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

```c
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

• 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 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

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-3/4</td>
<td>101.11₂</td>
</tr>
<tr>
<td>2-7/8</td>
<td>10.111₂</td>
</tr>
<tr>
<td>63/64</td>
<td>0.111111₁₂</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/3</td>
<td>0.010101010101[01]...₂</td>
</tr>
<tr>
<td>1/5</td>
<td>0.001100110011[0011]...₂</td>
</tr>
<tr>
<td>1/10</td>
<td>0.0001100110011[0011]...₂</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>significand</th>
</tr>
</thead>
</table>

- 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

- \( \exp \neq 000\ldots0 \) and \( \exp \neq 111\ldots1 \)

Exponent coded as **biased** value

\[
E = \text{Exp} - \text{Bias}
\]

- \( \text{Exp} \) : unsigned value denoted by \( \exp \)
- \( \text{Bias} \) : Bias value
  - Single precision: 127
  - Double precision: 1023

Mantissa coded with implied leading 1

\[
m = 1.\text{xxx}\ldots\text{x}_2
\]

- \( \text{xxx}\ldots\text{x} \): bits of significand
- Minimum when \( 000\ldots0 \) \( (m = 1.0) \)
- Maximum when \( 111\ldots1 \) \( (m = 2.0 - \varepsilon) \)
- Get extra leading bit for “free”
Normalized Encoding Example

Value

\[ \text{Float } F = 15213.0; \]
\[ 15213_{10} = 11101101101101_2 \times 2^{13} \]

Significand

\[ m = 1.1101101101101_2 \]
\[ \text{sig} = 11011011011010000000000000_2 \]

Exponent

\[ E = 13 \]
\[ \text{Bias} = 127 \]
\[ \text{Exp} = 140 = 10001100_2 \]

Floating Point Representation (Class 02):

<table>
<thead>
<tr>
<th>Hex</th>
<th>Binary</th>
<th>140</th>
<th>15213</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0100</td>
<td>100</td>
<td>1110</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>0110</td>
<td>1101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>1011</td>
<td>1011</td>
</tr>
<tr>
<td>D</td>
<td>0100</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>B</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>4</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>
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 $\neq 000...0$
  – Numbers very close to 0.0
  – Lose precision as get smaller
  – “Gradual underflow”
Interesting Numbers

<table>
<thead>
<tr>
<th>Description</th>
<th>Exp</th>
<th>Significand</th>
<th>Numeric Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>00...00</td>
<td>00...00</td>
<td>0.0</td>
</tr>
<tr>
<td>Smallest Pos. Denorm.</td>
<td>00...00</td>
<td>00...01</td>
<td>2^−{23,52} X 2^−{126,1022}</td>
</tr>
<tr>
<td>• Single</td>
<td></td>
<td></td>
<td>1.4 X 10^{−45}</td>
</tr>
<tr>
<td>• Double</td>
<td></td>
<td></td>
<td>4.9 X 10^{−324}</td>
</tr>
<tr>
<td>Largest Denormalized</td>
<td>00...00</td>
<td>11...11</td>
<td>(1.0 − ε) X 2^−{126,1022}</td>
</tr>
<tr>
<td>• Single</td>
<td></td>
<td></td>
<td>1.18 X 10^{−38}</td>
</tr>
<tr>
<td>• Double</td>
<td></td>
<td></td>
<td>2.2 X 10^{−308}</td>
</tr>
<tr>
<td>Smallest Pos. Normalized</td>
<td>00...01</td>
<td>00...00</td>
<td>1.0 X 2^−{126,1022}</td>
</tr>
<tr>
<td>• Just larger than largest denormalized</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>One</td>
<td>01...11</td>
<td>00...00</td>
<td>1.0</td>
</tr>
<tr>
<td>Largest Normalized</td>
<td>11...10</td>
<td>11...11</td>
<td>(2.0 − ε) X 2^{127,1023}</td>
</tr>
<tr>
<td>• Single</td>
<td></td>
<td></td>
<td>3.4 X 10^{38}</td>
</tr>
<tr>
<td>• Double</td>
<td></td>
<td></td>
<td>1.8 X 10^{308}</td>
</tr>
</tbody>
</table>
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);
}
Referencing Bug on Alpha

Alpha Stack Frame (-g)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td></td>
</tr>
<tr>
<td>a[1]</td>
<td></td>
</tr>
<tr>
<td>a[0]</td>
<td></td>
</tr>
</tbody>
</table>

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 $2^{30} \times (\text{smallest denorm } 2^{-1074}) = 2^{-1044}$
- $\approx 5.305 \times 10^{-315}$
Referencing Bug on MIPS

MIPS Stack Frame (-g)

```
......d......
 a[1]
 a[0]
```

```
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
  • \( \exp = 111\ldots1 \)

