

SWE2001: System Program

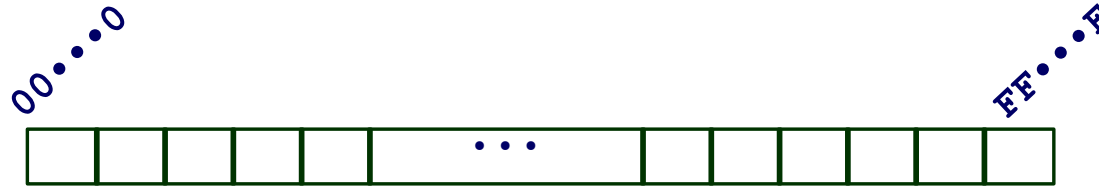
Lecture 0x03: Bits, Bytes, and Integers - 3

Hojoon Lee

Today: Bits, Bytes, and Integers

- ▶ Representing information as bits
- ▶ Bit-level manipulations
- ▶ Integers
 - Representation: unsigned and signed
 - Conversion, casting
 - Expanding, truncating
 - Addition, negation, multiplication, shifting
 - Summary
- ▶ Representations in memory, pointers, strings

Byte-Oriented Memory Organization



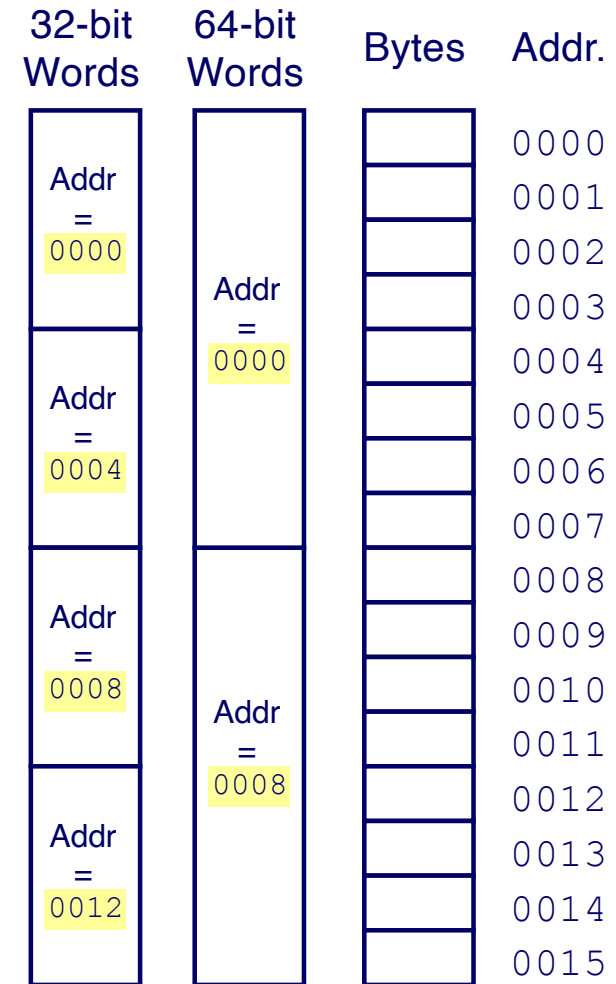
- ▶ Programs refer to data by address
 - Conceptually, envision it as a very large array of bytes
 - In reality, it's not, but can think of it that way
 - An address is like an index into that array
 - and, a pointer variable stores an address
- ▶ Note: system provides private address spaces to each “process”
 - Think of a process as a program being executed
 - So, a program can clobber its own data, but not that of others

Machine Words

- ▶ Any given computer has a “Word Size”
 - Nominal size of integer-valued data
 - and of addresses
 - Until recently, most machines used 32 bits (4 bytes) as word size
 - Limits addresses to 4GB (2^{32} bytes)
 - Increasingly, machines have 64-bit word size
 - Potentially, could have 18 EB (exabytes) of addressable memory
 - That's 18.4×10^{18}
 - Machines still support multiple data formats
 - Fractions or multiples of word size
 - Always integral number of bytes

Word-Oriented Memory Organization

- ▶ Addresses Specify Byte Locations
 - Address of first byte in word
 - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



Example Data Representations

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
float	4	4	4
double	8	8	8
long double	–	–	10/16
pointer	4	8	8

Byte Ordering

- ▶ So, how are the bytes within a multi-byte word ordered in memory?
- ▶ Conventions
 - Big Endian: Sun, PPC Mac, Internet
 - Least significant byte has highest address
 - Little Endian: x86, ARM processors running Android, iOS, and Windows
 - Least significant byte has lowest address

