Floating Point

These slides adapted from materials provided by the textbook authors

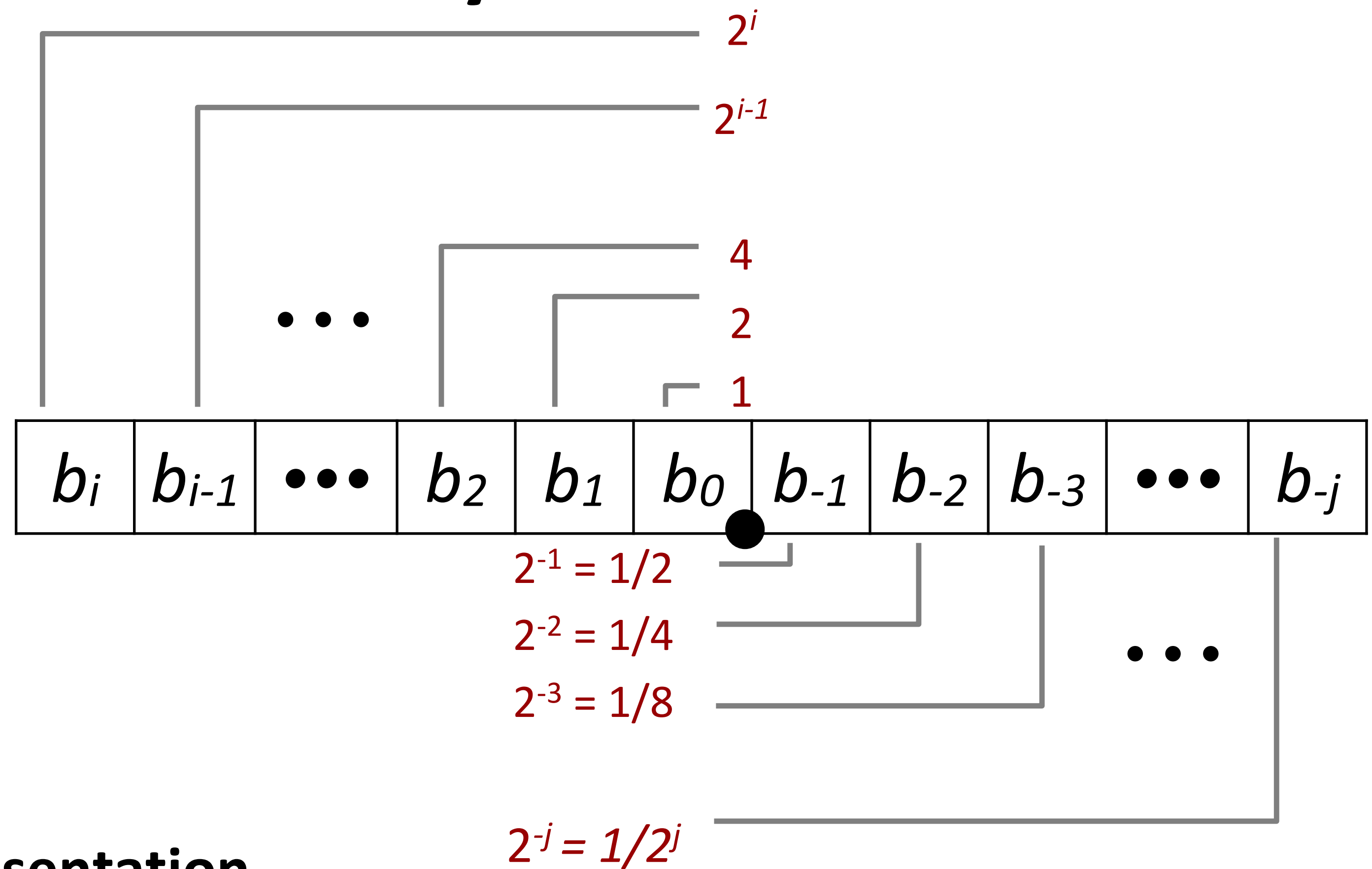
Floating Point

- **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

- What is 1011.101_2?

