Introduction to Computer Systems

Lecture 4 – Float

2022 Spring, CSE3030

Sogang University

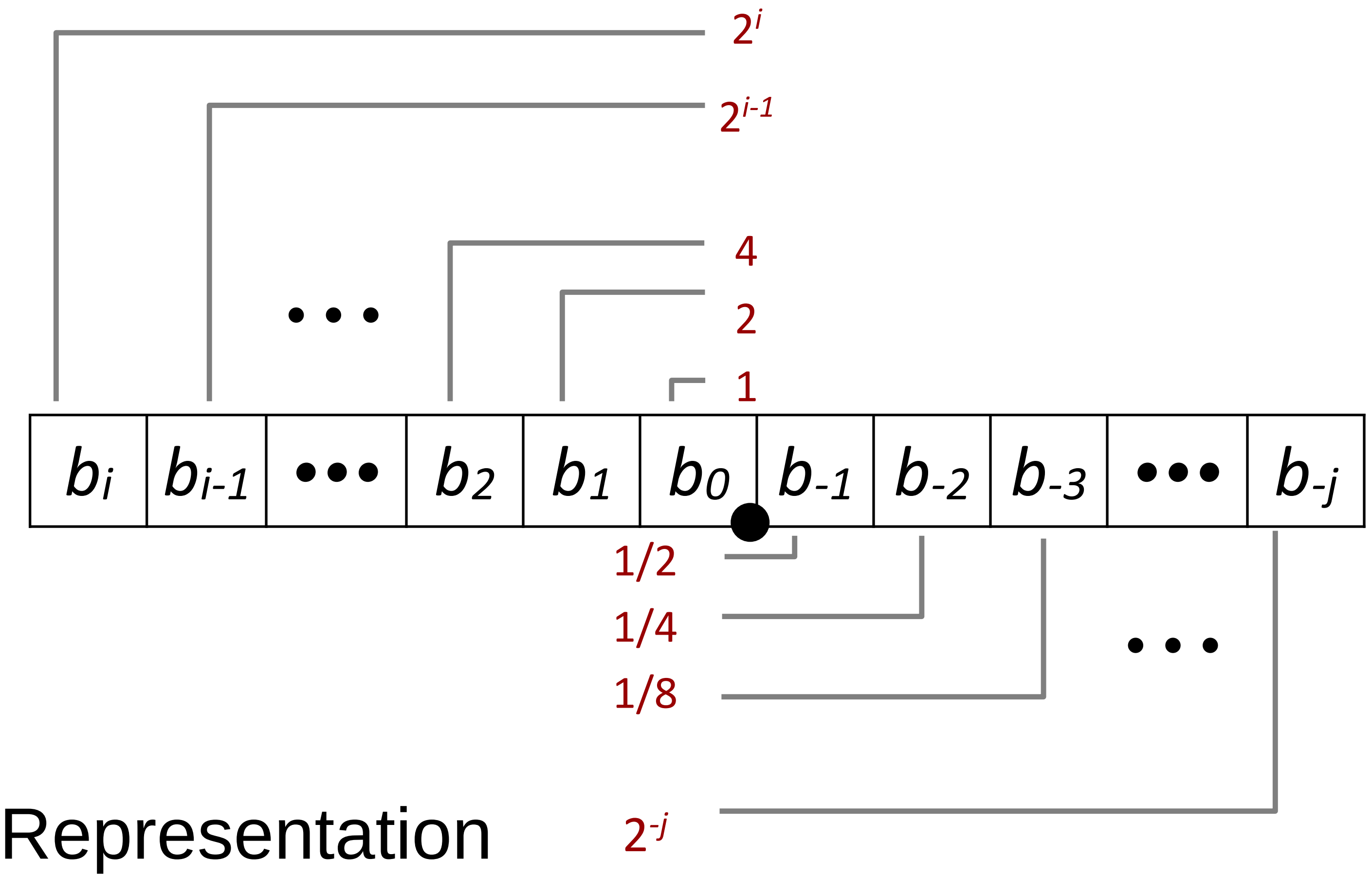
Fractional binary numbers

- What is 1011.101_2?

Integer part Decimal part

$$1011.101$$

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**

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[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?)