Byte Ordering Example

► Example

- Variable x has 4-byte value of 0x01234567
- Address given by &x is 0x100

Big Endian

0x100	0x101	0x102	0x103		
01	23	45	67		

Little Endian

0x100	0x101	0x102	0x103		
67	45	23	01		

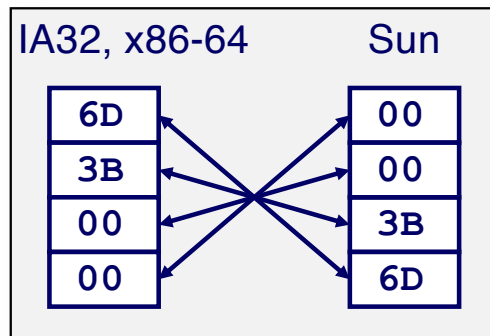
Representing Integers

Decimal: 15213

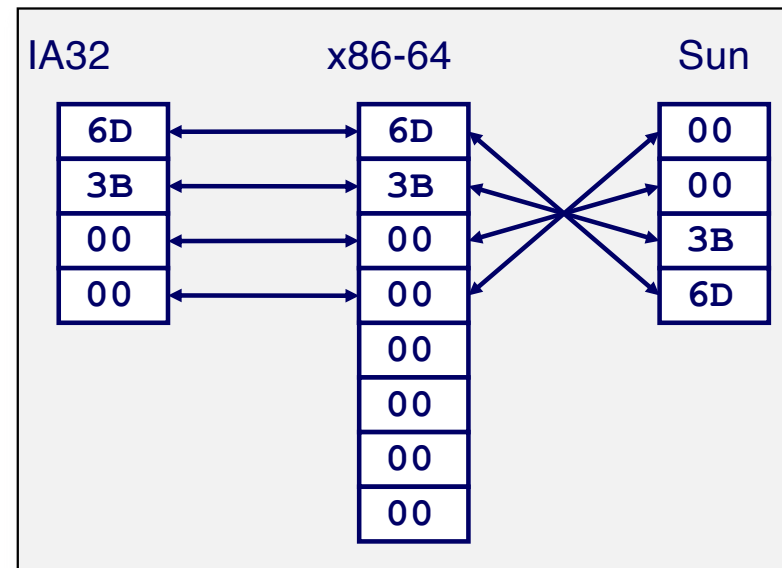
Binary: 0011 1011 0110 1101

Hex: 3 B 6 D

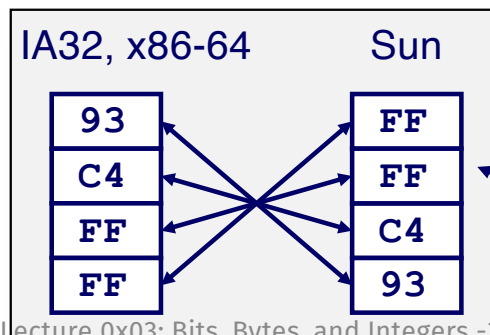
```
int A = 15213;
```



```
long int C = 15213;
```



```
int B = -15213;
```



Two's complement representation

Examining Data Representations

- ▶ Code to Print Byte Representation of Data
 - Casting pointer to unsigned char * allows treatment as a byte array

```
typedef unsigned char *pointer;

void show_bytes(pointer start, size_t len){
    size_t i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.2x\n", start+i, start[i]);
    printf("\n");
}
```

Printf directives:

%p: Print pointer

%x: Print Hexadecimal

show_bytes Execution Example

```
int a = 15213;  
printf("int a = 15213;\n");  
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux x86-64):

```
int a = 15213;  
0x7fffb7f71dbc    6d  
0x7fffb7f71dbd    3b  
0x7fffb7f71dbe    00  
0x7fffb7f71dbf    00
```

Representing Pointers

```
int B = -15213;  
int *P = &B;
```

Sun	IA32	x86-64
EF	AC	3C
FF	28	1B
FB	F5	FE
2C	FF	82
		FD
		7F
		00
		00

Different compilers & machines assign different locations to objects

Even get different results each time run program

Lecture 0x03: Bits, Bytes, and Integers -3

Representing Strings

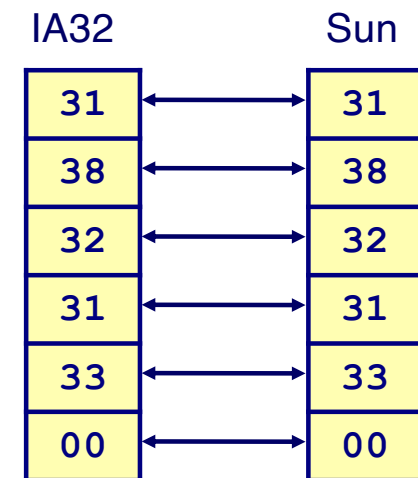
► Strings in C

- 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 = 0

► Compatibility

- Byte ordering not an issue

