15-213

"The course that gives CMU its Zip!"

Floating Point Arithmetic
Sept. 24, 1998

Topics

- **IEEE Floating Point Standard**
- **Rounding**
- **Floating Point Operations**
- **Mathematical properties**
- **Alpha 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

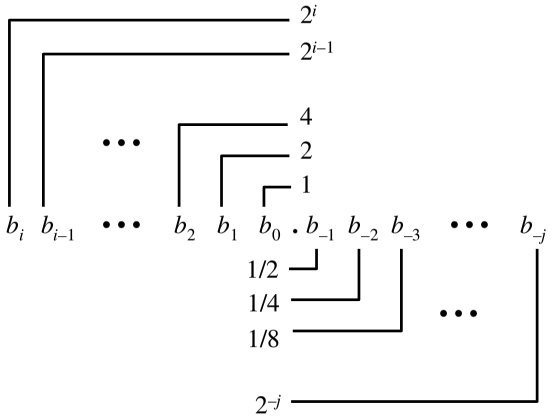
IEEE Standard 754

- **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**
  - Numercial 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 $0.111111\ldots_2$ just below 1.0**
  - Use notation $1.0 - \varepsilon$

Limitation

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

Value	Representation
1/3	$0.0101010101[01]\ldots_2$
1/5	$0.001100110011[0011]\ldots_2$
1/10	$0.0001100110011[0011]\ldots_2$

Floating Point Representation

Numerical Form

- $-1^s\ m\ 2^E$
  - Sign bit s determines whether number is negative or positive
  - Mantissa m normally a fractional value in range [1.0,2.0).
  - Exponent E weights value by power of two

Encoding

- **MSB is sign bit**
- **Exp field encodes *E***
- **Significand field encodes *m***

Sizes

- **Single precision: 8 exp bits, 23 significand bits**
  - 32 bits total
- **Double precision: 11 exp bits, 52 significand bits**
  - 64 bits total

"Normalized" Numeric Values

Condition

- **`exp` $\neq$ `000...0` and `exp` $\neq$ `111...1`**

Exponent coded as *biased* value

E* = *Exp* – *Bias

- *Exp* : unsigned value denoted by **`exp`**
- *Bias* : Bias value
  - » Single precision: 127
  - » Double precision: 1023

Mantissa coded with implied leading 1

***m* = `1.xxx...x`$_2$**

- **`xxx...x`**: bits of significand
- Minimum when **`000...0`** ($m = 1.0$)
- Maximum when **`111...1`** ($m = 2.0 - \varepsilon$)
- Get extra leading bit for "free"

Normalized Encoding Example

Value

`Float F = 15213.0;`

- $15213_{10} = 11101101101101_2 = 1.1101101101101_2 \times 2^{13}$

Significand

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

`sig` = $\underline{1101101101101}0000000000_2$

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**
- **Mantissa value *m* = 0.xxx…x_2**
  - **xxx…x:** bits of significand

Cases

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

Interesting Numbers

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

Memory Referencing Bug Example

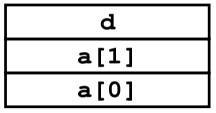
From Class 01

```
main ()
{
  long int a[2];
  double d = 3.14;
  a[2] = 1073741824; /* Out of bounds reference */
  printf("d = %.15g\n", d);
  exit(0);
}
```

	Alpha	MIPS	Sun
-g	5.30498947741318e-315	3.1399998664856	3.14
-O	3.14	3.14	3.14

Referencing Bug on Alpha

Alpha Stack Frame (-g)




```
long int a[2];
double d = 3.14;
a[2] = 1073741824;
```

Optimized Code

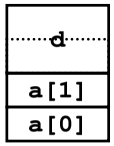
- **Double d stored in register**
- **Unaffected by errant write**

Alpha -g

- **1073741824 = 0x40000000 =** 2^{30}
- **Overwrites all 8 bytes with value 0x0000000040000000**
- **Denormalized value** 2^{30} X (smallest denorm 2^{-1074}) **=** 2^{-1044}
- ≈ 5.305 X 10^{-315}

Referencing Bug on MIPS

MIPS Stack Frame (-g)




```
long int a[2];
double d = 3.14;
a[2] = 1073741824;
```

MIPS -g

- **Overwrites lower 4 bytes with value** `0x40000000`
- **Original value** `3.14` **represented as** `0x40091eb851eb851f`
- **Modified value represented as** `0x40091eb840000000`
- *Exp* = 1024 *E* = 1024–1023 = 1
- **Mantissa difference:** $.0000011eb851f_{16}$
- **Integer value:** $11eb851f_{16} = 300{,}647{,}711_{10}$
- **Difference =** $2^1 \times 2^{-52} \times 300{,}647{,}711 \approx 1.34 \times 10^{-7}$
- **Compare to** 3.140000000 – 3.139999866 = 0.000000134

Special Values

Condition

- `exp = 111...1`

Cases

- `exp = 111...1, significand = 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, significand ≠ 000...0`
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), ∞ – ∞
  - No fixed meaning assigned to significand bits