Fractional Binary Numbers: Examples

- **Value** **Representation**

5 3/4	`101.11`$_2$
2 7/8	`10.111`$_2$
1 7/16	`1.0111`$_2$

- **Observations**
  - Divide by 2 by shifting right (unsigned)
  - Multiply by 2 by shifting left
  - Numbers of form $0.111111\ldots_2$ are just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - Use notation $1.0 - \varepsilon$

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 — $0.0101010101[\mathit{01}]\ldots_2$
    - 1/5 — $0.001100110011[\mathit{0011}]\ldots_2$
    - 1/10 — $0.0001100110011[\mathit{0011}]\ldots_2$

- **Limitation #2**
  - Just one setting of binary point within the w bits
    - Limited range of numbers (very small values? very large?)

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 Design Challenges

- Only so many bits. Need to be efficient
- Need to handle “special” numbers like Infinity
- Numbers near zero are “special” because 1/x is common
- Properly rounding numbers is important in many algorithms
- What to represent large numbers (1.23×10^{15}) and small (1.23×10^{-15})

Floating Point Representation

- **Numerical Form:**

  $$(-1)^s \; M \; 2^E$$

  - 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)

Exponents, single precision float

- **What is the range of exp?**

- **What is the range of E?**

"Normalized" Values

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

- **When: exp ≠ 000…0 and exp ≠ 111…1**

- **Exponent coded as a biased value: E = Exp – 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)

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

“Normalized” Values

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

- **Significand coded with implied leading 1:**
  $M = 1.xxx\ldots x_2$
  - xxx…x: bits of frac field
  - Minimum when frac=000…0 (M = 1.0)
  - Maximum when frac=111…1 (M = 2.0 – ε)
  - Get extra leading bit for “free”

$$v = (-1)^s * (1.[frac]) * 2^{(Exp\text{-}Bias)}$$

Normalized Encoding Example

- **Value:** `float F = 15213.0;`
  - $15213_{10} = 11101101101101_2$
    $= 1.1101101101101_2 \times 2^{13}$

- **Significand**

  M = `1.1101 1011 0110 1`$_2$

  `frac` = `1101 1011 0110 1000 0000 0000`$_2$

- **Exponent**

  E = 13

  $Bias$ = 127

  Exp = 140 = `10001100`$_2$

- **Result:**

s	exp	frac
0	10001100	11011011011010000000000

Denormalized Values

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

- **Decoded *differently* than normalized values!**
- **Condition: exp = 000…0**

- **Exponent value: E = 1 – Bias (instead of E = 0 – Bias)**
- **Significand coded with implied leading 0: M = 0.xxx…x2**
  - xxx…x: bits of frac

$$v = (-1)^s * (0.[\text{frac}]) * 2^{(1\text{-}Bias)}$$

Denormalized Values

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

- **Cases**
  - exp = 000…0, frac = 000…0
    - Represents zero value, without **denorm we are unable to represent zero**
    - Note distinct values: +0 and –0 (*why?*)
  - exp = 000…0, frac ≠ 000…0
    - Numbers closest to 0.0
    - Equispaced

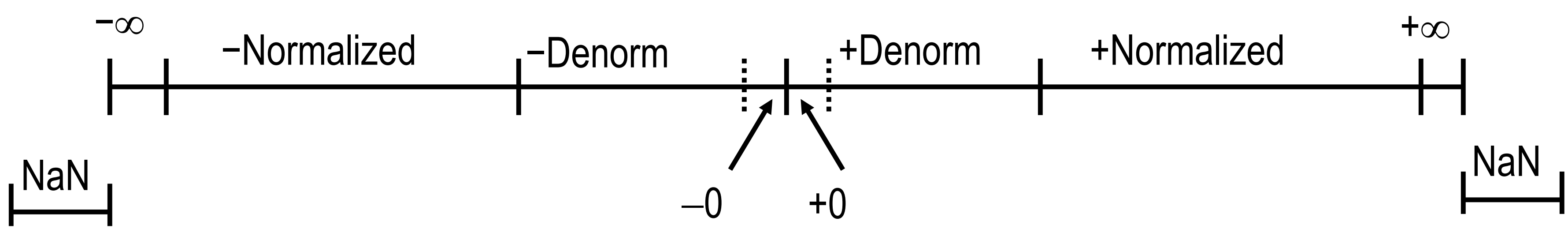
Special Values

- **Condition: exp = 111…1 , Exponent all ones**

- **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 ≠ 000…0**
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), $\infty - \infty$, $\infty \times 0$

