



Jin-Soo Kim  
[\(jinsoo.kim@snu.ac.kr\)](mailto:jinsoo.kim@snu.ac.kr)

Systems Software &  
Architecture Lab.  
Seoul National University

Fall 2019

# Floating Points

# The Problem

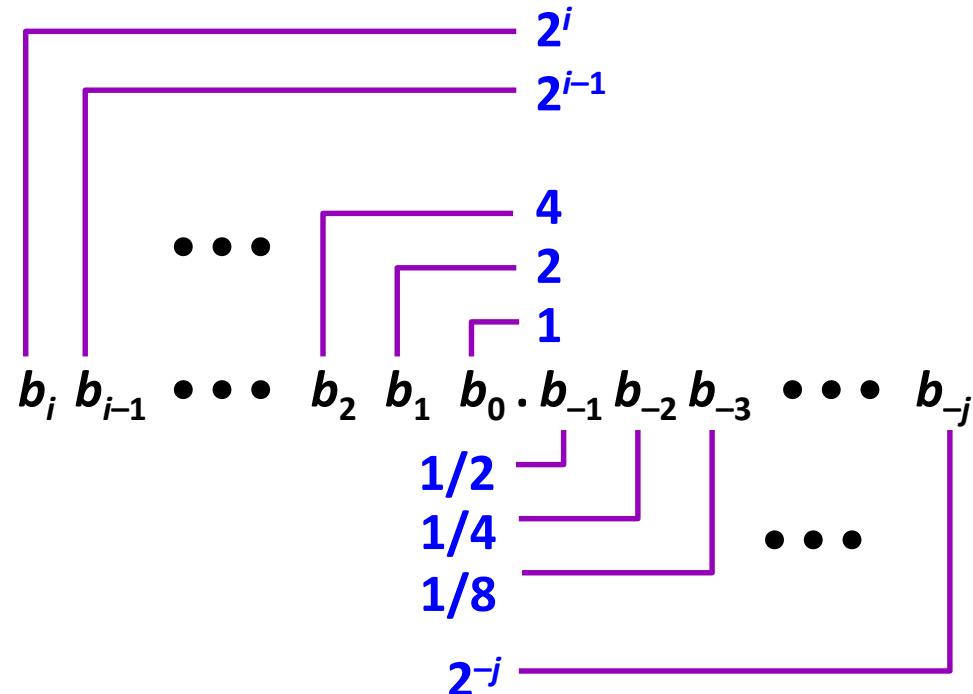
- How to represent fractional values with finite number of bits?
  - 0.1
  - 0.612
  - 3.14159265358979323846264338327950288...
- Wide ranges of numbers
  - 1 Light-Year = 9,460,730,472,580.8 km
  - The radius of a hydrogen atom: 0.00000000025 m

# Fractional Binary Numbers (I)

## ■ Representation

- Bits to right of “binary point” represent fractional powers of 2
- Represents rational number:

$$\sum_{k=-j}^i b_k \cdot 2^k$$



# Fractional Binary Numbers (2)

- Examples:

Value	Representation
$5 \frac{3}{4}$	$101.11_2$
$2 \frac{7}{8}$	$10.111_2$
$\frac{63}{64}$	$0.111111_2$

- Observations

- Divide by 2 by shifting right
- Multiply by 2 by shifting left
- Numbers of form  $0.111111.._2$  just below 1.0
  - $1/2 + 1/4 + 1/8 + \dots + 1/2^i + \dots \rightarrow 1.0$
  - Use notation  $1.0 - \varepsilon$

# Fractional Binary Numbers (3)

- Representable numbers
  - Can only exactly represent numbers of the form  $x / 2^k$
  - Other numbers have repeating bit representations

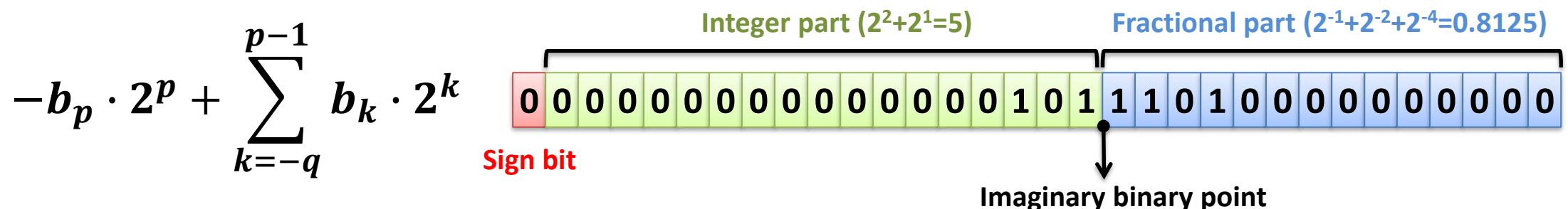
Value	Representation
$1/3$	$0.0101010101[01]..._2$
$1/5$	$0.001100110011[0011]..._2$
$1/10$	$0.0001100110011[0011]..._2$