```
char S[6] = "18213";
```



Floating Point

- ▶ Background: Fractional binary numbers
- ▶ IEEE floating point standard: Definition
- ▶ Example and properties
- ▶ Rounding, addition, multiplication
- ▶ Floating point in C
- ▶ Summary

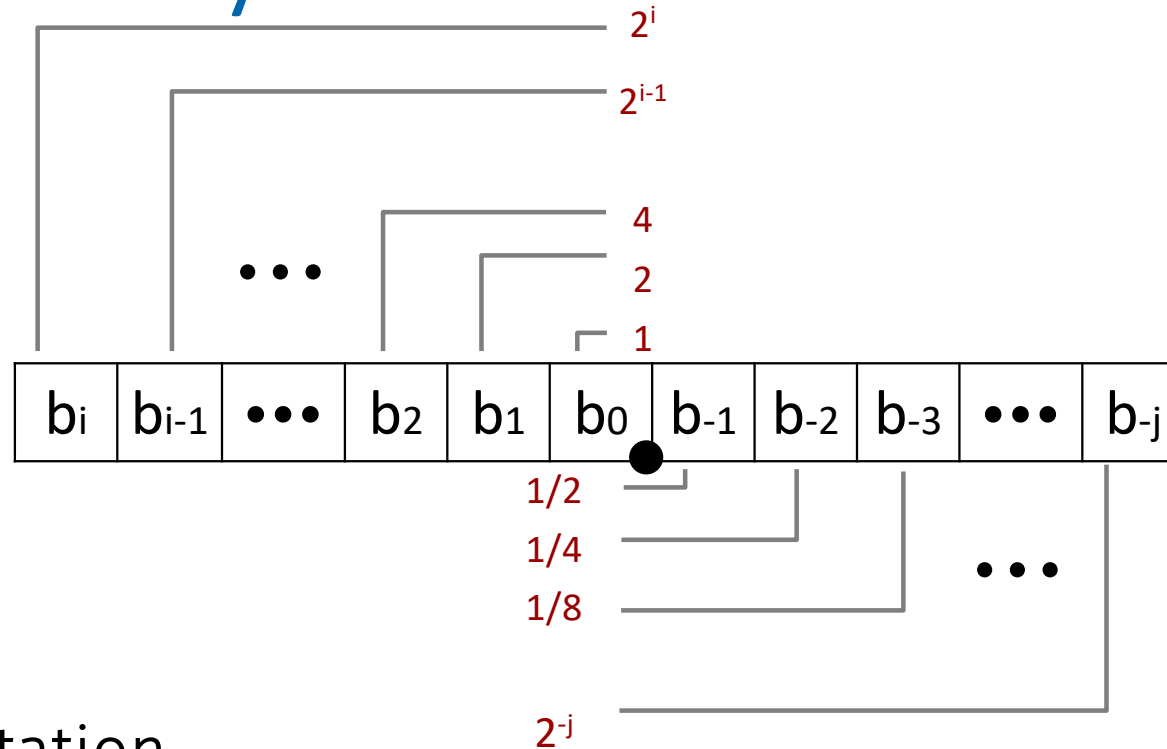
Floating Point

- ▶ Background: Fractional binary numbers
- ▶ IEEE floating point standard: Definition
- ▶ Example and properties
- ▶ Rounding, addition, multiplication
- ▶ Floating point in C
- ▶ Summary

Fractional binary numbers

- ▶ What is 1011.101_2 ?

Fractional Binary Numbers



► Representation

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

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

Fractional Binary Numbers: Examples

Value	Representation
$5 \frac{3}{4}$	101.11_2
$2 \frac{7}{8}$	10.111_2
$1 \frac{7}{16}$	1.0111_2

Observations

Divide by 2 by shifting right (unsigned)

Multiply by 2 by shifting left

Numbers of form $0.111111\dots_2$ are just below 1.0

$$1/2 + 1/4 + 1/8 + \dots + 1/2^i + \dots \rightarrow 1.0$$

Use notation $1.0 - \epsilon$

Representable Numbers

- ▶ Limitation #1
 - Can only exactly represent numbers of the form $x/2^k$
 - Other rational 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**
- ▶ Limitation #2
 - Just one setting of binary point within the w bits
 - Limited range of numbers (very small values? very large?)

Today: Floating Point

- ▶ Background: Fractional binary numbers
- ▶ IEEE floating point standard: Definition
- ▶ Example and properties
- ▶ Rounding, addition, multiplication
- ▶ Floating point in C
- ▶ Summary

IEEE Floating Point

- ▶ IEEE Standard 754
 - Established in 1985 as uniform standard for floating point arithmetic
 - Before that, many idiosyncratic formats
 - Supported by all major CPUs
- ▶ Driven by numerical concerns
 - Nice standards for rounding, overflow, underflow
 - Hard to make fast in hardware
 - Numerical analysts predominated over hardware designers in defining standard

Floating Point Representation

- ▶ Numerical Form:

$$(-1)^s \mathbf{M} 2^E$$

- 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

- ▶ Encoding

- MSB **s** is sign bit **s**
- **exp** field encodes **E** (but is not equal to E)
- **frac** field encodes **M** (but is not equal to M)



Precision options

- ▶ Single precision: 32 bits



- ▶ Double precision: 64 bits



- ▶ Extended precision: 80 bits (Intel only)



