Signed Representation using two's complement

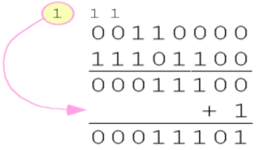
- As following rules :
  - MSB is sign bit : 0 – Positive, 1 – Negative
  - Other bits present value of postitive number or two's complement of negative number
  - With n bit, values can be present $-(2^{n-1})$ to $(2^{n-1} - 1)$
- Examples:

17 : 010001	26 : 011010
-17 : 101111	-26 : 100110

Add 2 sign number using one's complement

- With one's complement addition, the carry bit is "carried around" and added to the sum.
- Example**:** Using one's complement binary arithmetic, find the sum of 48 and – 19

13	**001101**	**-13**	**110010**
+11	**+001011**	**-11**	**+ 110100**
+24	**011000**	**-24**	**100110**
			+ 1
			100111



Add 2 sign number using two’s complement

- With two’s complement arithmetic, all we do is add our two binary numbers. Just discard any carries emitting from the high order bit.
- Example**:** Using one’s complement binary arithmetic, find the sum of 48 and - 19.

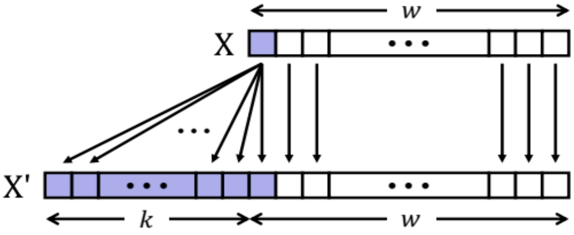
12	001100	-12	110100
+ 9	+ 001001	+ -9	+110111
21	010101	-21	1101011
		discard, result is :	101011

Sign Extension

Task: Given a w-bit signed integer X, convert it to w+k-bit signed integer X′ *with the same value*

Rule: Add k copies of sign bit

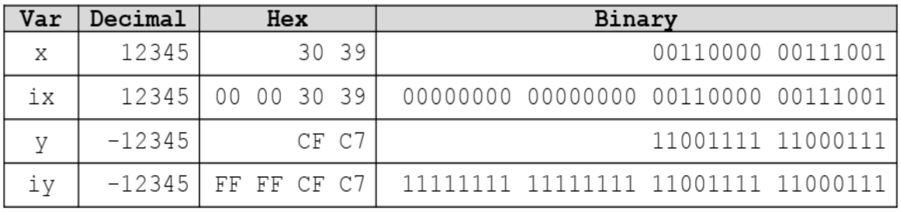
- Let x_i be the i-th digit of X in binary
- $X' = \underbrace{x_{w-1}, \dots, x_{w-1}}_{k \text{ copies of MSB}}, \underbrace{x_{w-1}, x_{w-2}, \dots, x_1, x_0}_{\text{original X}}$



Examples

- Convert from smaller to larger integral data types
- C automatically performs sign extension

```
short int x =   12345;
int      ix = (int) x;
short int y =  -12345;
int      iy = (int) y;
```



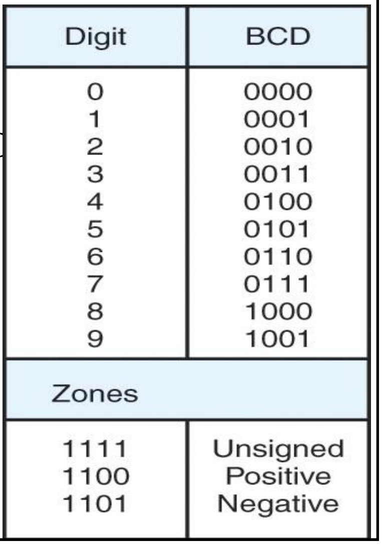
Var	Decimal	Hex	Binary
x	12345	30 39	00110000 00111001
ix	12345	00 00 30 39	00000000 00000000 00110000 00111001
y	-12345	CF C7	11001111 11000111
iy	-12345	FF FF CF C7	11111111 11111111 11001111 11000111