# Fixed Points

# Fixed-Point Representation (I)

## ■ $p.q$ Fixed-point representation

- Use the rightmost  $q$  bits of an integer as representing a fraction
- Example: 17.14 fixed-point representation
  - 1 bit for sign bit
  - 17 bits for the integer part
  - 14 bits for the fractional part
  - An integer  $x$  represents the real number  $x / 2^{14}$
  - Maximum value:  $(2^{31} - 1) / 2^{14} \approx 131071.999$

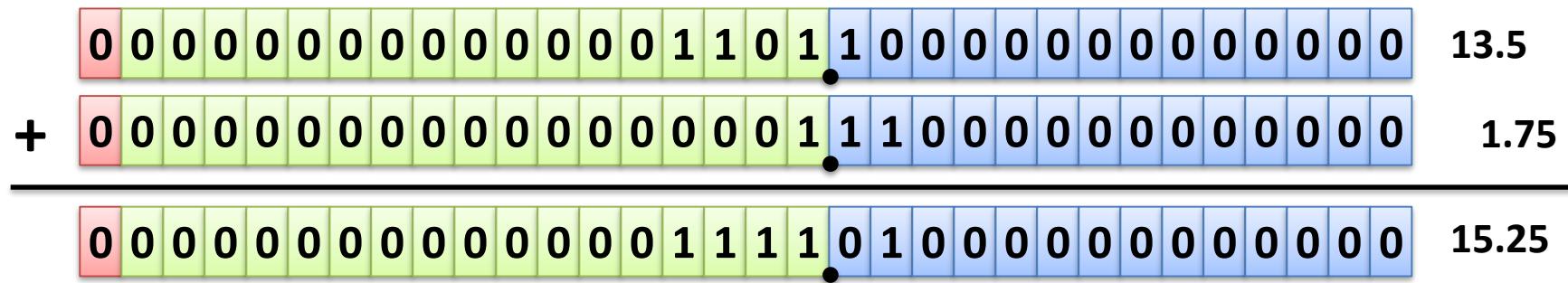


# Fixed-Point Representation (2)

## ■ Properties

- Convert  $n$  to fixed point:
  - Add  $x$  and  $y$ :

$n * f$  ( $= n \ll q$ )  
 $x + y$



- Subtract  $y$  from  $x$ :
  - Add  $x$  and  $n$ :
  - Multiply  $x$  by  $n$ :
  - Divide  $x$  by  $n$ :

$x - y$   
 $x + n * f$   
 $x * n$   
 $x / n$

$x, y$ : fixed-point number  
 $n$ : integer  
 $f = 1 << q$

# Fixed-Point Representation (3)

## ■ Pros

- Simple range 변하지 않는 경우 better be used!
- Can use integer arithmetic to manipulate
- No floating-point hardware needed
- Used in many low-cost embedded processors or DSPs (digital signal processors)

## ■ Cons

- Cannot represent wide ranges of numbers

# Representing Floating Points

Chap. 3.5

# Representing Floating Points

- IEEE standard 754
  - Established in 1985 as uniform standard for floating-point arithmetic
    - Before that, many idiosyncratic formats
    - Portability issues for scientific code
  - Supported by all major CPUs
  - William Kahan, a primary architect of IEEE 754, won the Turing Award in 1989
  - Driven by numerical concerns
    - Nice standards for rounding, overflow, underflow
    - Hard to make go fast
    - Numerical analysts predominated over hardware types in defining standard

# FP Representation

즉 \*sign magnitude 방법 -> 0 표현 두 개이다. (0000 / 10000)

- Numerical form:  $-I^s \times M \times 2^E$

- Sign bit **s** determines whether number is negative or positive
- Significand **M** normally a fractional value in range [1.0, 2.0)
- Exponent **E** weights value by power of two  
normalized form (0이 아닌 경우) 1.xxx로 표현 가능

- Encoding



- MSB is sign bit **s** (0: non-negative, 1: negative)
- **exp** field encodes **E** (Exponent)
- **frac** field encodes **M** (Mantissa)  
fractional mantissa

# FP Precisions

**frac** 파트가 길다 -> 좀 더 **precise**한, 정밀한 수 표현,  
**exp** 파트가 길다 -> 수 표현 범위가 는다. 얼마나 작은수를  
혹은 얼마나 큰 수를 나타낼수 있는지



- Single precision `float f;` 임의로 정할수도,
    - 8 **exp** bits, 23 **frac** bits (32 bits total)
  - Double precision `double d;`
    - 11 **exp** bits, 52 **frac** bits (64 bits total)
  - Extended precision
    - 15 **exp** bits, 63 **frac** bits intel만 지원,
    - Only found in Intel-compatible machines
    - Stored in 80 bits (1 bit wasted)