Visualization: Floating Point Encodings

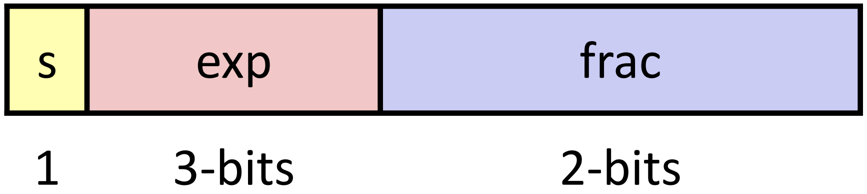


Floating Point

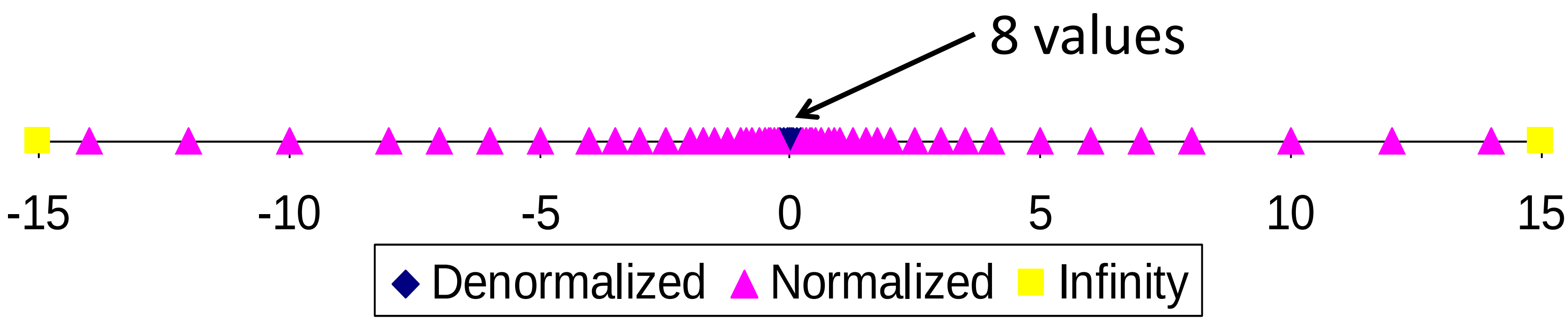
- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- **Example and properties**
- Rounding, addition, multiplication
- Floating point in C
- Summary

Distribution of Values

- **6-bit IEEE-like format**
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is $2^{3-1}-1 = 3$



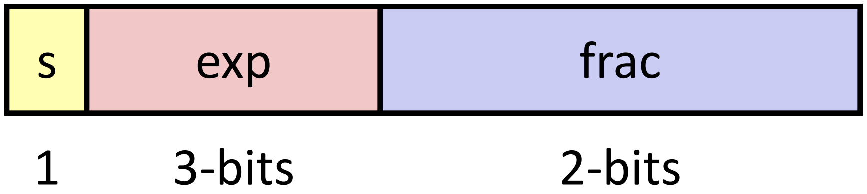
- **Notice how the distribution gets denser toward zero.**