BCD code (Binary coded Decimal)

- Is code of ten number from 0 – 9 with 4 bit
- Examples :

1941D = 11110010101

1941D = 0001 1001 0100 0001BCD



Digit	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
Zones	
1111	Unsigned
1100	Positive
1101	Negative

Arthmetic on BCD number

- A+B : using the following rules
  - Carry at low decade to move high decace and edit low decade
  - If which decade of total greater 9 also is edited
  - Editting is performed by adding with 6
- Example :

$$\begin{array}{rrrl} 18 & 0001 & 1000 & \\ \underline{+\ 26} & +\ \underline{0010} & \underline{0110} & \\ 44 & 0011 & 1110 & \\ & & \underline{+\ 0110} & \text{(edit decade S0)} \\ & 0100 & 0100 & \end{array}$$

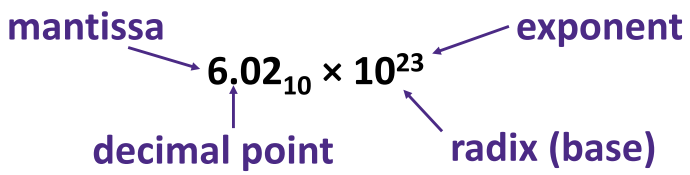
- Example

$$\begin{array}{rrrl} & 1 & & \text{(carry from decade S0)} \\ 28 & 0010 & 1000 & \\ \underline{+\ 19} & \underline{0001} & \underline{1001} & \\ 47 & 0100 & 0001 & \\ & \underline{+\ \ } & \underline{0110} & \text{(editting S0)} \\ & 0100 & 0111 & \end{array}$$

Floating point numbers

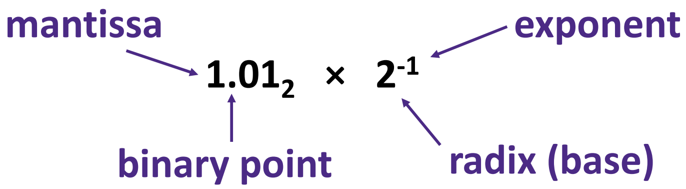
- The signed magnitude, one's complement, and two's complement representation that we have just presented deal with integer values only.
- Without modification, these formats are not useful in scientific or business applications that deal with real number values.
- How do we encode the following:
  - Real numbers (*e.g.* 3.14159)
  - Very large numbers (*e.g.* 6.02×10^{23})
  - Very small numbers (*e.g.* 6.626×10^{-34})
  - Special numbers (*e.g.* ∞, NaN)
- Floating-point representation solves this problem.

Scientific Notation (Decimal)



- *Normalized form*: exactly one digit (non-zero) to left of decimal point
- Alternatives to representing 1/1,000,000,000
  - Normalized: 1.0×10^{-9}
  - Not normalized: $0.1\times10^{-8}, 10.0\times10^{-10}$

Scientific Notation (Binary)



mantissa

$1.01_2 \times 2^{-1}$

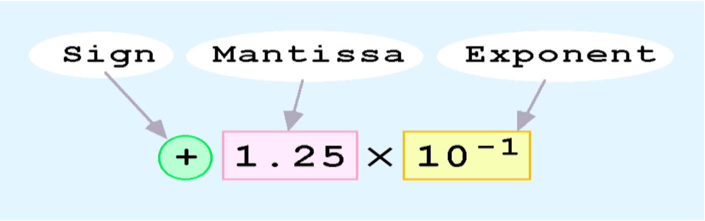
exponent

binary point

radix (base)

- Computer arithmetic that supports this called floating point due to the “floating” of the binary point
  - Declare such variable in C as `float` (or `double`)

- Computers use a form of scientific notation for floating-point representation
- Numbers written in scientific notation have three components:

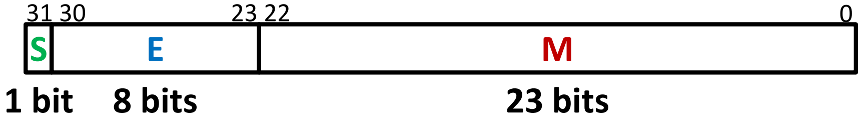


Scientific Notation Translation

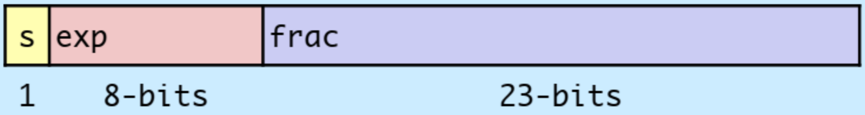
- Convert from scientific notation to binary point
  - Perform the multiplication by shifting the decimal until the exponent disappears
    - Example: $1.011_2 \times 2^4 = 10110_2 = 22_{10}$
    - Example: $1.011_2 \times 2^{-2} = 0.01011_2 = 0.34375_{10}$
- Convert from binary point to *normalized* scientific notation
  - Distribute out exponents until binary point is to the right of a single digit
    - Example: $1101.001_2 = 1.101001_2 \times 2^3$
- **Practice:** Convert 11.375_{10} to binary scientific notation
  - $8+2+1+0.25+0.125 = 1011.001 = 1.011001\text{x}2^3$

Floating Point Encoding

- Use normalized, base 2 scientific notation:
  - Value: $\pm 1 \times \text{Mantissa} \times 2^{\text{Exponent}}$
  - Bit Fields: $(-1)^S \times 1.M \times 2^{(E\text{–bias})}$
- Representation Scheme:
  - Sign bit (0 is positive, 1 is negative)
  - Mantissa (a.k.a. significand) is the fractional part of the number in normalized form and encoded in bit vector **M**
  - Exponent weights the value by a (possibly negative) power of 2 and encoded in the bit vector **E**

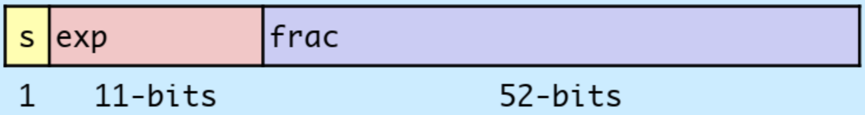


- **Single precision (float)**



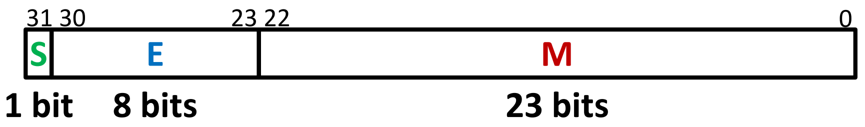
  - **range: ±1.8×10^{-38} – ±3.4×10^{38}, ~7 decimal digits**

- **Double Precision (double)**



  - **range: ±2.23×10^{-308} – ±1.8×10^{308}, ~16 decimal digits**

- Use biased notation
  - Read exponent as unsigned, but with *bias* of $2^{w-1}-1$ = 127
  - Representable exponents roughly ½ positive and ½ negative
  - Exponent 0 (Exp = 0) is represented as E = 0b 0111 1111
- Why biased?
  - Makes floating point arithmetic easier
  - Makes somewhat compatible with two's complement
- **Practice:** To encode in biased notation, add the bias then encode in unsigned:
  - Exp = 1 → 128 → E = 0b 1000 0000
  - Exp = 127→ 254 → E = 0b 1111 1110
  - Exp = -63 → 64 → E = 0b 0100 0000



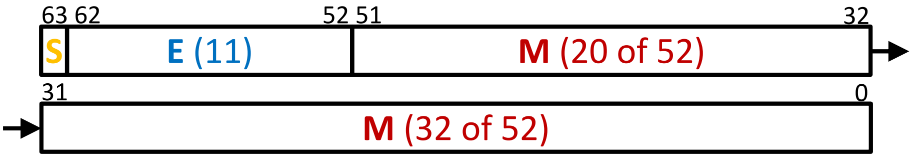
$$(-1)^S \times (1 . M) \times 2^{(E-bias)}$$