# Normalized Values

- Condition:  $\text{exp} \neq 000\dots0$  and  $\text{exp} \neq 111\dots1$
- Exponent coded as a biased value
  - $E = Exp - Bias$
  - $Exp$ : unsigned value denoted by  $\text{exp}$
  - $Bias$ : Bias value ( $=2^{k-1}-1$ , where  $k$  is the number of  $\text{exp}$  bits)
    - Single precision ( $k=8$ ): 127 ( $Exp: 1..254, E: -126..127$ )
    - Double precision ( $k=11$ ): 1023 ( $Exp: 1..2046, E: -1022..1023$ )
- Significand coded with implied leading 1
  - $M = 1.x_{k-1}\dots x_0$ 
    - Minimum when  $frac = 000\dots0$  ( $M = 1.0$ )
    - Maximum when  $frac = 111\dots1$  ( $M = 2.0 - \varepsilon$ )
  - Get extra leading bit for “free”

# Normalized Values: Example

- float  $f = 2003.0;$ 
  - $2003_{10} = 1111010011_2 = 1.\underline{1111010011}_2 \times 2^{10}$
- Significand
  - **M** = 1.1111010011<sub>2</sub>
  - **frac** = 111101001100000000000000<sub>2</sub>
- Exponent
  - **E** = 10
  - **Exp** =  $E + Bias = 10 + 127 = 137 = 10001001_2$

Hex:	4	4	F	A	6	0	0	0
Binary:	0100	0100	1111	1010	0110	0000	0000	0000
137:	100	0100	1					
2003:				1111	1010	0110		

# Denormalized Values

- Condition:  $\text{exp} = 000\dots0$
- Value
  - Exponent value  $E = l - \text{Bias}$
  - Significand value  $M = 0.\text{xxx}\dots x_2$  (no implied leading 1)
- Case 1:  $\text{exp} = 000\dots0, \text{frac} = 000\dots0$ 
  - Represents value 0.0      0에 근사한 값들의 정밀한 표현 위해
  - Note that there are distinct values +0 and -0
- Case 2:  $\text{exp} = 000\dots0, \text{frac} \neq 000\dots0$ 
  - Numbers very close to 0.0
  - “Gradual underflow”: possible numeric values are spaced evenly near 0.0

# Special Values

- Condition:  $\exp = 111\dots1$

이 경우는 모든 합차곱나누기 모두 infinity

- Case 1:  $\exp = 111\dots1, \text{frac} = 000\dots0$

- Represents value  $\infty$  (infinity)
- Operation that overflows
- Both positive and negative 혹은 계산의 결과가 너무 클때!
- e.g.  $1.0/0.0 = -1.0/-0.0 = +\infty, 1.0/-0.0 = -\infty$

- Case 2:  $\exp = 111\dots1, \text{frac} \neq 000\dots0$

- Not-a-Number (NaN)
- Represents case when no numeric value can be determined
- e.g.  $\sqrt{-1}, \infty - \infty, \infty * 0.0, \dots$

# Tiny FP Example (I)

- 8-bit floating point representation
  - The sign bit is in the most significant bit
  - The next four bits are the **exp**, with a bias of 7
  - The last three bits are the **frac**
- Same general form as IEEE format
  - Normalized, denormalized
  - Representation of 0, NaN, infinity



# Tiny FP Example (2)

- Values related to the exponent (**Bias = 7**)

Description	Exp	exp	E = Exp - Bias	$2^E$
Denormalized	0	0000	-6	1/64
Normalized	1	0001	-6	1/64
	2	0010	-5	1/32
	3	0011	-4	1/16
	4	0100	-3	1/8
	5	0101	-2	1/4
	6	0110	-1	1/2
	7	0111	0	1
	8	1000	1	2
	9	1001	2	4
	10	1010	3	8
	11	1011	4	16
	12	1100	5	32
	13	1101	6	64
	14	1110	7	128
inf, NaN	15	1111	-	-

