15-213

"The course that gives CMU its Zip!"

Floating Point Arithmetic
Feb 17, 2000

Topics

- **IEEE Floating Point Standard**
- **Rounding**
- **Floating Point Operations**
- **Mathematical properties**
- **IA32 floating point**

Floating Point Puzzles

- **For each of the following C expressions, either:**
  - Argue that 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 == (float) d`
- `f == -(-f);`
- `2/3 == 2/3.0`
- `d < 0.0` ⇒ `((d*2) < 0.0)`
- `d > f` ⇒ `-f < -d`
- `d * d >= 0.0`
- `(d+f)-d == f`

IEEE Floating Point

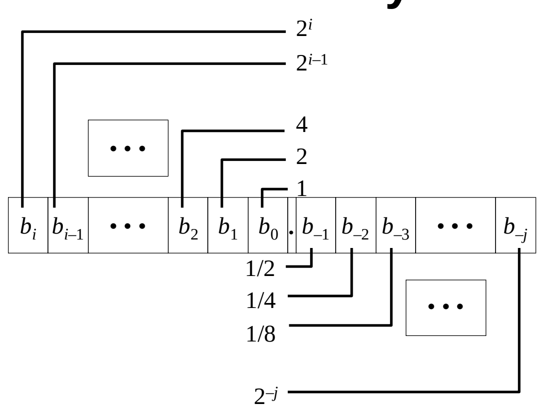
IEEE Standard 754

- **Estabilished 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 go fast**
  - Numerical analysts predominated over hardware types in defining standard

Fractional Binary Numbers



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 Number Examples

Value	Representation
5-3/4	`101.11`$_2$
2-7/8	`10.111`$_2$
63/64	`0.111111`$_2$

Observation

- Divide by 2 by shifting right
- Numbers of form `0.111111…`$_2$ just below 1.0
  - Use notation 1.0 – ε

Limitation

- 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]…`$_2$

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

s	exp	frac

- MSB is sign bit
- exp field encodes E
- `frac` field encodes M

Sizes

- Single precision: 8 exp bits, 23 `frac` bits
  - 32 bits total
- Double precision: 11 exp bits, 52 `frac` bits
  - 64 bits total

"Normalized" Numeric Values

Condition

- **exp ≠ 000…0 and exp ≠ 111…1**

Exponent coded as *biased* value

E* = *Exp* – *Bias

- *Exp* : unsigned value denoted by **exp**
- *Bias* : Bias value
  - » Single precision: 127 (*Exp*: 1…254, *E*: -126…127)
  - » Double precision: 1023 (*Exp*: 1…2046, *E*: -1022…1023)
  - » in general: *Bias* = 2^{m-1} - 1, where m is the number of exponent bits

Significand coded with implied leading 1

***m* = 1.xxx…x$_2$**

- **xxx…x**: bits of `frac`
- Minimum when **000…0** (*M* = 1.0)
- Maximum when **111…1** (*M* = 2.0 – ε)
- Get extra leading bit for "free"

Normalized Encoding Example

Value

Float F = 15213.0;

- 15213_{10} = 11101101101101_2 = 1.1101101101101_2 X 2^{13}

Significand

M = $1.\underline{1101101101101}_2$

frac = $\underline{\mathbf{1101101101101}}\mathbf{0000000000}_2$

Exponent

E = **13**

Bias = **127**

Exp = **140** = **10001100_2**

Floating Point Representation (Class 02):

Hex:	**4**	**6**	**6**	**D**	**B**	**4**	**0**	**0**
Binary:	**0100**	**0110**	**0110**	**1101**	**1011**	**0100**	**0000**	**0000**
140:	**100**	**0110**	**0**					
15213:			***1*110**	**1101**	**1011**	**01**		

Denormalized Values

Condition

- **exp = 000…0**

Value

- **Exponent value *E* = –*Bias* + 1**
- **Significand value *m* = 0.xxx…x$_2$**
  - **xxx…x:** bits of frac

Cases

- **exp = 000…0, frac = 000…0**
  - Represents value 0
  - Note that have distinct values +0 and –0
- **exp = 000…0, frac ≠ 000…0**
  - Numbers very close to 0.0
  - Lose precision as get smaller
  - “Gradual underflow”

Interesting Numbers

Description	**exp**	**frac**	**Numeric Value**
Zero	00…00	00…00	0.0
Smallest Pos. Denorm. • Single ≈ 1.4×10^{-45} • Double ≈ 4.9×10^{-324}	00…00	00…01	$2^{-\{23,52\}} \times 2^{-\{126,1022\}}$
Largest Denormalized • Single ≈ 1.18×10^{-38} • Double ≈ 2.2×10^{-308}	00…00	11…11	$(1.0 - \varepsilon) \times 2^{-\{126,1022\}}$
Smallest Pos. Normalized • Just larger than largest denormalized	00…01	00…00	$1.0 \times 2^{-\{126,1022\}}$
One	01…11	00…00	1.0
Largest Normalized • Single ≈ 3.4×10^{38} • Double ≈ 1.8×10^{308}	11…10	11…11	$(2.0 - \varepsilon) \times 2^{\{127,1023\}}$