Special Properties of Encoding

FP Zero Same as Integer Zero

- **All bits = 0**

Can (Almost) Use Unsigned Integer Comparison

- **Must first compare sign bits**
- **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 significand

Rounding Modes (illustrate with $ rounding)

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

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**:** $s1$ ^ $s2$
- **Mantissa** m**:** $m1 * m2$
- **Exponent** E**:** $E1 + E2$

Fixing

- **Overflow if** E **out of range**
- **Round** m **to fit significand precision**

Implementation

- **Biggest chore is multiplying mantissas**

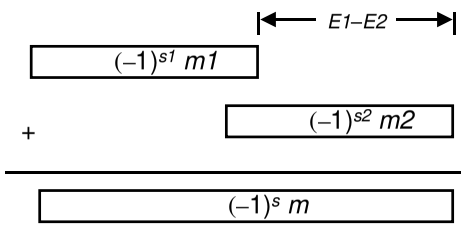
FP Addition

Operands

$(-1)^{s1}\ m1\ 2^{E1}$

$(-1)^{s2}\ m2\ 2^{E2}$

- **Assume** $E1 > E2$



Exact Result

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

- **Sign** s**, mantissa** m**:**
  - Result of signed align & add
- **Exponent** E**:** $E1 - E2$

Fixing

- **Shift** m **right, increment** E **if** $m \geq 2$
- **Shift** m **left** k **positions, decrement** E **by** k **if** $m < 1$
- **Overflow if** E **out of range**
- **Round** m **to fit significand 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

- $a \leq b \Rightarrow a+c \leq b+c$? **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

- $a \leq b \ \& \ c \geq 0 \Rightarrow a*c \leq b*c$? **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 mantissa**
- `x == (int)(double) x` — **Yes: 53 bit mantissa**
- `f == (float)(double) f` — **Yes: increases precision**
- `d == (float) d` — **No: looses 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**

Alpha Floating Point

Implemented as Separate Unit

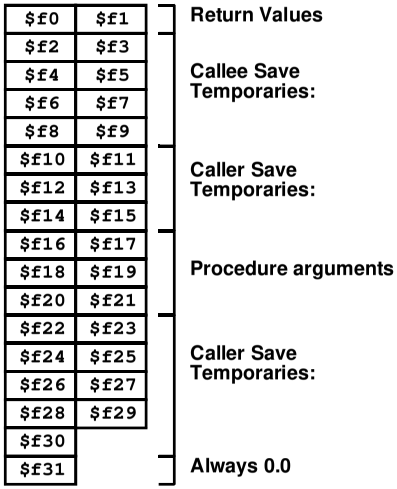
- Hardware to add, multiply, and divide
- Floating point data registers
- Various control & status registers

Floating Point Formats

- S_Floating (C `float`): 32 bits
- T_Floating (C `double`): 64 bits

Floating Point Data Registers

- 32 registers, each 8 bytes
- Labeled `$f0` to `$f31`
- `$f31` is always `0.0`



Floating Point Code Example

Compute Inner Product of Two Vectors

- Single precision arithmetic

```
float inner_prodF
  (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;
}
```

```
     cpys $f31,$f31,$f0 # result = 0.0
     bis $31,$31,$3     # i = 0
     cmplt $31,$18,$1   # 0 < n?
     beq $1,$102        # if not, skip loop
     .align 5
$104:
     s4addq $3,0,$1     # $1 = 4 * i
     addq $1,$16,$2     # $2 = &x[i]
     addq $1,$17,$1     # $1 = &y[i]
     lds $f1,0($2)      # $f1 = x[i]
     lds $f10,0($1)     # $f10 = y[i]
     muls $f1,$f10,$f1  # $f1 = x[i] * y[i]
     adds $f0,$f1,$f0   # result += $f1
     addl $3,1,$3       # i++
     cmplt $3,$18,$1    # i < n?
     bne $1,$104        # if so, loop
$102:
     ret $31,($26),1    # return
```


Numeric Format Conversion

Between Floating Point and Integer Formats

- Special conversion instructions `cvttq`, `cvtqt`, `cvtts`, `cvtst`, ...
- Convert source operand in one format to destination in other
- Both source & destination must be FP register
  - Transfer to and from GP registers via memory store/load

C Code

```
float double2float(double d)
{
  return (float) d;
}
```

Conversion Code

```
cvtts $f16,$f0
```

[Convert T_Floating to S_Floating]

```
double long2double(long i)
{
  return (double) i;
}
```

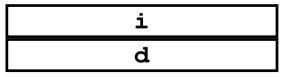
```
stq $16,0($30)
ldt $f1,0($30)
cvtqt $f1,$f0
```

[Pass through stack and convert]

Getting FP Bit Pattern

```
double bit2double(long i)
{
  union {
    long i;
    double d;
  } arg;
  arg.i = i;
  return arg.d;
}
```

```
double long2double(long i)
{
  return (double) i;
}
```




```
stq $16,0($30)
ldt $f0,0($30)
```

```
stq $16,0($30)
ldt $f1,0($30)
cvtqt $f1,$f0
```

- Union provides direct access to bit representation of double
- `bit2double` generates double with given bit pattern
  - NOT the same as `(double) i`
  - Bypasses rounding step