유일하게 다른점:  
이건 1일때랑 같이, E: -6

# Tiny FP Example (3)

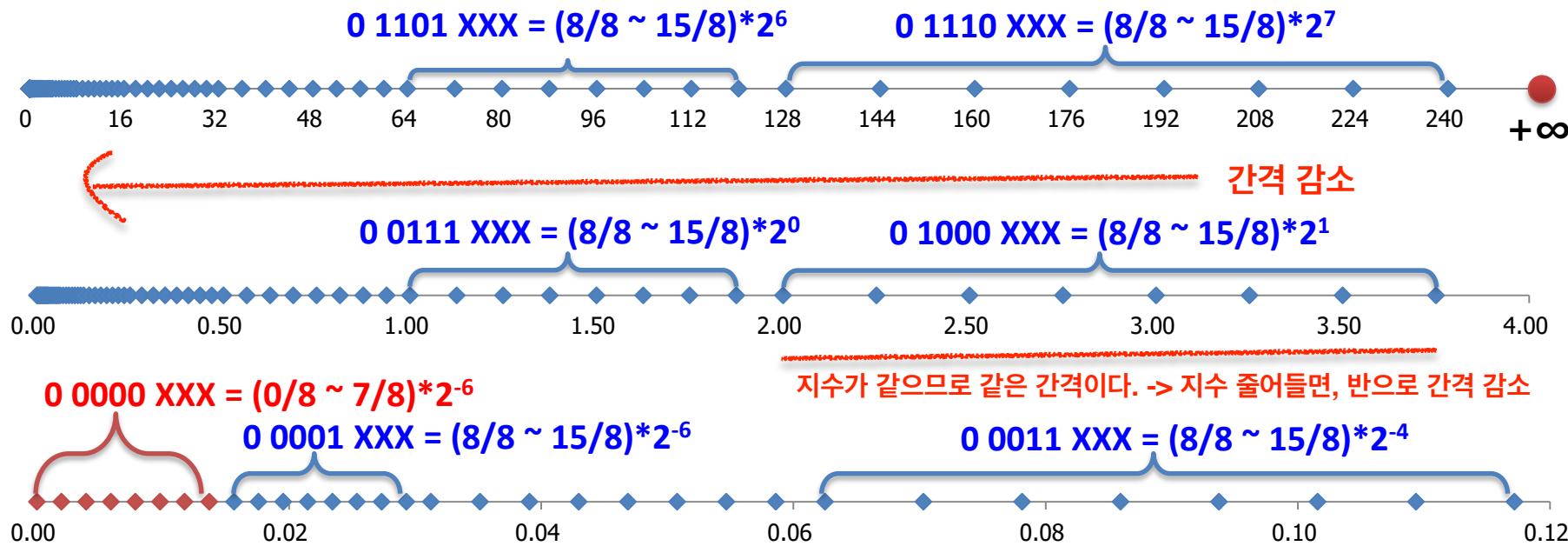
- Dynamic range

Description	Bit representation	e	E	f	M	V
Zero	0 0000 000	0	-6	0	0	0
Smallest pos.	0 0000 001	0	-6	1/8	1/8	1/512
	0 0000 010	0	-6	2/8	2/8	2/512
	0 0000 011	0	-6	3/8	3/8	3/512
	0 0000 110	0	-6	6/8	6/8	6/512
	0 0000 111	0	-6	7/8	7/8	7/512
Largest denorm.	0 0001 000	1	-6	0	8/8	8/512
	0 0001 001	1	-6	1/8	9/8	9/512
	0 0110 110	6	-1	6/8	14/8	14/16
	0 0110 111	6	-1	7/8	15/8	15/16
	0 0111 000	7	0	0	8/8	1
	0 0111 001	7	0	1/8	9/8	9/8
One	0 0111 010	7	0	2/8	10/8	10/8
	0 1110 110	14	7	6/8	14/8	224
	0 1110 111	14	7	7/8	15/8	240
Infinity	0 1111 000	-	-	-	-	$+\infty$

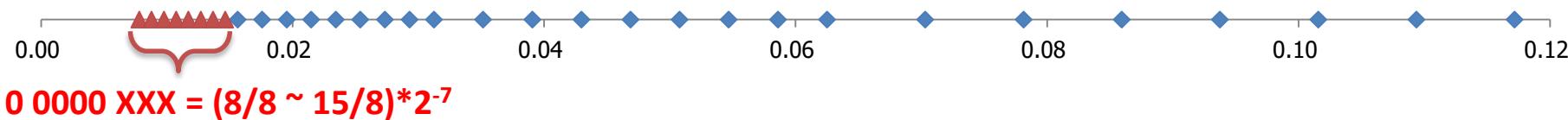
M : 0.xxxxx

# Tiny FP Example (4)

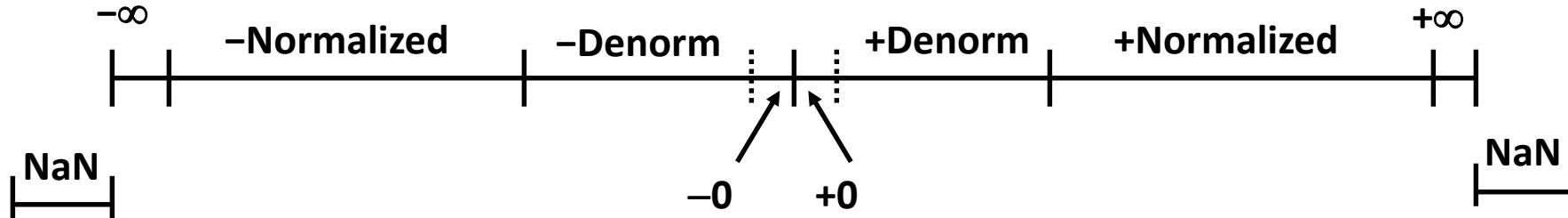
- Encoded values (nonnegative numbers only)



(Without denormalization) 0과 그 주변을 채우지 못한다. exponent 아무리 크더라도. : denormalized form 그래서 필요