Limited range of numbers

- The range of fractional binary representation is less than signed integer representation.
  - FMAX < TMAX
- How can we represent different scales of real numbers?
  - Numbers could be very small or very large.
  - Sun to Earth: 149,600,000,000 m
  - Light speed: 299,792,458 m/s
  - Time to travel 1 um in light speed: 0.00000000000000333564 sec
  - 1um = 0.000001 m = 0. 00000 00000 00000 00001 000.... $_{(2)}$ m
- What representation can support arithmetic operations for a wide range of real numbers?

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

https://en.wikipedia.org/wiki/Scientific_notation

- Numerical Form:

$$(-1)^s \, \boldsymbol{M} \;\; 2^{E}$$

  - **Sign bit *s*** determines whether number is negative or positive
  - **Significand (or *mantissa*) *M*** normally a fractional value in range [1.0,2.0).
  - **Exponent *E*** weights value by power of two

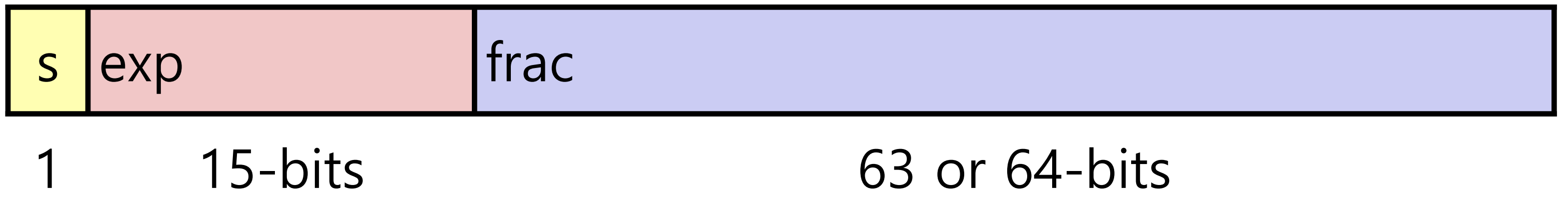
- Encoding
  - MSB s is sign bit ***s***
  - exp field encodes ***E*** (but is not equal to E)
  - frac field encodes ***M*** (but is not equal to M)

Precision options

- Single precision: 32 bits

- Double precision: 64 bits

- Extended precision: 80 bits (Intel only)



“Normalized” Values

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

- When: exp ≠ 000…0 and exp ≠ 111…1

- Exponent coded as a ***biased*** value: $\boldsymbol{E} = \boldsymbol{Exp} - \boldsymbol{Bias}$
  - *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: $\boldsymbol{M} = 1.xxx\ldots x_2$
  - 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”

Normalized Encoding Example

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

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

- Significand
  M = `1.`$\underline{1101101101101}_2$
  `frac=` $\underline{1101101101101}0000000000_2$

- Exponent
  E = 13
  $Bias$ = 127
  Exp = 140 = $\mathbf{10001100_2}$

- Result:

0	10001100	11011011011010000000000
s	**exp**	**frac**

Denormalized Values

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

- Condition: exp = 000…0

- Exponent value: $E = 1 - \text{Bias}$ (instead of $E = 0 - Bias$)
- Significand coded with implied leading 0: $M = 0.\text{xxx}\ldots\text{x}_2$
  - `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
    - Equispaced

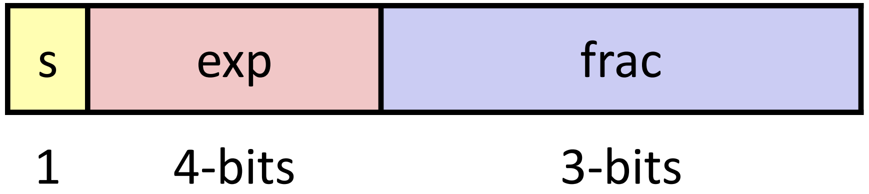
Special Values

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

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)