- Note the implicit 1 in front of the M bit vector
  - Example: 0b 0011 1111 1100 0000 0000 0000 0000 0000

    is read as $1.1_2 = 1.5_{10}$, *not* $0.1_2 = 0.5_{10}$
  - Gives us an extra bit of *precision*
- Mantissa "limits"
  - Low values near M = 0b0…0 are close to 2^{Exp}
  - High values near M = 0b1…1 are close to 2^{Exp+1}

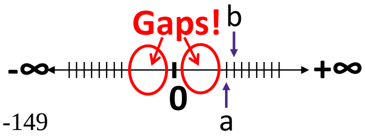
- Precision is a count of the number of bits in a computer word used to represent a value
  - Capacity for accuracy
- Accuracy is a measure of the difference between the *actual value of a number* and its computer representation
  - *High precision permits high accuracy but doesn't guarantee it. It is possible to have high precision but low accuracy.*
  - **Example:** `float pi = 3.14;`
    - `pi` will be represented using all 24 bits of the mantissa (highly precise), but is only an approximation (not accurate)

- Double Precision (vs. Single Precision) in 64 bits



  - C variable declared as `double`
  - Exponent bias is now $2^{10}–1 = 1023$
  - **Advantages:** greater precision (larger mantissa), greater range (larger exponent)
  - **Disadvantages:** more bits used, slower to manipulate

- But wait… what happened to zero?
  - Using standard encoding 0x00000000 =
  - *Special case:* E and M all zeros = 0
    - Two zeros! But at least 0x00000000 = 0 like integers
- New numbers closest to 0:



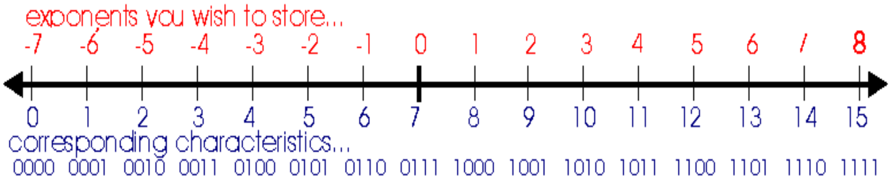
  - $a = 1.0\ldots0_2 \times 2^{-126} = 2^{-126}$
  - $b = 1.0\ldots01_2 \times 2^{-126} = 2^{-126} + 2^{-149}$
  - Normalization and implicit 1 are to blame
  - *Special case:* E = 0, M ≠ 0 are denormalized numbers

Floating-point binary fields

- Signed bit (1): 0 for positive, 1 for negative
- Biased exponent (characteristic): $c = e + 2^{t-1} - 1$