# Interesting Numbers



Description	exp	frac	Numeric Value
Zero	000 ... 00	000 ... 00	0.0
Smallest Positive denormalized	000 ... 00	000 ... 01	Single: $2^{-23} \times 2^{-126} \approx 1.4 \times 10^{-45}$ Double: $2^{-52} \times 2^{-1022} \approx 4.9 \times 10^{-324}$
Largest Denormalized	000 ... 00	111 ... 11	Single: $(1.0 - \epsilon) \times 2^{-126} \approx 1.18 \times 10^{-38}$ Double: $(1.0 - \epsilon) \times 2^{-1022} \approx 2.2 \times 10^{-308}$
Smallest Positive Normalized	000 ... 01	000 ... 00	Single: $1.0 \times 2^{-126}$ , Double: $1.0 \times 2^{-1022}$ (Just larger than largest denormalized)
One	011 ... 11	000 ... 00	1.0
Largest Normalized	111 ... 10	111 ... 11	Single: $(2.0 - \epsilon) \times 2^{127} \approx 3.4 \times 10^{38}$ Double: $(2.0 - \epsilon) \times 2^{1023} \approx 1.8 \times 10^{308}$

# IEEE FP16 vs. Google Bfloat16



## ■ Google bfloat16

- Introduced by Google in 2018 for TPUs (Supported by Intel NPUs too)
- Same dynamic range as FP32
- Smaller mantissa reduces power and physical silicon area

# Manipulating Floating Points

Chap. 3.5, 3.9 – 3.10

# Special Properties

- FP zero same as integer zero
  - All bits = 0
- Can (almost) use unsigned integer comparison
  - Must first compare sign bits
  - Must consider  $-0 = 0$
  - NaNs problematic
    - Will be greater than any other values
  - Otherwise OK
    - Denormalized vs. normalized
    - Normalized vs. Infinity

위 제외 시 unsigned integer처럼 크기 비교시  
굳이 bias 계산 안해도 ok

# Rounding

- For a given value  $x$ , finding the “closest” matching value  $x'$  that can be represented in the FP format
- IEEE 754 defines four rounding modes
  - Round-to-even avoids statistical bias by rounding upward or downward so that the least significant digit is even

Rounding modes	\$1.40	\$1.60	\$1.50	\$2.50	\$-1.50
Round-toward-zero	\$1	\$1	\$1	\$2	\$-1
Round-down ( $-\infty$ )	\$1	\$1	\$1	\$2	<u>\$-2</u>
Round-up ( $+\infty$ )	\$2	\$2	\$2	\$3	<u>\$-1</u>
<b>Round-to-even (default) or Round-to-nearest</b>	\$1	\$2	\$2	\$2	\$-2

짝수니까 무조건 2!

반올림과 비슷하나, 짝수 쪽으로 이동  
1.5 2.5등 가운데 있는 수의 경우

# Round-to-Even

올림. 잘라낸다-> 버림 (수가 작아짐)

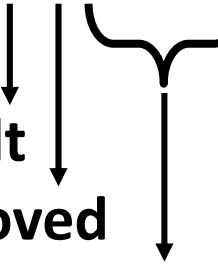
## ■ Round up conditions

- R = I, S = I  $\rightarrow > 0.5$
- G = I, R = I, S = 0  
 $\rightarrow$  Round to even

잘리기 직전 bit

잘리는 첫번째 bit

1. BBGRXXX



Guard bit: LSB of result

Round bit: 1<sup>st</sup> bit removed

Sticky bit: OR of remaining bits

Value	Fraction	GRS	Up?	Rounded
128	1.0000000 ( $\times 2^7$ )	000	No	1.000
13	1.1010000 ( $\times 2^3$ )	100	No	1.101
17	1.0001000 ( $\times 2^4$ )	010	No	1.000
19	1.0011000 ( $\times 2^4$ )	110	Yes	1.010
138	1.0001010 ( $\times 2^7$ )	011	Yes	1.001
63	1.1111100 ( $\times 2^5$ )	111	Yes	10.000

round bit 1,  
1) sticky bit == 0  
if(G == 0) 버림  
if(G != 0) 올림  
2) sticky bit != 0  
올림

round bit 0  
버림

이 경우에는 다시  
normalize 해야한다\*\*\*\*

# FP Addition

- Adding two numbers:

(Assume  $E1 > E2$ )