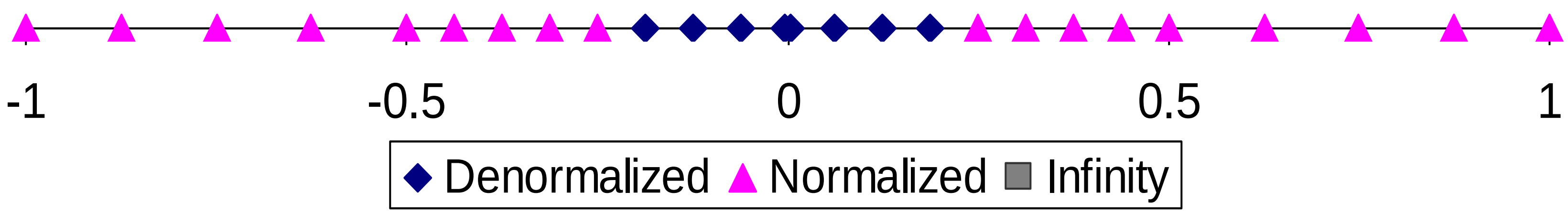


$$v = (-1)^s \, M \, 2^E$$

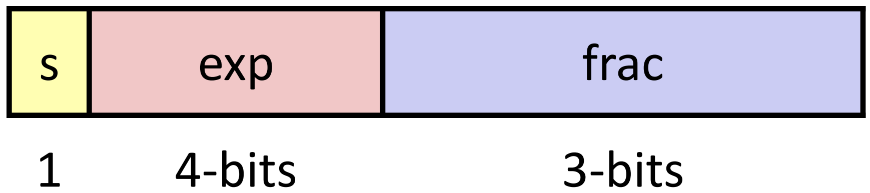
Distribution of Values (close-up view)

- **6-bit IEEE-like format**
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is 3





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 = Exp – Bias

d: E = 1 – Bias

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

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

- $x +_f y = \mathrm{Round}(x + y)$

- $x \times_f y = \mathrm{Round}(x \times y)$

- **Basic idea**
  - First compute exact result
  - Make it fit into desired precision
    - Possibly overflow if exponent too large
    - Possibly round to fit into `frac`

Closer Look at Round-To-Even

- **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
    Expected behavior except when halfway

7.8949999	7.89	(Less than half way)
7.8950001	7.90	(Greater than half way)
7.8950000	7.90	(Half way - round up , even 0)
7.8850000	7.88	(Half way - round down , even 8)

Rounding Binary Numbers

- **Binary Fractional Numbers**
  - "Even" when least significant bit is `0`
  - "Half way" when bits to right of rounding position = $100\ldots_2$

Rounding Binary Numbers

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

Value	Binary	Rounded	Action	Rounded
2 3/32	$10.00\underline{011}_2$		10.00_2	(<1/2—down)
2 3/16	$10.00\underline{110}_2$		10.01_2	(>1/2—up)
2 7/8	$10.11\underline{100}_2$		11.00_2	(1/2—up)
2 5/8	$10.10\underline{100}_2$		10.10_2	(1/2—down)

Rounding

1.BBGRXXX

Guard bit: LSB of result

Sticky bit: OR of remaining bits

Round bit: 1st bit removed

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

Value	*Fraction*	*GRS*	*Incr?*	*Rounded*
128	1.0000000	000	N	1.000
15	1.1010000	100	N	1.101
17	1.0001000	010	N	1.000
19	1.0011000	110	Y	1.010
138	1.0001010	011	Y	1.001
63	1.1111100	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

FP Multiplication

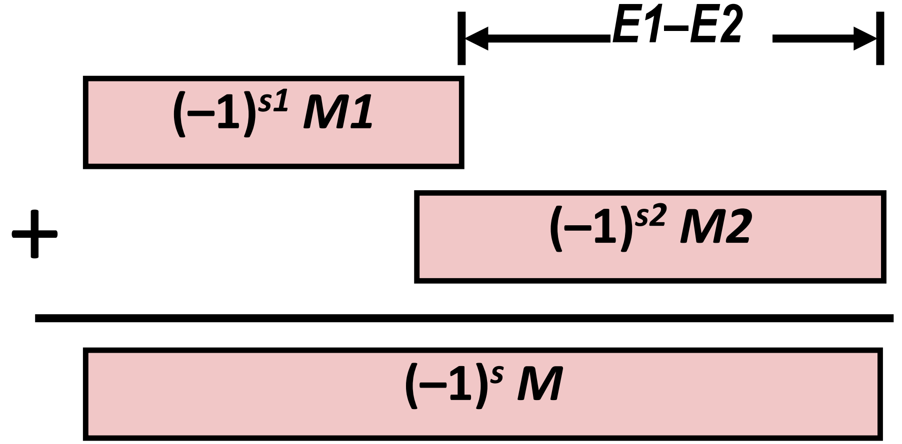
- $(-1)^{s1}\, M1\ \ 2^{E1}\ \ \times\ \ (-1)^{s2}\, M2\ \ 2^{E2}$
- **Exact Result:** $(-1)^{s}\, M\ \ 2^{E}$
  - Sign *s*: $s1 \wedge s2$
  - Significand *M*: $M1 \times M2$
  - Exponent *E*: $E1 + E2$