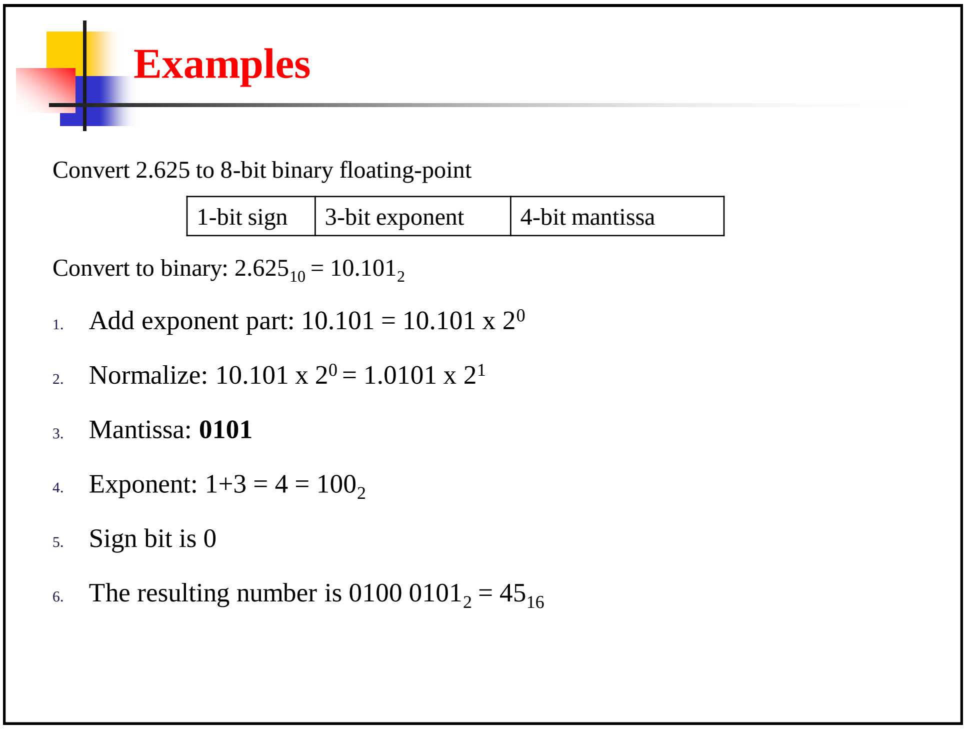
e: exponent needed to store, t: bit size of this field

Ex: need to store the exponent 4 in 6-bit field. $c = 4+2^5 = 36 = 100101$



Converting decimal to floating-point

1. Convert the absolute value of the number to binary,
2. Append × 2^0 to the end of the binary number (which does not change its value),
3. Normalize the number. Move the binary point so that it is one bit from the left. Adjust the exponent of two so that the value does not change,
4. Place the mantissa into the mantissa field of the number. Omit the leading one, and fill with zeros on the right,
5. Add the bias to the exponent of two, and place it in the exponent field,
6. Set the sign bit