“Normalized” Values

- ▶ When: $\text{exp} \neq 000\dots 0$ and $\text{exp} \neq 111\dots 1$
- ▶ Exponent coded as a **biased** value: $E = \text{Exp} - \text{Bias}$
 - **Exp**: unsigned value of **exp field**
 - **Bias** = $2^{k-1} - 1$, where **k** is number of exponent bits
 - Single precision: 127 (Exp: 1...254, E: -126...127)
 - Double precision: 1023 (Exp: 1...2046, E: -1022...1023)
- ▶ Significand coded with implied leading 1: $M = 1.\text{xxx}\dots\text{x}_2$
 - **xxx...x**: bits of **frac field**
 - Minimum when **frac**=000...0 ($M = 1.0$)
 - Maximum when **frac**=111...1 ($M = 2.0 - \epsilon$)
 - Get extra leading bit for “free”

$$v = (-1)^s M 2^E$$

Normalized Encoding Example

$$v = (-1)^s M 2^E$$
$$E = \text{Exp} - \text{Bias}$$

► Value: `float F = 15213.0;`

$$\begin{aligned} 15213_{10} &= 11101101101101_2 \\ &= 1.1101101101101_2 \times 2^{13} \end{aligned}$$

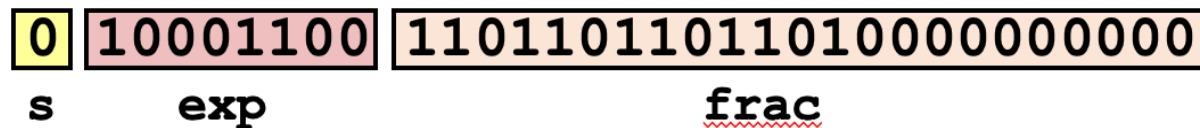
► Significand

$$\begin{aligned} M &= 1.\underline{1101101101101}_2 \\ \text{frac} &= \underline{1101101101101}0000000000_2 \end{aligned}$$

► Exponent

$$\begin{aligned} E &= 13 \\ \text{Bias} &= 127 \\ \text{Exp} &= 140 = 10001100_2 \end{aligned}$$

► Result:



Denormalized Values

- ▶ Condition: **exp** = 000...0

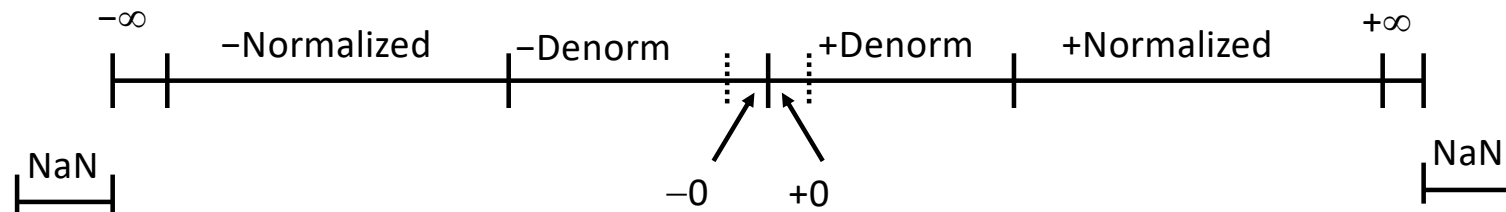
$$v = (-1)^s M 2^E$$
$$E = 1 - \text{Bias}$$

- ▶ Implicit "1." before fraction now becomes "0." (not normalized to (1.0,2.0])
- ▶ Significand coded with implied leading 0: **M** = 0.xxx...x₂
 - xxx...x: bits of **frac**
- ▶ Cases
 - **exp** = 000...0, **frac** = 000...0
 - Represents zero value
 - Note distinct values: +0 and -0 (why?)
 - **exp** = 000...0, **frac** ≠ 000...0
 - Numbers closest to 0.0

Special Values

- ▶ Condition: **exp** = **111...1**
- ▶ Case: **exp** = **111...1**, **frac** = **000...0**
 - Represents value ∞ (infinity)
 - Operation that overflows
 - Both positive and negative
 - E.g., $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$
- ▶ Case: **exp** = **111...1**, **frac** \neq **000...0**
 - Not-a-Number (NaN)
 - Represents case when no numeric value can be determined
 - E.g., $\text{sqrt}(-1)$, $\infty - \infty$, $\infty \times 0$

Visualization: Floating Point Encodings



Lab1 will be out this week

- ▶ You will be notified via email when Lab1 is out
- ▶ Our TA be helping you with your Labs

Our TA



Duy Kha Dinh

khadinh@g.skku.edu

[Personal Page](#) [CV](#)