- **Fixing**
  - If $M \geq 2$, shift *M* right, increment *E*
  - If *E* out of range, overflow
  - Round *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

- **Commutative? , *Yes***
- **Associative? , *No***
  - Overflow and inexactness of rounding
  - `(3.14+1e20)-1e20 = 0,`
    `3.14+(1e20-1e20) = 3.14`
- **0 is additive identity? , *Yes***
- **Every element has additive inverse? , *Almost***
  - Yes, except for infinities & NaNs
- **Monotonicity**
  $a \geq b \Rightarrow a+c \geq b+c$? , *Almost*
  - Except for infinities & NaNs

Mathematical Properties of FP Mult

- **Multiplication Commutative? , *Yes***
- **Multiplication is Associative? , *No***
  - Possibility of overflow, inexactness of rounding
  - Ex: `(1e20*1e20)*1e-20= 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 = NaN`
- **Monotonicity**
  $a \geq b$ **&** $c \geq 0 \Rightarrow a * c \geq b * c$**? , *Almost***
  - Except for infinities & NaNs

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

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

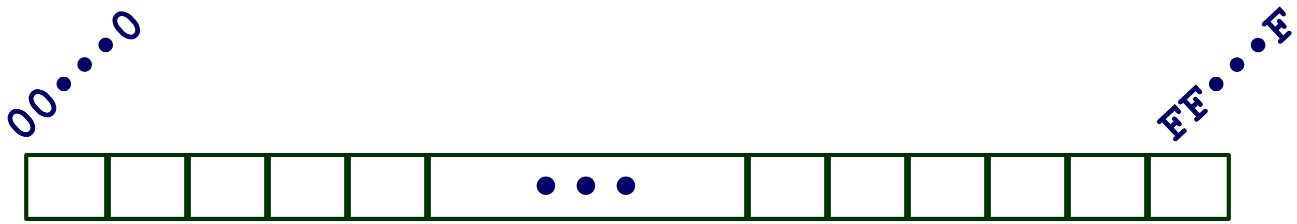
Strings and Memory Representations

Reading: CS:APP Chapter 2.1

Strings and Memory Representations

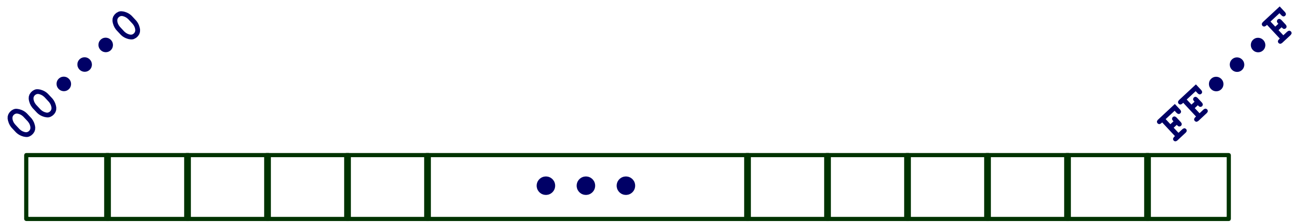
- From Bytes to Words
- Byte ordering
- Pointer representation
- Character and string representation
- Summary