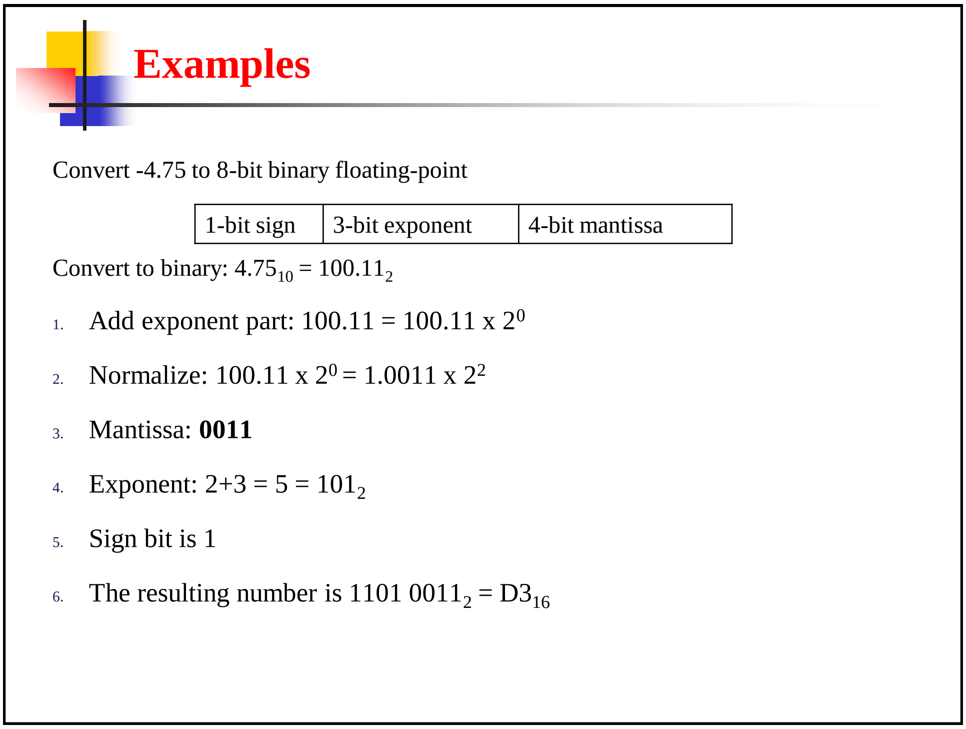
Examples

Convert 2.625 to 8-bit binary floating-point

1-bit sign	3-bit exponent	4-bit mantissa

Convert to binary: $2.625_{10} = 10.101_2$

1. Add exponent part: $10.101 = 10.101 \times 2^0$
2. Normalize: $10.101 \times 2^0 = 1.0101 \times 2^1$
3. Mantissa: **0101**
4. Exponent: $1+3 = 4 = 100_2$
5. Sign bit is 0
6. The resulting number is $0100\ 0101_2 = 45_{16}$



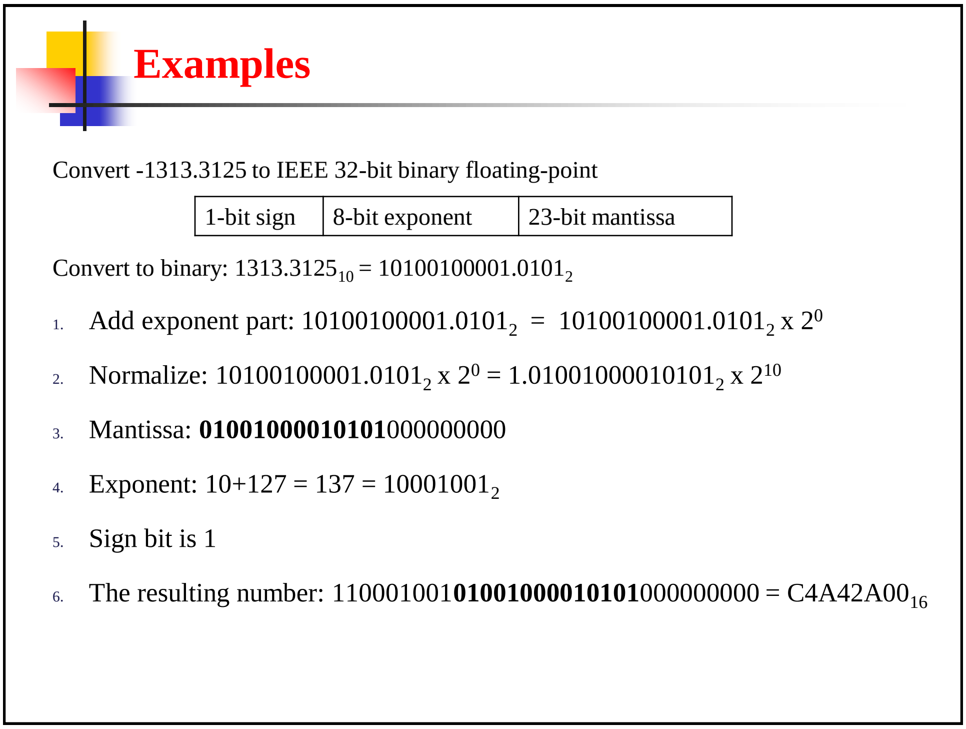
Examples

Convert -4.75 to 8-bit binary floating-point

1-bit sign	3-bit exponent	4-bit mantissa

Convert to binary: $4.75_{10} = 100.11_2$

1. Add exponent part: $100.11 = 100.11 \times 2^0$
2. Normalize: $100.11 \times 2^0 = 1.0011 \times 2^2$
3. Mantissa: **0011**
4. Exponent: $2+3 = 5 = 101_2$
5. Sign bit is 1
6. The resulting number is $1101\ 0011_2 = D3_{16}$