Kha Dinh is currently an integrated MS-PhD student at Sungkyunkwan University, South Korea. He graduated from Hochiminh University of Technology, Vietnam in 2018, majored in Computer Science. His main research interest is designing secure systems.

Lecture Discussion

03 | 3주차

시작일: 3월 23일 오전 00:01

공개

+ 차시 추가

1차시

+ 페이지 추가

1페이지

Lecture 0x3 Discussion

DISCUSSION | 1점 | 0 | 마감일: 4월 5일 오후 11:59

공개

+ 가져오기

동영상 PDF 파일 소셜미디어 웹 링크 텍스트

과제 퀴즈 토론

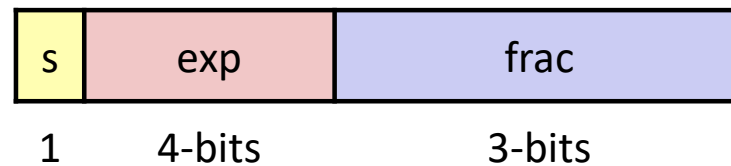
Coming up...

- ▶ Machine-level Representation of Programs
- ▶ Read CH3 of Textbook

Today: Floating Point

- ▶ Background: Fractional binary numbers
- ▶ IEEE floating point standard: Definition
- ▶ Example and properties
- ▶ Rounding, addition, multiplication
- ▶ Floating point in C
- ▶ Summary

Tiny Floating Point Example



- ▶ 8-bit Floating Point Representation
 - the sign bit is in the most significant bit
 - the next four bits are the exponent, 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

Dynamic Range (Positive Only)

$$v = (-1)^s M 2^E$$

n: $E = \text{Exp} - \text{Bias}$

d: $E = 1 - \text{Bias}$

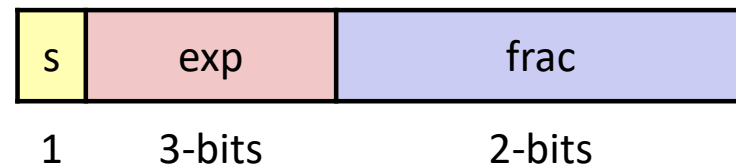
closest to zero

	s	exp	frac	E	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	$1/8 * 1/64 = 1/512$	
	0	0000	010	-6	$2/8 * 1/64 = 2/512$	
	...					
	0	0000	110	-6	$6/8 * 1/64 = 6/512$	
	0	0000	111	-6	$7/8 * 1/64 = 7/512$	
Normalized numbers	0	0001	000	-6	$8/8 * 1/64 = 8/512$	largest denorm
	0	0001	001	-6	$9/8 * 1/64 = 9/512$	smallest norm
	...					
	0	0110	110	-1	$14/8 * 1/2 = 14/16$	
	0	0110	111	-1	$15/8 * 1/2 = 15/16$	closest to 1 below
	0	0111	000	0	$8/8 * 1 = 1$	
	0	0111	001	0	$9/8 * 1 = 9/8$	
	0	0111	010	0	$10/8 * 1 = 10/8$	closest to 1 above
	...					
	0	1110	110	7	$14/8 * 128 = 224$	
	0	1110	111	7	$15/8 * 128 = 240$	largest norm
	0	1111	000	n/a	inf	

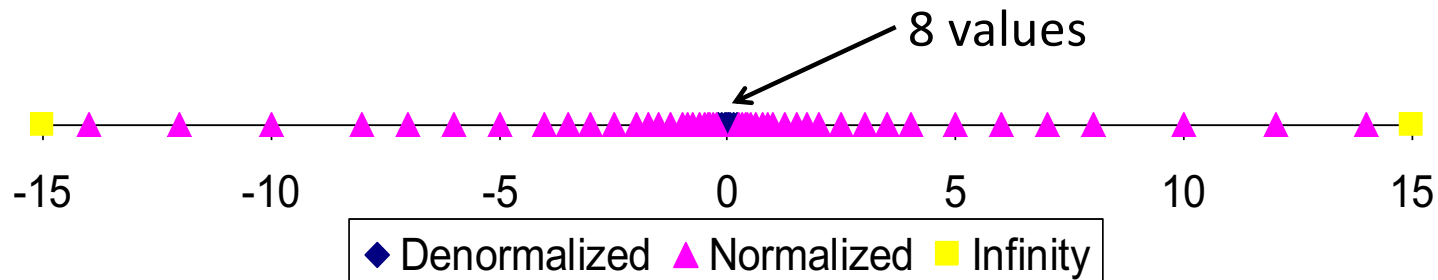
Distribution of Values

- ▶ 6-bit IEEE-like format

- $e = 3$ exponent bits
- $f = 2$ fraction bits
- Bias is $2^{3-1}-1 = 3$



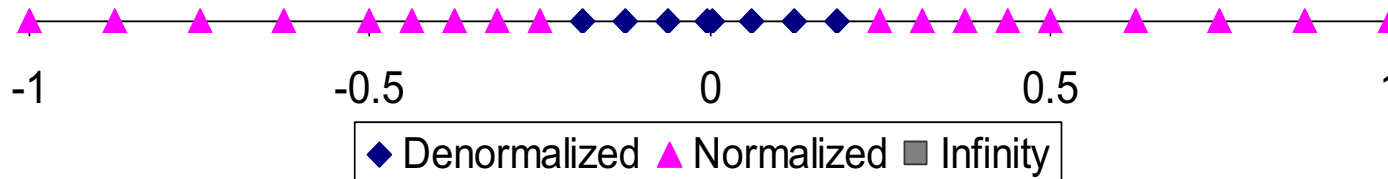
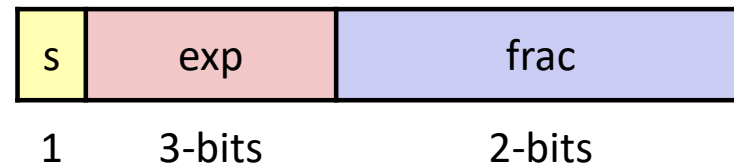
- ▶ Notice how the distribution gets denser toward zero.



Distribution of Values (close-up view)

- ▶ 6-bit IEEE-like format

- $e = 3$ exponent bits
- $f = 2$ fraction bits
- Bias is 3



Special Properties of the IEEE Encoding

- ▶ 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