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

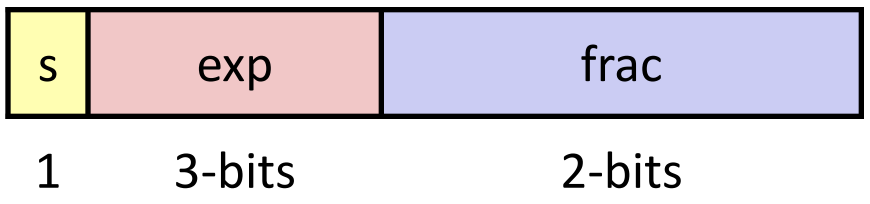
n: E = Exp – Bias

d: E = 1 – Bias

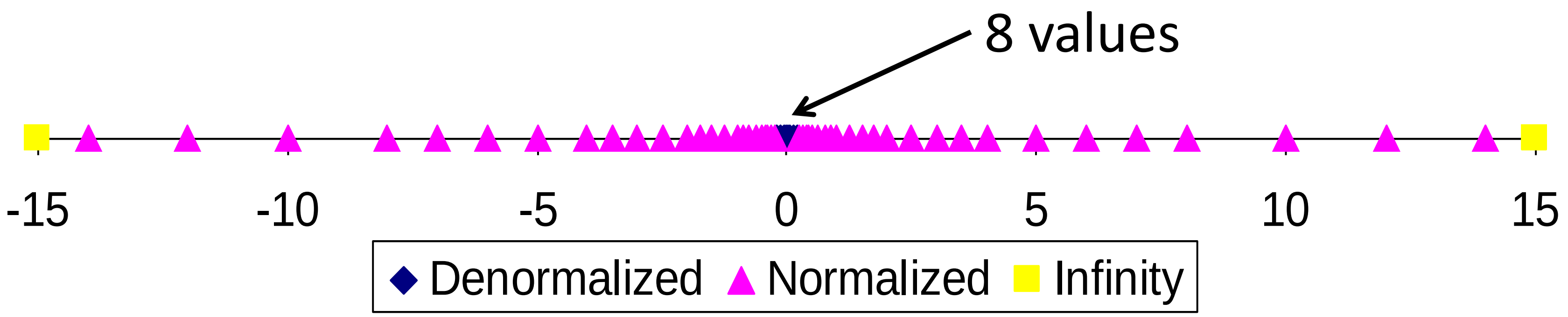
	s	exp	frac	E	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	closest to zero
	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	largest denorm
Normalized numbers	0	0001	000	-6	8/8*1/64 = 8/512	smallest norm
	0	0001	001	-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	

Distribution of Values

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

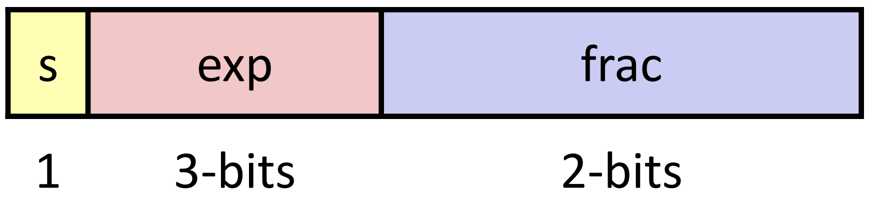


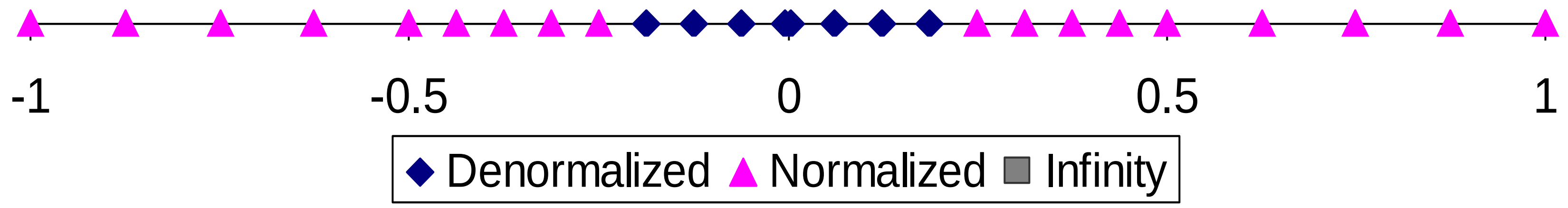
- Notice how the distribution gets denser toward zero.