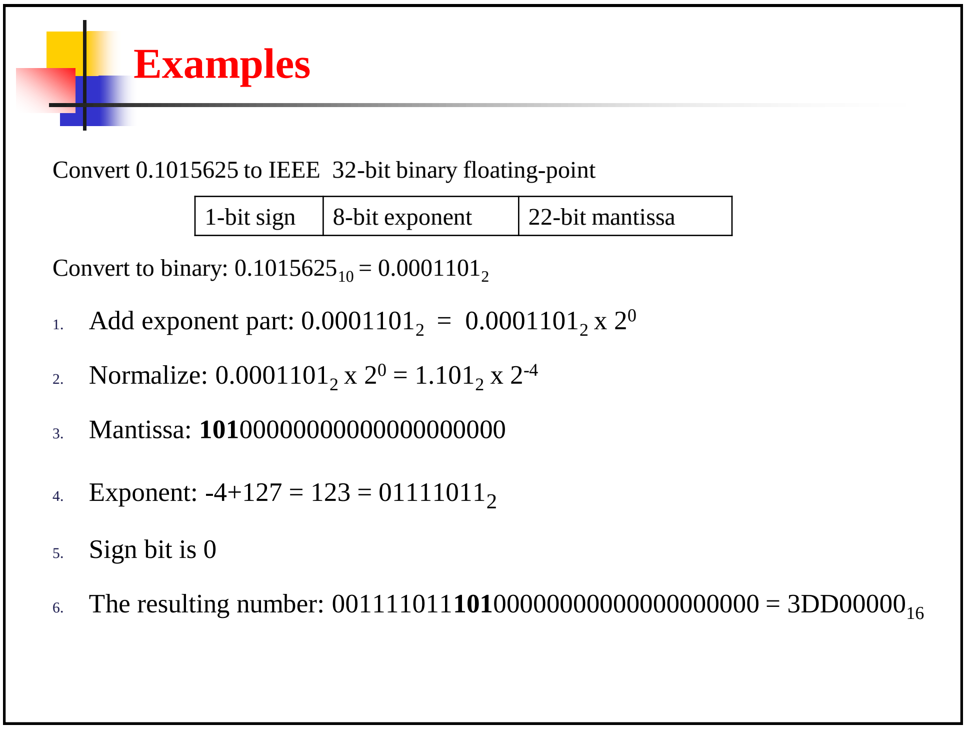
Examples

Convert -1313.3125 to IEEE 32-bit binary floating-point

1-bit sign	8-bit exponent	23-bit mantissa

Convert to binary: $1313.3125_{10} = 10100100001.0101_2$

1. Add exponent part: $10100100001.0101_2 = 10100100001.0101_2 \times 2^0$
2. Normalize: $10100100001.0101_2 \times 2^0 = 1.01001000010101_2 \times 2^{10}$
3. Mantissa: **0100100001010 1**000000000
4. Exponent: $10+127 = 137 = 10001001_2$
5. Sign bit is 1
6. The resulting number: 110001001**01001000010101**000000000 = $C4A42A00_{16}$



Examples

Convert 0.1015625 to IEEE 32-bit binary floating-point

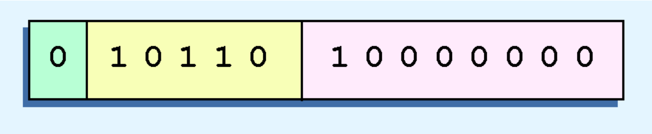
1-bit sign	8-bit exponent	22-bit mantissa

Convert to binary: $0.1015625_{10} = 0.0001101_2$

1. Add exponent part: $0.0001101_2 = 0.0001101_2 \times 2^0$
2. Normalize: $0.0001101_2 \times 2^0 = 1.101_2 \times 2^{-4}$
3. Mantissa: **101**00000000000000000000
4. Exponent: $-4+127 = 123 = 01111011_2$
5. Sign bit is 0
6. The resulting number: 001111011**101**00000000000000000000 = $3DD00000_{16}$