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

Byte-Oriented Memory Organization



- **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

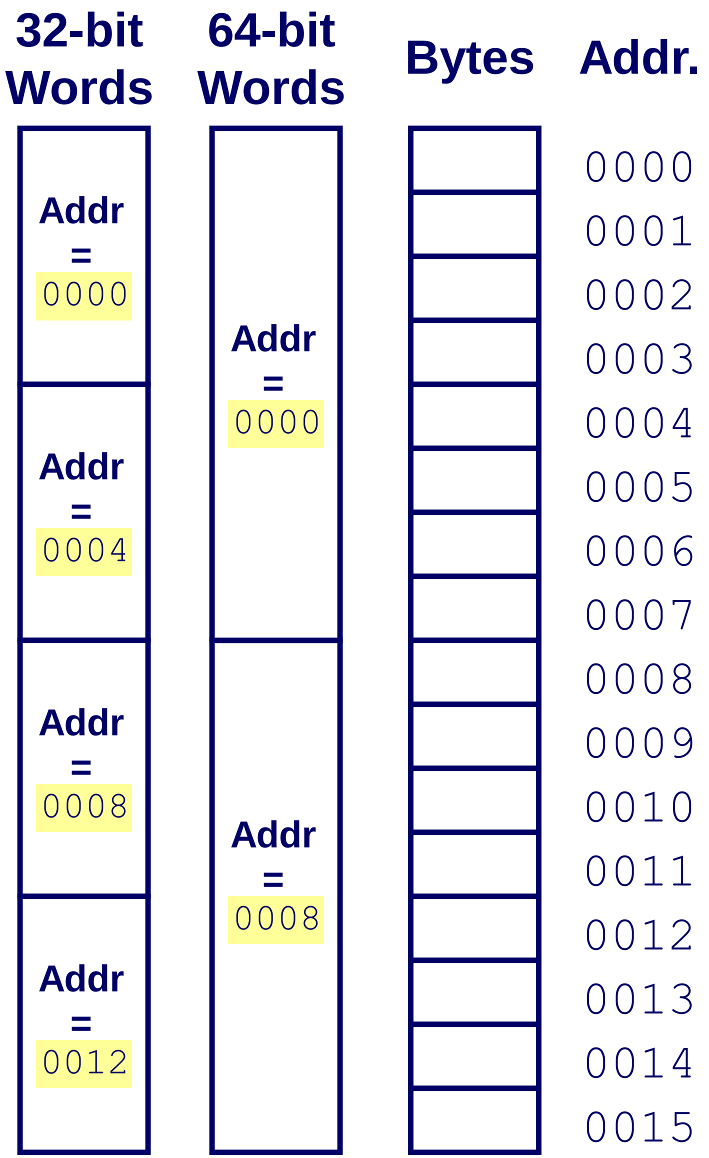
- **Any given computer has a "Word Size"**
  - Nominal size of integer-valued data
    - and of addresses
  - Until recently, most machines used 32 bits (4 bytes) as word size
    - Limits addresses to 4GB (2^{32} bytes)

Machine Words

- **Any given computer has a "Word Size"**
  - Except for embedded systems, machines have 64-bit word size
    - Potentially, could have 18 EB (exabytes) of addressable memory
    - That's 18.4 X 10^{18}
    - Really, only 47 bits of the word specify addresses on x64
      14.1 X 10^{13} bytes , Still much more RAM than your machine has
  - 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 Y has 4-byte value of 0x01234567
  - Address given by &Y is 0x100