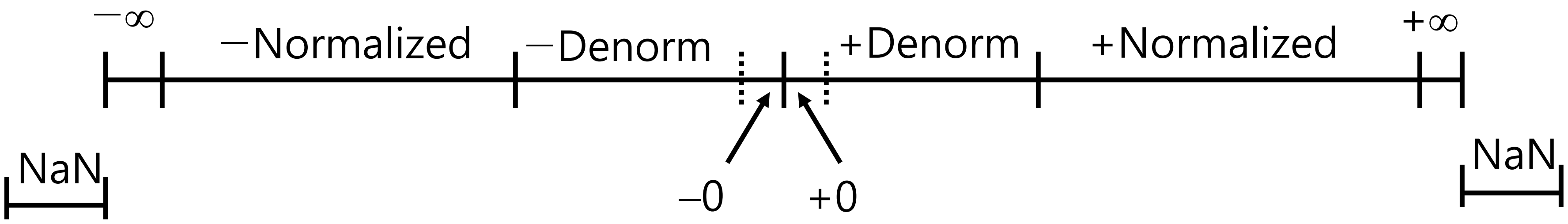
Distribution of Values (close-up view)

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





Visualization: Floating Point Encodings



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

Rounding

- For single precision, we only have 23 bits for a frac part.

- If there is number *x* that requires more than 23 bits for a frac part, we need to find a number x' that is close to *x.*
  - *Ex)* 33554431 = 1 11111111 11111111 11111111 (2) (25 bits required for frac parts)

-

	$1.40	$1.60	$1.50	$2.50	–$1.50
Towards zero	$1	$1	$1	$2	–$1
Round down (−∞)	$1	$1	$1	$2	–$2
Round up (+∞)	$2	$2	$2	$3	–$1
Nearest Even (default)	$1	$2	$2	$2	–$2

Rounding

1.BBGRXXX

Guard bit: LSB of result

Round bit: 1st bit removed

Sticky bit: OR of remaining bits

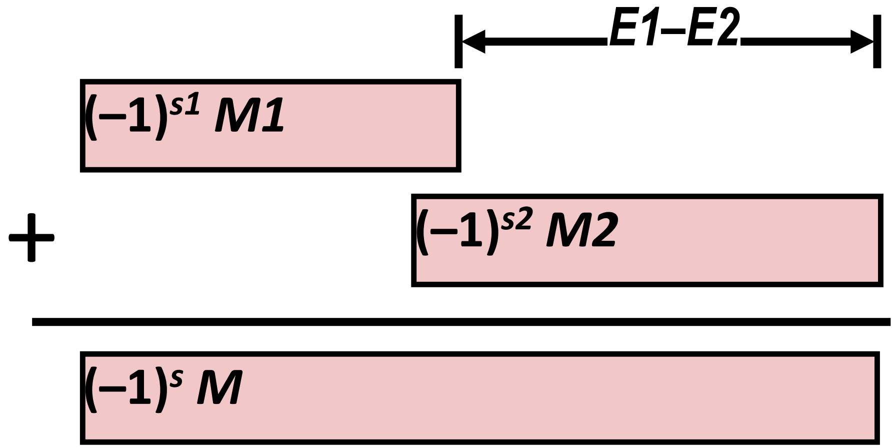
- Round up conditions
  - Round = 1, Sticky = 1 → > 0.5
  - Guard = 1, Round = 1, Sticky = 0 → Round to even

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

Floating Point Addition

- $(-1)^{s1}\ \boldsymbol{M1}\ \ 2^{E1}\ \ +\ \ (-1)^{s2}\ \boldsymbol{M2}\ \ 2^{E2}$
  - Assume $E1 > E2$

Get binary points lined up



- Exact Result: $(-1)^{s}\ \boldsymbol{M}\ \ 2^{E}$
  - 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

FP Multiplication

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

- Fixing
  - If *M* ≥ 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 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

Ariane 5 S/W Bug

- Ariane 5 flight 501 (June 4, 1996)
  - Exploded 37 seconds after liftoff
  - Lost 4 cluster mission