- I. Align binary points

- Shift number with smallest exponent
- Shift right  $M2$  by  $E1 - E2$

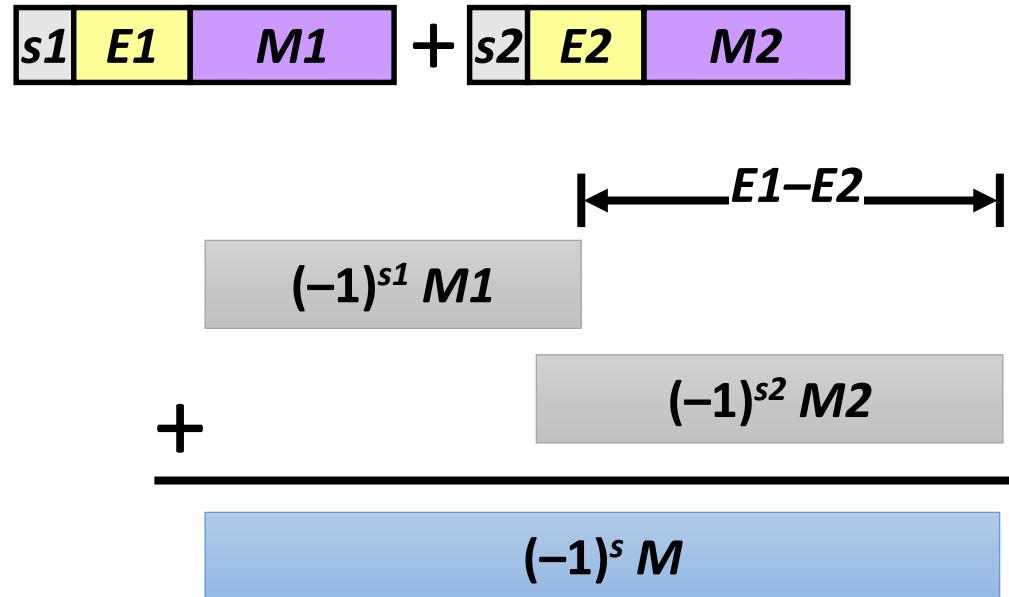
2. Add significands

- Result: Sign  $s$ , Significand  $M$ , Exponent  $E$  ( $= E1$ )

3. Normalize result & check for over/underflow

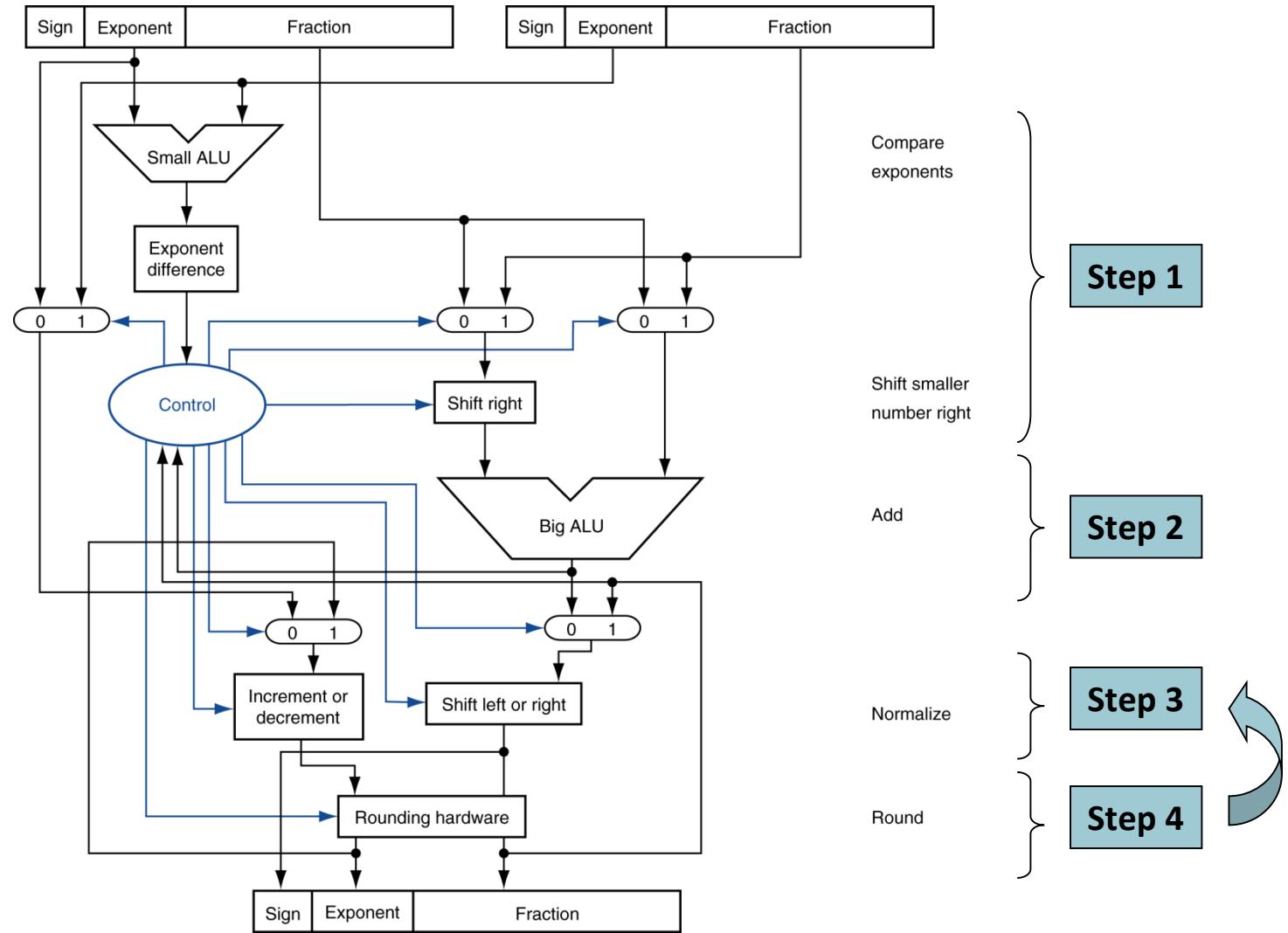
- if ( $M \geq 2$ ), shift  $M$  right, increment  $E$
- if ( $M < 1$ ), shift  $M$  left  $k$  positions, decrement  $E$  by  $k$