Cases
  • \( \exp = 111\ldots1, \text{significand} = 000\ldots0 \)
    – Represents value \( \infty \) (infinity)
    – Operation that overflows
    – Both positive and negative
    – E.g., \( 1.0/0.0 = -1.0/-0.0 = +\infty, \ 1.0/-0.0 = -\infty \)
  • \( \exp = 111\ldots1, \text{significand} \neq 000\ldots0 \)
    – Not-a-Number (NaN)
    – Represents case when no numeric value can be determined
    – E.g., \( \sqrt{-1}, \infty - \infty \)
    – 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)

<table>
<thead>
<tr>
<th>Value</th>
<th>$1.40</th>
<th>$1.60</th>
<th>$1.50</th>
<th>$2.50</th>
<th>−$1.50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>$1.00</td>
<td>$2.00</td>
<td>$1.00</td>
<td>$2.00</td>
<td>−$1.00</td>
</tr>
<tr>
<td>−∞</td>
<td>$1.00</td>
<td>$2.00</td>
<td>$1.00</td>
<td>$2.00</td>
<td>−$2.00</td>
</tr>
<tr>
<td>+∞</td>
<td>$1.00</td>
<td>$2.00</td>
<td>$2.00</td>
<td>$3.00</td>
<td>−$1.00</td>
</tr>
<tr>
<td>Nearest Even (default)</td>
<td>$1.00</td>
<td>$2.00</td>
<td>$2.00</td>
<td>$2.00</td>
<td>−$2.00</td>
</tr>
</tbody>
</table>
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 significant 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…₂

Examples

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary</th>
<th>Rounded</th>
<th>Action</th>
<th>Rounded Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-3/32</td>
<td>10.00011₂</td>
<td>10.00₂</td>
<td>(&lt;1/2—down)</td>
<td>2</td>
</tr>
<tr>
<td>2-3/16</td>
<td>10.00110₂</td>
<td>10.01₂</td>
<td>(&gt;1/2—up)</td>
<td>2-1/4</td>
</tr>
<tr>
<td>2-7/8</td>
<td>10.11100₂</td>
<td>11.00₂</td>
<td>(1/2—up)</td>
<td>3</td>
</tr>
<tr>
<td>2-5/8</td>
<td>10.10100₂</td>
<td>10.1₀₂</td>
<td>(1/2—down)</td>
<td>2-1/2</td>
</tr>
</tbody>
</table>
FP Multiplication

Operands
\((-1)^{s_1} m_1 \ 2^{E_1}\)
\((-1)^{s_2} m_2 \ 2^{E_2}\)

Exact Result
\((-1)^s m \ 2^E\)
- Sign \(s\): \(s_1 \wedge s_2\)
- Mantissa \(m\): \(m_1 \times m_2\)
- Exponent \(E\): \(E_1 + E_2\)

Fixing
- Overflow if \(E\) out of range
- Round \(m\) to fit significand precision

Implementation
- Biggest chore is multiplying mantissas
FP Addition

Operands

\[(–1)^{s_1} m^1 2^{E_1}\]
\[(–1)^{s_2} m^2 2^{E_2}\]

• Assume \(E_1 > E_2\)

Exact Result

\[(–1)^s m 2^E\]

• Sign \(s\), mantissa \(m\):
  – Result of signed align & add
• Exponent \(E\): \(E_1 – E_2\)

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 addition? **NO**
  - Possibility of overflow, inexactness of rounding

Montonicity

- \( a \leq b \) & \( c \geq 0 \) \Rightarrow a \times c \leq b \times 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

Assume neither d nor f is NAN

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

- `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

Return Values
- Procedure arguments: $f1$, $f3$, $f5$, $f7$, $f9$, $f11$, $f13$, $f15$, $f17$, $f19$, $f21$, $f23$, $f25$, $f27$, $f29$, $f31$
- Caller Save Temporaries: $f0$, $f2$, $f4$, $f6$, $f8$, $f10$, $f12$, $f14$, $f16$, $f18$, $f20$, $f22$, $f24$, $f26$, $f28$, $f30$
- Caller Save Temporaries: $f1$, $f3$, $f5$, $f7$, $f9$, $f11$, $f13$, $f15$, $f17$, $f19$, $f21$, $f23$, $f25$, $f27$, $f29$, $f31$
- Always 0.0
Floating Point Code Example

Compute Inner Product of Two Vectors

- Single precision arithmetic

```c
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`, `cvtt`, `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

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

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

Conversion Code

```c
  cvtts $f16,$f0

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

[Convert T_Floating to S_Floating]

[Pass through stack and convert]```
double bit2double(long i)  
{  
    union {  
        long i;  
        double d;  
    } arg;  
    arg.i = i;  
    return arg.d;  
}

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

- 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