Big Endian

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

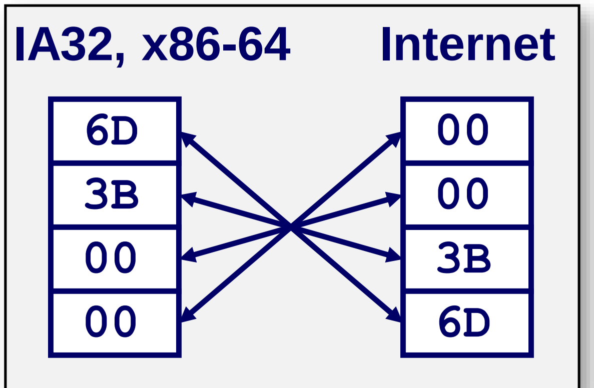
Little Endian

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

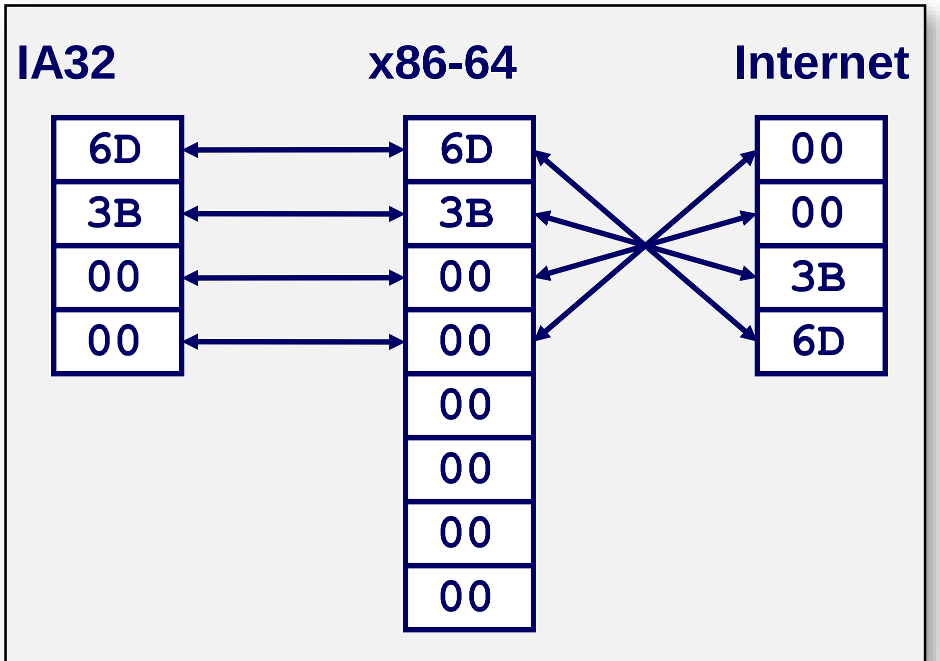
Representing Integers

Decimal:	15213
Binary:	0011 1011 0110 1101
Hex:	3 B 6 D

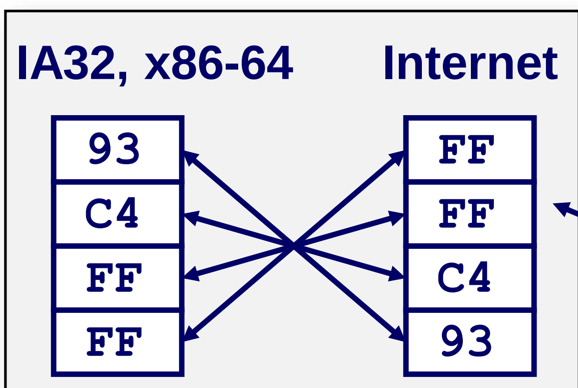
`int A = 15213;`



`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 Characters

- Representing letters and words by numbers: morse code
- Later TTY systems converted typewriter into morse / baudot other codes
- Led to ASCII (American Standard Code for Information Interchange) in ~1960
- 7 (then 8) bit code for letters, numbers, “actions”
- Expanded to Unicode in 90’s and UTF-8 in ‘00’s

ف ஔ ఈ ಋ

USASCII code chart

b_7					0	0	0	0	1	1	1	1
b_6					0	0	1	1	0	0	1	1
b_5					0	1	0	1	0	1	0	1
b_4	b_3	b_2	b_1	Column / Row	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

ASCII collating order

- Sequences 0..9, a..z and A..Z are in order

- Checking if ‘d’ is a digit:

  (d >= ‘0’) && (d <= ‘9’)

- Checking if ‘l’ is a letter:

  ((l >= ‘a’) && (l <= ‘z’)) || ((l >= ‘A’) && (l <= ‘Z’))