4. Round  $M$  and renormalize if necessary



# FP Adder Hardware

- Much more complex than integer adder



# FP Multiplication

- Multiplying two numbers:

1. Add exponents

- $E = E1 + E2$

2. Multiply significands

- $M = M1 \times M2$

3. Normalize result & check for over/underflow

check for overflow (E out of range?)

- if ( $M \geq 2$ ), shift  $M$  right, increment  $E$
  - if ( $M < 1$ ), shift  $M$  left  $k$  positions, decrement  $E$  by  $k$

4. Round  $M$  and renormalize if necessary

5. Determine sign

- $s = s1 \wedge s2$



# FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root, ...
  - FP  $\leftrightarrow$  integer conversion
- Completing an operation in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- Operations usually take several cycles
  - Can be pipelined

# Floating Points in C

- C guarantees two levels
  - **float** (single precision) vs. **double** (double precision)
- Conversions
  - **double or float → int**
    - Truncates fractional part
    - Like rounding toward zero
    - Not defined when out of range or NaN (Generally sets to TMin)
  - **int → double**
    - Exact conversion, as long as int has  $\leq$  53 bit word size
  - **int → float**
    - Will round according to rounding mode

# FP Example I

```
#include <stdio.h>

int main ()
{
    int n = 123456789;
    int nf, ng;
    float f;
    double g;

    f = (float) n;           123456792가 된다.
    g = (double) n;          bit가 길기 때문,
    nf = (int) f;
    ng = (int) g;
    printf ("nf=%d ng=%d\n", nf, ng);
}
```

# FP Example 2

```
#include <stdio.h>

int main ()
{
    double d;      잘라서 저장, -> 2.00000088828 =>
                    rounding 등등으로 인한 값의 변화
                    floating point 계산 결과 비교 조심해야한다.
    d = 1.0 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 +
        + 0.1 + 0.1 + 0.1 + 0.1 + 0.1;

    printf ("d = %.20f\n", d);
}
```

# FP Example 3

```
#include <stdio.h>

int main ()
{
    float f1 = (3.14 + 1e20) - 1e20;
    float f2 = 3.14 + (1e20 - 1e20);

    printf ("f1 = %f, f2 = %f\n", f1, f2);
}
```

3.14는 지수가 1e20에 맞춰지니까 저 밑에  
f1 0  
f2 3.14

지수가 비슷한 것끼리 먼저 계싼 -> 잃어버리는 것 최소화  
결합법칙 성립하지 않는다. 분배 법칙도?????

# Ariane 5

- Ariane 5 tragedy (June 4, 1996)
  - Exploded 37 seconds after liftoff
  - Satellites worth \$500 million
- Why?
  - Computed horizontal velocity as floating-point number
  - Converted to 16-bit integer
    - Careful analysis of Ariane 4 trajectory proved 16-bit is enough
  - Reused a module from 10-year-old software
    - Overflowed for Ariane 5
    - No precise specification for the software