 - What should comparison yield?
 - Otherwise OK
 - Denorm vs. normalized
 - Normalized vs. infinity

Today: Floating Point

- ▶ Background: Fractional binary numbers
- ▶ IEEE floating point standard: Definition
- ▶ Example and properties
- ▶ Rounding, addition, multiplication
- ▶ Floating point in C
- ▶ Summary

Floating Point Operations: Basic Idea

- ▶ $x +_f y = \text{Round}(x + y)$
- ▶ $x \times_f y = \text{Round}(x \times y)$
- ▶ Basic idea
 - First **compute exact result**
 - Make it fit into desired precision
 - Possibly overflow if exponent too large
 - Possibly **round to fit into `frac`**

Rounding

- ▶ Rounding Modes (illustrate with \$ rounding)

▶	\$1.40	\$1.60	\$1.50	\$2.50	-\$1.50
• Towards zero	\$1	\$1	\$1	\$2	-\$1
• Round down ($-\infty$)	\$1	\$1	\$1	\$2	-\$2
• Round up ($+\infty$)	\$2	\$2	\$2	\$3	-\$1
• Nearest Even (default)	\$1	\$2	\$2	\$2	-\$2

Closer Look at Round-To-Even

- ▶ Default Rounding Mode

- Hard to get any other kind without dropping into assembly
- All others are statistically biased
 - Sum of set of positive numbers will consistently be over- or under- estimated

- ▶ Applying to Other Decimal Places / Bit Positions

- When exactly halfway between two possible values
 - Round so that least significant digit is even
- E.g., round to nearest hundredth

7.89499997.89	(Less than half way)
7.89500017.90	(Greater than half way)
7.8950000	7.90 (Half way—round up)
7.8850000	7.88 (Half way—round down)

Rounding Binary Numbers

- ▶ Binary Fractional Numbers
 - “Even” when least significant bit is 0
 - “Half way” when bits to right of rounding position = $100..._2$

- ▶ Examples

- Round to nearest $1/4$ (2 bits right of binary point)

Value	Binary	Rounded	Action	Rounded Value
$2 \frac{3}{32}$	10.00011_2	10.00_2	($<1/2$ —down)	2
$2 \frac{3}{16}$	10.00110_2	10.01_2	($>1/2$ —up)	$2 \frac{1}{4}$
$2 \frac{7}{8}$	10.11100_2	11.00_2	($=1/2$ —up)	3
$2 \frac{5}{8}$	10.10100_2	10.10_2	($=1/2$ —down)	$2 \frac{1}{2}$

FP Multiplication

- ▶ $(-1)^{s1} \mathbf{M1} 2^{E1} \times (-1)^{s2} \mathbf{M2} 2^{E2}$
- ▶ Exact Result: $(-1)^s \mathbf{M} 2^E$
 - Sign s : $s1 \wedge s2$
 - Significand \mathbf{M} : $\mathbf{M1} \times \mathbf{M2}$
 - Exponent E : $E1 + E2$
- ▶ Fixing
 - If $\mathbf{M} \geq 2$, shift \mathbf{M} right, increment E
 - If E out of range, overflow
 - Round \mathbf{M} to fit **frac** precision
- ▶ Implementation
 - Biggest chore is multiplying significands