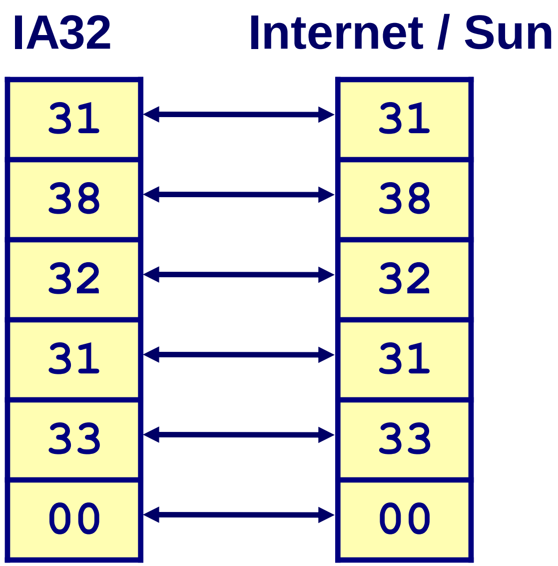
- Change letter ‘A’..’Z’ to lower case:

  (l – ‘A’) + ‘a’

Representing Strings

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

- **Strings in C**
  - Represented by array of ‘char’
  - Char ‘x’ enclosed in single quotes, string “enclosed in double”
  - ‘abcd’ is not same as “abcd”
    - ‘abcd’ is a 4–byte integer, not a string
    - “abcd” is 5 byte string with null terminator
  - Each character encoded in ASCII format
  - String should be null-terminated
    - Final character = 0



- **Compatibility**
  - Byte ordering not an issue for strings

Demo Program: 03_stringSize.c

Bonus Material: Copying Strings / Memory

String and Memory Functions in C

http://www.cplusplus.com/reference/cstring/

String

- strcpy(dst, src) – copy
- strlen(str) – length
- strcat(dest, src) – append
- strcmp(src1, src2)–compare

- strnlen(str, max)
- strncpy(dst, src, dstsize)
- strncat(dest, src, dstsize)
- strncmp(src1, src2, max)

Memory

- memcpy(dst, src, len)
- memmove(dst, src, len)
- memset(dst, value, len)
- memcmp(dst, src, len)

C string functions – strcpy 3 ways

- And all of them bad!

```
char * strcpy(char *dst, char *src){
   int i;
   for( i = 0; src[i] != '\0'; i++) {
     dst[i] = src[i];
   }
   dst[i] = 0;
   return dst;
}
```

```
char * strcpy(char *dst, char *src){
   char *orig = dst;
   for( ; *src ; dst++, src++ ) {
     *dst = *src;
   }
   *dst = 0;
   return orig;
}
```

```
char * strcpy(char *dst, char *s
   char *orig = dst;
   while(*src) {
     *dst++ = *src++;
   }
   *dst = 0;
   return orig;
}
```

C string functions – strncpy

- Most compilers will warn about strcpy, strlen, etc

```
char * strcpy(char *dst, char *src, size_t len){
   int i;

   for( i = 0; i < len && src[i] != '\0'; i++) {
     dst[i] = src[i];
   }
   dst[i] = 0;
   return dst;
}
```

- You'll be directed to the 'n' versions

C memory functions – memcpy

- **The mem* functions don't look for null terminator**

```
void* memcpy(void *dst, void *src, size_t len){
   int i;
   char *cdst = (char *) dst;
   char *csrc = (char *) src;
   for( i = 0; i < len ; i++) {
     cdst[i] = csrc[i];
   }
   return dst;
}
```

- **Use provided str* and mem* functions rather than writing your own.**
  - Much, much faster code
  - They already made it correct

C strcmp and memcmp

- **strncmp(a,b) and memcmp(a,b,len) return -1, 0 or 1**
  - -1 a < b
  - 0 0 == b
  - 1 a > b
- **Leads to weird looking programming idiom**

```
if ( ! strncmp(a,b,n)) {
   /* a is equal to b */
} else {
   /* a != b */
}
```

- **Better to write this like..**

```
if ( strncmp(a,b,n)== 0 ) {
   /* a is equal to b */
}
```