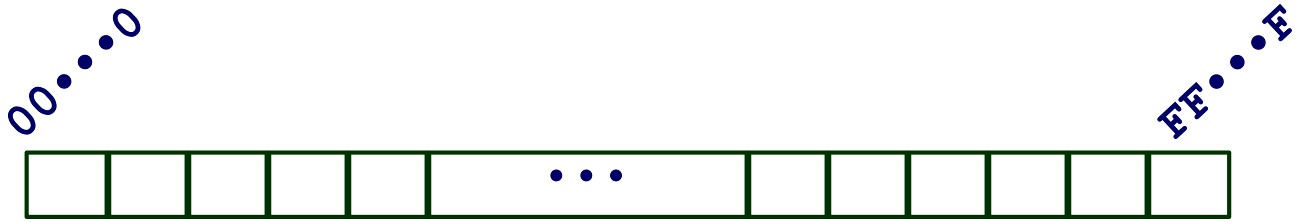
  spacecraft worth $370 million
- Why?
  - Computed horizontal velocity as floating point number (64bit)
  - Converted to 16-bit integer
    - Careful analysis of Ariane 4 trajectory proved 16-bit is enough
  - Reused a module from 10-year-old software
    - Overflowed for Ariane 5
    - No precise specification for the S/W

ON 4 JUNE 1996, THE MAIDEN FLIGHT OF the Ariane 5 launcher exploded about 37 seconds after liftoff. Scientists with experiments on board that had taken years to prepare were devastated. For many software engineering researchers, however, the disaster is a case study rich in lessons. To begin learning from this disaster, we need look no further than a report on it issued by an independent inquiry board set up by the French and European Space Agencies.

VARIED VIEWS. Here are some of the interpretations of the report that I have heard.

- *What the programmers said:* The disaster is clearly the result of a programming error. An incorrectly handled software exception resulted from a data conversion of a 64-bit floating point to a 16-bit signed integer value. The value of the floating point number that was converted was larger than what could be represented by a 16-bit integer, resulting in an operand error not anticipated by the Ada code. Better programming practice would have prevented this failure from occurring.

Byte-Oriented Memory Organization

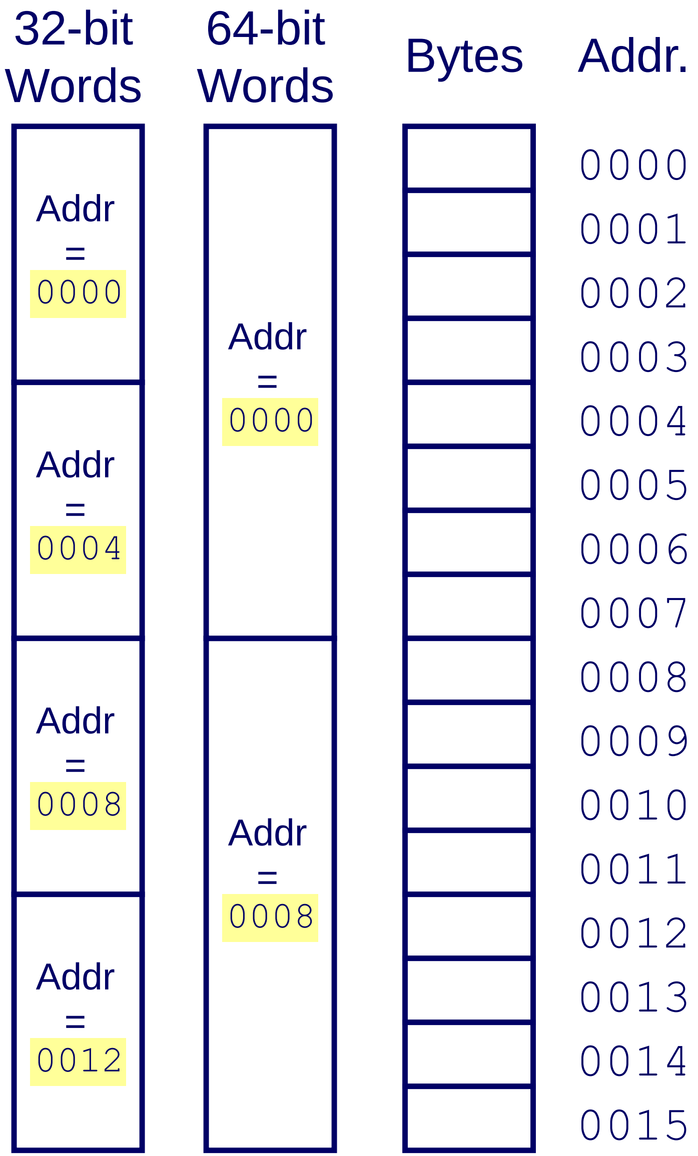


- Programs refer to data by address
  - Conceptually, envision it as a very large array of bytes
    - In reality, it's not, but can think of it that way
  - An address is like an index into that array
    - and, a pointer variable stores an address