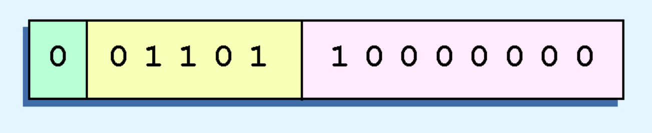
Examples

- Express 32_{10} in the revised 14-bit floating-point model.
- We know that $32 = 1.0 \times 2^5 = 0.1 \times 2^6$.
- To use our excess 16 biased exponent, we add 16 to 6, giving 22_{10} ($=10110_2$).
- Graphically:



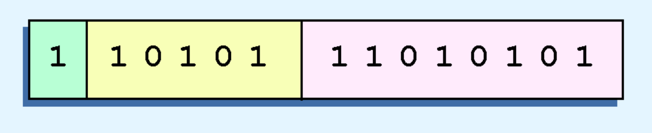
0	1 0 1 1 0	1 0 0 0 0 0 0 0

- Express 0.0625_{10} in the revised 14-bit floating-point model.
- We know that 0.0625 is 2^{-4}. So in (binary) scientific notation $0.0625 = 1.0 \times 2^{-4} = 0.1 \times 2^{-3}$.
- To use our excess 16 biased exponent, we add 16 to -3, giving 13_{10} ($=01101_2$).



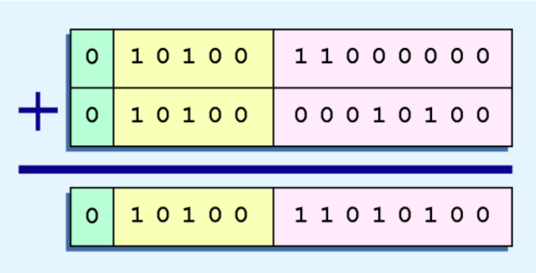
0	0 1 1 0 1	1 0 0 0 0 0 0 0

- Express -26.625$_{10}$ in the revised 14-bit floating-point model.
- We find 26.625$_{10}$ = 11010.101$_2$. Normalizing, we have: 26.625$_{10}$ = 0.11010101 x 2^5.
- To use our excess 16 biased exponent, we add 16 to 5, giving 21$_{10}$ (=10101$_2$). We also need a 1 in the sign bit (for a negative number).



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

- Find the sum of 12$_{10}$ and 1.25$_{10}$ using the 14-bit floating-point model.
- We find 12$_{10}$ = 0.1100 x 2^4. And 1.25$_{10}$ = 0.101 x 2^1 = 0.000101 x 2^4.
- Thus, our sum is 0.110101 x 2 4.

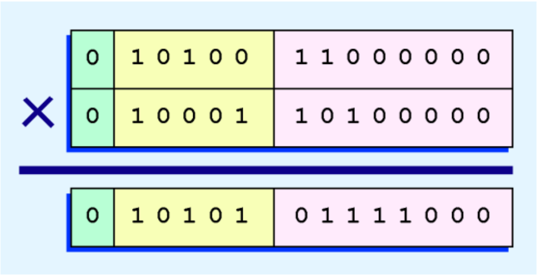


	0	1 0 1 0 0	1 1 0 0 0 0 0 0
+	0	1 0 1 0 0	0 0 0 1 0 1 0 0
	0	1 0 1 0 0	1 1 0 1 0 1 0 0

- Floating-point multiplication is also carried out in a manner akin to how we perform multiplication using pencil and paper.
- We multiply the two operands and add their exponents.
- If the exponent requires adjustment, we do so at the end of the calculation.

Examples

- Find the product of 12_{10} and 1.25_{10} using the 14-bit floating-point model.
- We find $12_{10} = 0.1100 \text{ x } 2^{4}$. And $1.25_{10} = 0.101 \text{ x } 2^{1}$.
- Thus, our product is 0.0111100 x 2 5 = 0.1111 x 2 4.
- The normalized product requires an exponent of $20_{10} = 10110_2$.



	0	1 0 1 0 0	1 1 0 0 0 0 0 0
×	0	1 0 0 0 1	1 0 1 0 0 0 0 0
	0	1 0 1 0 1	0 1 1 1 1 0 0 0