Floating Point Addition

▶ $(-1)^{s1} M1 2^{E1} + (-1)^{s2} M2 2^{E2}$

• Assume $E1 > E2$

▶ Exact Result: $(-1)^s M 2^E$

• Sign s , significand M :

• Result of signed align & add

• Exponent E : $E1$

▶ Fixing

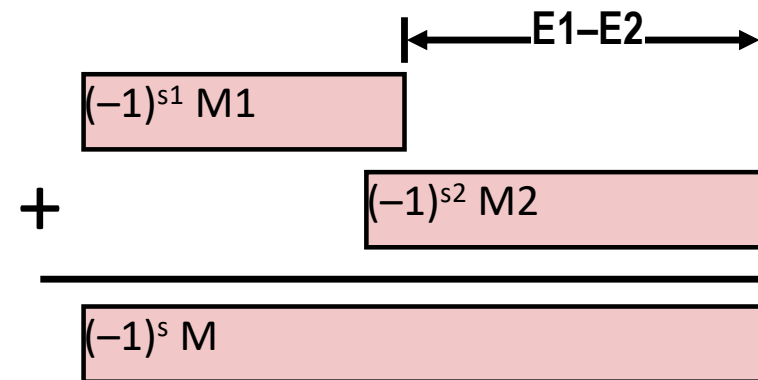
• If $M \geq 2$, shift M right, increment E

• if $M < 1$, shift M left k positions, decrement E by k

• Overflow if E out of range

• Round M to fit **frac** precision

Get binary points lined up



Mathematical Properties of FP Add

- ▶ Compare to those of Abelian Group
 - Closed under addition? **Yes**
 - But may generate infinity or NaN
 - Commutative? **Yes**
 - Associative? **No**
 - Overflow and inexactness of rounding
 - $(3.14+1e10)-1e10 = 0$, $3.14+(1e10-1e10) = 3.14$
 - 0 is additive identity?
 - Every element has additive inverse? **Yes**
 - Yes, except for infinities & NaNs **Almost**
- ▶ Monotonicity
 - $a \geq b \Rightarrow a+c \geq b+c$? **Almost**
 - Except for infinities & NaNs

Mathematical Properties of FP Mult

- ▶ Compare to Commutative Ring
 - Closed under multiplication? **Yes**
 - But may generate infinity or NaN
 - Multiplication Commutative? **Yes**
 - Multiplication is Associative? **No**
 - Possibility of overflow, inexactness of rounding
 - Ex: $(1e20 * 1e20) * 1e-20 = \text{inf}$, $1e20 * (1e20 * 1e-20) = 1e20$
 - 1 is multiplicative identity? **Yes**
 - Multiplication distributes over addition? **No**
 - Possibility of overflow, inexactness of rounding
 - $1e20 * (1e20 - 1e20) = 0.0$, $1e20 * 1e20 - 1e20 * 1e20 = \text{NaN}$
- ▶ Monotonicity
 - $a \geq b \ \& \ c \geq 0 \Rightarrow a * c \geq b * c$? **Almost**
 - Except for infinities & NaNs

Today: Floating Point

- ▶ Background: Fractional binary numbers
- ▶ IEEE floating point standard: Definition
- ▶ Example and properties
- ▶ Rounding, addition, multiplication
- ▶ Floating point in C
- ▶ Summary

Floating Point in C

- ▶ C Guarantees Two Levels

- **float** single precision
- **double** double precision

- ▶ Conversions/Casting

- Casting between **int**, **float**, and **double** changes bit representation
- **double/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 ≤ 53 bit word size
- **int** → **float**
 - Will round according to rounding mode

Floating Point Puzzles

- ▶ For each of the following C expressions, either:
 - Argue that it is true for all argument values
 - Explain why not true