- 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

Word-Oriented Memory Organization

- Addresses Specify Byte Locations
  - Address of first byte in word
  - 32 bit OS supports 4 GBytes (2^30 x 4 x 1 byte)
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



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
- 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		

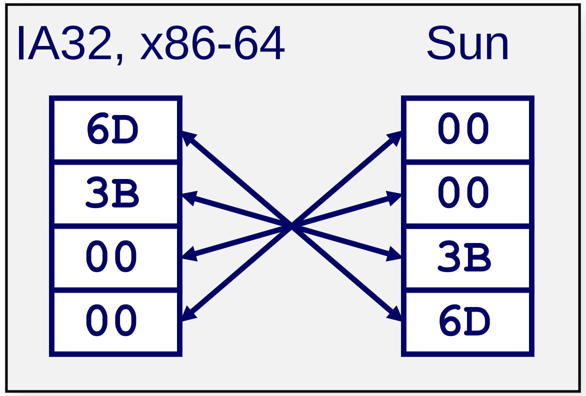
Little Endian

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

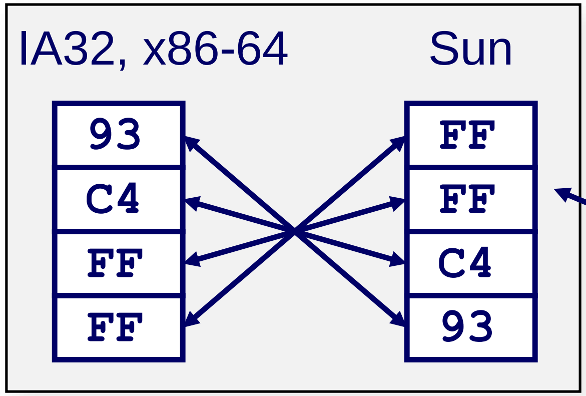
Representing Integers

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

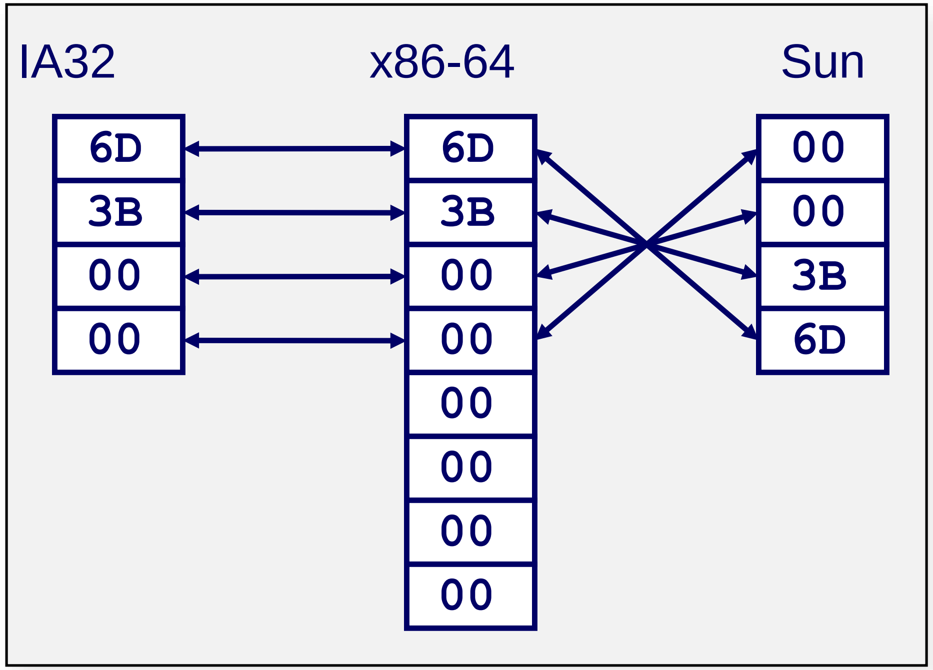
```
int A = 15213;
```




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




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



Two’s complement representation