Special Values

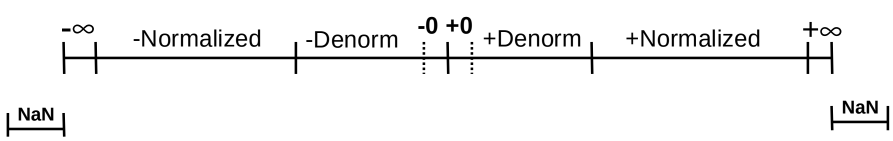
Condition

- **`exp` = `111…1`**

Cases

- **`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 = +∞, 1.0/–0.0 = –∞
- **`exp` = `111…1`, `frac` ≠ `000…0`**
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), ∞ – ∞

Summary of Floating Point Real Number Encodings



Tiny floating point example

Assume an 8-bit floating point representation where

- **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`**

- **Otherwise, the rules are the same as IEEE floating point format (normalized, denormalized, representation of 0, NaN, infinity)**

7	6 3	2 0
s	exp	frac

Values related to the exponent

Exp	exp	E
0	0000	-6 (denorms)
1	0001	-6
2	0010	-5
3	0011	-4
4	0100	-3
5	0101	-2
6	0110	-1
7	0111	0
8	1000	1
9	1001	2
10	1010	3
11	1011	4
12	1100	5
13	1101	6
14	1110	7
15	1111	(inf, Nan)

Dynamic Range

	exp	E	value	
	0 0000 000	n/a	0	
	0 0000 001	-6	1/512	closest to zero
Denormalized numbers	0 0000 010	-6	2/512	
	…			
	0 0000 110	-6	6/512	
	0 0000 111	-6	7/512	largest denorm
	0 0001 000	-6	8/512	smallest norm
	0 0001 001	-6	9/512	
	…			
	0 0110 110	-1	28/32	
	0 0110 111	-1	30/32	closest to 1 below
Normalized numbers	0 0111 000	0	1	
	0 0111 001	0	36/32	closest to 1 above
	0 0111 010	0	40/32	
	…			
	0 1110 110	7	224	
	0 1110 111	7	240	largest norm
	0 1111 000	n/a	inf	

Special Properties of 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

Floating Point Operations

Conceptual View

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

Rounding Modes (illustrate with $ rounding)

	$1.40	**$1.60**	**$1.50**	**$2.50**	**–$1.50**
• **Zero**	$1.00	$1.00	$1.00	$2.00	–$1.00
• **Round down (-∞)**	$1.00	$1.00	$1.00	$2.00	–$2.00
• **Round up (+∞)**	$2.00	$2.00	$2.00	$3.00	–$1.00
• **Nearest Even (default)**	$1.00	$2.00	$2.00	$2.00	–$2.00

Note:
1. Round down: rounded result is close to but no greater than true result.
2. Round up: rounded result is close to but no less than true result.

A 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

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

1.2349999	1.23	(Less than half way)
1.2350001	1.24	(Greater than half way)
1.2350000	1.24	(Half way—round up)
1.2450000	1.24	(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 = $100\ldots_2$

Examples

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

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

FP Multiplication

Operands

$(-1)^{s1}\ M1\ \ 2^{E1}$

$(-1)^{s2}\ M2\ \ 2^{E2}$

Exact Result

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

- **Sign** *s***:** $s1$ ^ $s2$
- **Significand** *M***:** $M1 * M2$
- **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**

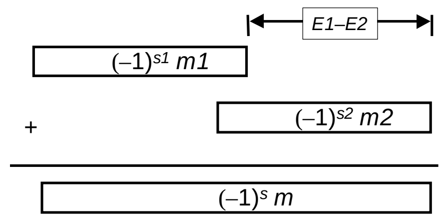
FP Addition

Operands

$(-1)^{s1}\ M1\ \ 2^{E1}$

$(-1)^{s2}\ M2\ \ 2^{E2}$

- **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**

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
- **0 is additive identity?** **YES**
- **Every element has additive inverse** **ALMOST**
  - Except for infinities & NaNs

Montonicity

- $a \geq b \Rightarrow a+c \geq b+c$**?** **ALMOST**
  - Except for infinities & NaNs

Algebraic 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
- **1 is multiplicative identity?** **YES**
- **Multiplication distributes over addtion?** **NO**
  - Possibility of overflow, inexactness of rounding

Montonicity

- $a \geq b$ & $c \geq 0 \Rightarrow a * c \geq b * c$**?** **ALMOST**
  - Except for infinities & NaNs

Floating Point in C

C Supports Two Levels

`float` **single precision**

`double` **double precision**

Conversions

- **Casting between `int`, `float`, and `double` changes numeric values**
- **`Double` or `float` to `int`**
  - Truncates fractional part
  - Like rounding toward zero
  - Not defined when out of range
    - » Generally saturates to TMin or TMax
- **`int` to `double`**
  - Exact conversion, as long as int has ≤ 54 bit word size
- **`int` to `float`**
  - Will round according to rounding mode