```
int x = ...;  
float f = ...;  
double d = ...;
```

Assume neither
d nor f is NaN

- `x == (int)(float) x`
- `x == (int)(double) x`
- `f == (float)(double) f`
- `d == (double)(float) d`
- `f == -(-f);`
- `2/3 == 2/3.0`
- `d < 0.0` \Rightarrow `((d*2) < 0.0)`
- `d > f` \Rightarrow `-f > -d`
- `d * d >= 0.0`
- `(d+f)-d == f`

Summary

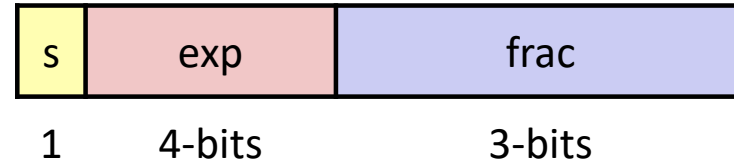
- ▶ IEEE Floating Point has clear mathematical properties
- ▶ Represents numbers of form $M \times 2^E$
- ▶ One can reason about operations independent of implementation
 - As if computed with perfect precision and then rounded
- ▶ Not the same as real arithmetic
 - Violates associativity/distributivity
 - Makes life difficult for compilers & serious numerical applications programmers

Additional Slides

Creating Floating Point Number

▸ Steps

- Normalize to have leading 1
- Round to fit within fraction
- Postnormalize to deal with effects of rounding



▸ Case Study

- Convert 8-bit unsigned numbers to tiny floating point format

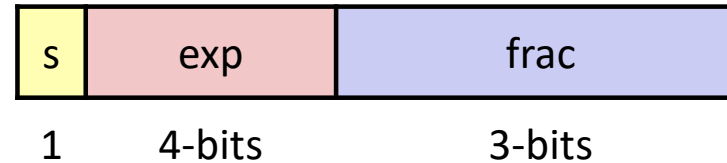
Example Numbers

128	10000000
15	00001101
33	00010001
35	00010011
138	10001010
63	00111111

Normalize

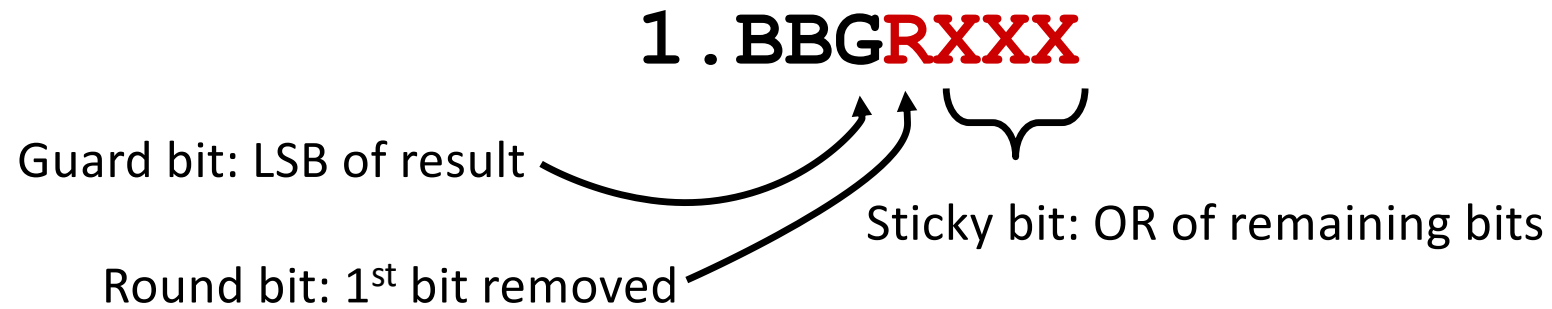
► Requirement

- Set binary point so that numbers of form 1.xxxxx
- Adjust all to have leading one
 - Decrement exponent as shift left



Value	Binary	Fraction	Exponent
128	10000000	1.0000000	7
15	00001101	1.1010000	3
17	00010001	1.0001000	4
19	00010011	1.0011000	4
138	10001010	1.0001010	7
63	00111111	1.1111100	5

Rounding



- ▶ Round up conditions
 - Round = 1, Sticky = 1 → > 0.5
 - Guard = 1, Round = 1, Sticky = 0 → Round to even

Value	Fraction	GRS	Incr?	Rounded
128	1.000 0000	000	N	1.000
15	1.101 0000	100	N	1.101
17	1.000 1000	010	N	1.000
19	1.001 1000	110	Y	1.010
138	1.000 1010	011	Y	1.001
63	1.111 1100	111	Y	10.000

Postnormalize

- ▶ Issue
 - Rounding may have caused overflow
 - Handle by shifting right once & incrementing exponent

Value	Rounded	Exp	Adjusted	Result
128	1.000	7		128
15	1.101	3		15
17	1.000	4		16
19	1.010	4		20
138	1.001	7		134
63	10.000	5	1.000/6	64

Interesting Numbers

{single, double}

Description	exp	frac	Numeric Value
▶ Zero	00...00	00...00	0.0
▶ Smallest Pos. Denorm. <ul style="list-style-type: none"> • Single $\approx 1.4 \times 10^{-45}$ • Double $\approx 4.9 \times 10^{-324}$ 	00...00	00...01	$2^{-\{23,52\}} \times 2^{-\{126,1022\}}$
▶ Largest Denormalized <ul style="list-style-type: none"> • Single $\approx 1.18 \times 10^{-38}$ • Double $\approx 2.2 \times 10^{-308}$ 	00...00	11...11	$(1.0 - \epsilon) \times 2^{-\{126,1022\}}$
▶ Smallest Pos. Normalized <ul style="list-style-type: none"> • Just larger than largest denormalized 	00...01	00...00	$1.0 \times 2^{-\{126,1022\}}$
▶ One	01...11	00...00	1.0
▶ Largest Normalized <ul style="list-style-type: none"> • Single $\approx 3.4 \times 10^{38}$ • Double $\approx 1.8 \times 10^{308}$ 	11...10	11...11	$(2.0 - \epsilon) \times 2^{\{127,1023\}}$