# Byte Ordering

Chap. 2.3, 2.9

# Data Types in C

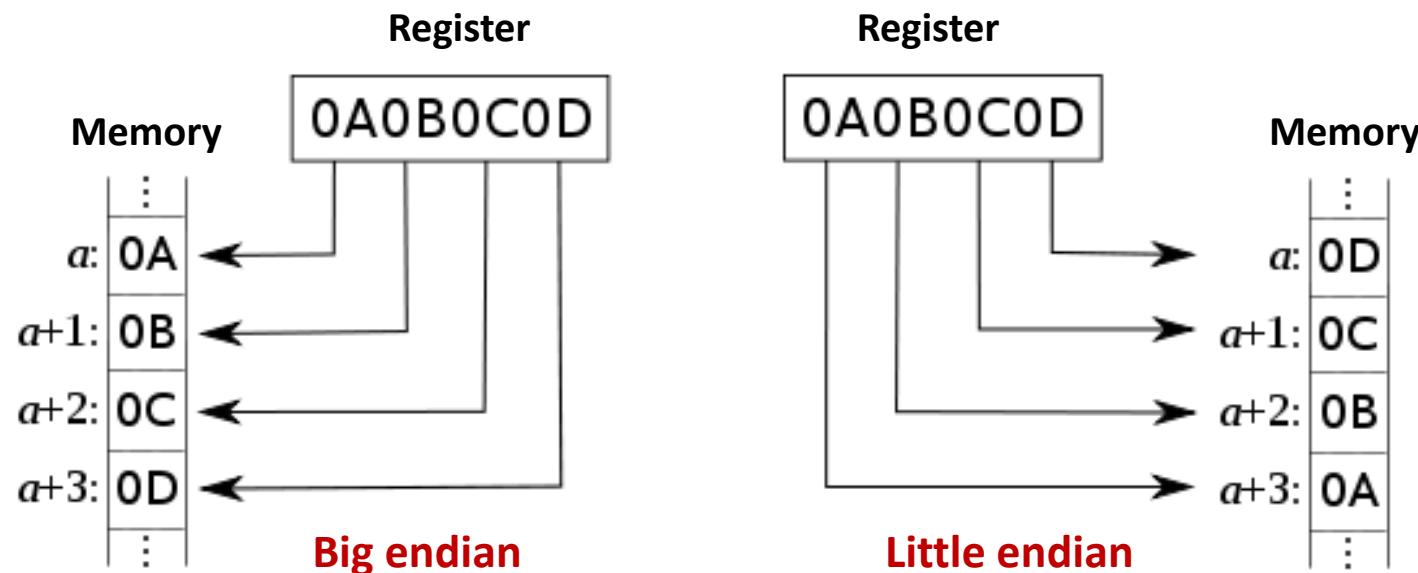
C Data Type	Typical 32-bit	Typical 64-bit	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	8	8
long long	8	8	8
float	4	4	4
double	8	8	8
long double	-	-	10/16
pointer	4	8	8

# Byte Ordering

- How are the bytes within a multi-byte word ordered in memory?
- Conventions
  - Big endian: Sun, PowerPC Mac, Internet
  - Little endian: Intel x86, ARM running Android & iOS, RISC-V
- Note:
  - Alpha and PowerPC can run in either mode, with the byte ordering convention determined when the chip is powered up
  - Problem when the binary data is communicated over a network between different machines

# Big vs. Little Endian

- Big endian
  - Least significant byte has highest address
- Little endian
  - Least significant byte has lowest address



# Example 1

- Disassembly
  - Text representation of binary machine code
  - Generated by program that reads the machine code
- Example fragment

Address	Instruction Code	Assembly Rendition
8048365:	5b	pop %ebx
8048366:	81 c3 ab 12 00 00	add \$0x12ab,%ebx
804836c:	83 bb 28 00 00 00 00	cmpl \$0x0,0x28(%ebx)

- Deciphering numbers:

Value: 0x12ab  
Pad to 32 bits: 0x000012ab  
Split into bytes: 00 00 12 ab  
Reverse: ab 12 00 00

# Example 2

- What is the output of this program?

- Solaris/SPARC: ?
- Linux/x86-64: ?

```
#include <stdio.h>

union {
    int          i;
    unsigned char c[4];
} u;

int main () {
    u.i = 0x12345678;
    printf ("%x %x %x %x\n",
            u.c[0], u.c[1], u.c[2], u.c[3]);
}
```

# Character Sets

- ASCII: 128 characters

- 95 graphic, 33 control

- Latin-1: 256 characters

- ASCII + 96 more graphic characters

- Unicode: 32-bit character set

- Used in Java, C++ wide characters, ...
- Most of the world's alphabets, plus symbols
- UTF-8, UTF-16: variable-length encodings

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

# Representing Strings

## ■ Strings in C

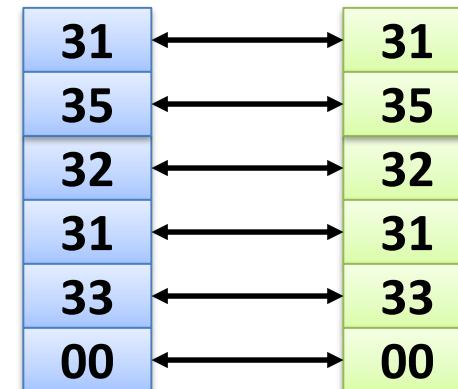
- Represented by array of characters
- Each character encoded in ASCII format
  - Standard 7-bit encoding of character set
  - Character ‘0’ has code 0x30
  - Digit  $i$  has code  $0x30 + i$
- String should be null-terminated
  - Final character = 0x00

## ■ Compatibility

- Byte ordering not an issue

```
char S[6] = "15213";
```

Linux/Alpha S Sun S



# Summary

- Floating points important for scientific code (and machine learning)
- Computer representations of numbers
  - Finite range and precision
  - Overflow/underflow
  - Floating points violate associativity / distributivity
  - Different machines follow different byte ordering
- Bits have no inherent meaning
  - Interpretation depends on the instructions applied