Answers to Floating Point Puzzles

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

Assume neither d nor f is NAN

- `x == (int)(float) x` — No: 24 bit significand
- `x == (int)(double) x` — Yes: 53 bit significand
- `f == (float)(double) f` — Yes: increases precision
- `d == (float) d` — No: loses precision
- `f == -(-f);` — Yes: Just change sign bit
- `2/3 == 2/3.0` — No: 2/3 == 1
- `d < 0.0` ⇒ `((d*2) < 0.0)` — Yes!
- `d > f` ⇒ `-f < -d` — Yes!
- `d * d >= 0.0` — Yes!
- `(d+f)-d == f` — No: Not associative

x86 Floating Point

History

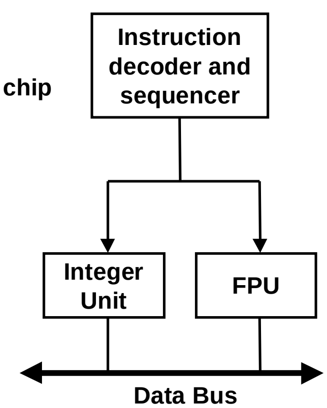
- 8086: first computer to implement IEEE fp
  - separate 8087 FPU (floating point unit)
- 486: merged FPU and Integer Unit onto one chip

Summary

- Hardware to add, multiply, and divide
- Floating point data registers
- Various control & status registers

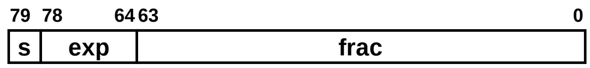
Floating Point Formats

- single precision (C `float`): 32 bits
- double precision (C `double`): 64 bits
- extended precision: 80 bits



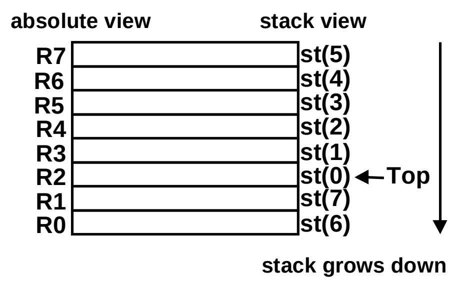
FPU Data Register Stack

FPU register format (extended precision)



FPU register stack

- stack grows down
  - wraps around from R0 -> R7
- FPU registers are typically referenced relative to top of stack
  - st(0) is top of stack (Top)
  - followed by st(1), st(2),…
- push: increment Top, load
- pop: store, decrement Top



FPU instructions

Large number of floating point instructions and formats

- ~50 basic instruction types
- load, store, add, multiply
- sin, cos, tan, arctan, and log!

Sampling of instructions:

Instruction	Effect	Description
`fldz`	`push 0.0`	Load zero
`flds S`	`push S`	Load single precision real
`fmuls S`	`st(0) <- st(0)*S`	Multiply
`faddp`	`st(1) <- st(0)+st(1); pop`	Add and pop

Floating Point Code Example

Compute Inner Product of Two Vectors

- Single precision arithmetic
- Scientific computing and signal processing workhorse

```
float ipf (float x[],
           float y[],
           int n)
{
  int i;
  float result = 0.0;

  for (i = 0; i < n; i++) {
    result += x[i] * y[i];
  }
  return result;
}
```

```
   pushl %ebp               # setup
   movl %esp,%ebp
   pushl %ebx

   movl 8(%ebp),%ebx        # %ebx=&x
   movl 12(%ebp),%ecx       # %ecx=&y
   movl 16(%ebp),%edx       # %edx=n
   fldz                     # push +0.0
   xorl %eax,%eax           # i=0
   cmpl %edx,%eax           # if i>=n done
   jge .L3
.L5:
   flds (%ebx,%eax,4)       # push x[i]
   fmuls (%ecx,%eax,4)      # st(0)*=y[i]
   faddp                    # st(1)+=st(0); pop
   incl %eax                # i++
   cmpl %edx,%eax           # if i<n repeat
   jl .L5
.L3:
   movl -4(%ebp),%ebx       # finish
   leave
   ret                      # st(0) = result
```


Inner product stack trace

1. `fldz`

st(0)	0

2. `flds (%ebx,%eax,4)`

st(1)	0
st(0)	x[0]

3. `fmuls (%ecx,%eax,4)`

st(1)	0
st(0)	x[0]*y[0]

4. `faddp %st,%st(1)`

st(0)	0 + x[0]*y[0]

5. `flds (%ebx,%eax,4)`

st(1)	0 + x[0]*y[0]
st(0)	x[1]

6. `fmuls (%ecx,%eax,4)`

st(1)	0 + x[0]*y[0]
st(0)	x[1]*y[1]

7. `faddp %st,%st(1)`

st(0)	0 + x[0]*y[0] + x[1]*y[1]