Lecture 6: MIPS Floating Point

September 17, 2009
Floating Point

- Representation for dynamically rescalable numbers
  - Including very small and very large numbers, non-integers
- Like scientific notation
  - $-2.34 \times 10^{56}$
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^9$
- In binary
  - $\pm 1.xxxxxx_2 \times 2^{yyyy}$
- Types float and double in C
Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Two representations
  - Single precision (32-bit)
  - Double precision (64-bit)
IEEE Floating-Point Format

- **S**: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand: $1.0 \leq |\text{significand}| < 2.0$
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the “1.” restored
- **Exponent**: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1203

### Format

<table>
<thead>
<tr>
<th></th>
<th>Single: 8 bits</th>
<th>Single: 23 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Double: 11 bits</td>
<td>Double: 52 bits</td>
</tr>
</tbody>
</table>

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$
Single-Precision Range

- Exponents 00000000 and 11111111 reserved

- Smallest value
  - Exponent: 00000001
    ⇒ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 ⇒ significand = 1.0
  - ±1.0 × 2^{-126} ≈ ±1.2 × 10^{-38}

- Largest value
  - exponent: 11111110
    ⇒ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 ⇒ significand ≈ 2.0
  - ±2.0 × 2^{+127} ≈ ±3.4 × 10^{+38}
Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    ⇒ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 ⇒ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    ⇒ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 ⇒ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$
Floating-Point Precision

- Relative precision
  - all fraction bits are significant
  - Single: approx $2^{-23}$
    - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
  - Double: approx $2^{-52}$
    - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision
Floating-Point Example

- **Represent** $-0.75$
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - $S = 1$
  - Fraction $= 1000\ldots00_2$
  - Exponent $= -1 + $ Bias
    - Single: $-1 + 127 = 126 = 01111110_2$
    - Double: $-1 + 1023 = 1022 = 0111111110_2$
- **Single:** $1011111101000\ldots00$
- **Double:** $1011111111101000\ldots00$
Floating-Point Example

What number is represented by the single-precision float

\[11000000101000\ldots00\]

- \(S = 1\)
- Fraction = 01000\ldots00_2
- Exponent = 10000001_2 = 129

\[x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}\]
\[= (-1) \times 1.25 \times 2^2\]
\[= -5.0\]
Denormal Numbers

- Exponent = 000...0 ⇒ hidden bit is 0
  \[ x = (-1)^S \times (0 + \text{Fraction}) \times 2^{-\text{Bias}} \]

- Smaller than normal numbers
  - allow for gradual underflow, with diminishing precision

- Denormal with fraction = 000...0

\[ x = (-1)^S \times (0 + 0) \times 2^{-\text{Bias}} = \pm 0.0 \]

Two representations of 0.0!
Infinities and NaNs

- **Exponent = 111...1, Fraction = 000...0**
  - ±Infinity
  - Can be used in subsequent calculations, avoiding need for overflow check

- **Exponent = 111...1, Fraction ≠ 000...0**
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - e.g., 0.0 / 0.0
  - Can be used in subsequent calculations
Floating-Point Addition

- Consider a 4-digit decimal example
  - $9.999 \times 10^1 + 1.610 \times 10^{-1}$
- 1. Align decimal points
  - Shift number with smaller exponent
  - $9.999 \times 10^1 + 0.016 \times 10^1$
- 2. Add significands
  - $9.999 \times 10^1 + 0.016 \times 10^1 = 10.015 \times 10^1$
- 3. Normalize result & check for over/underflow
  - $1.0015 \times 10^2$
- 4. Round and renormalize if necessary
  - $1.002 \times 10^2$
Floating-Point Addition

- Now consider a 4-digit binary example
  - \(1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2} (0.5 + -0.4375)\)
- 1. Align binary points
  - Shift number with smaller exponent
  - \(1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}\)
- 2. Add significands
  - \(1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}\)
- 3. Normalize result & check for over/underflow
  - \(1.000_2 \times 2^{-4}\), with no over/underflow
- 4. Round and renormalize if necessary
  - \(1.000_2 \times 2^{-4}\) (no change) = 0.0625
FP Adder Hardware

- Much more complex than integer adder
- Doing it in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles
  - Can be pipelined
FP Adder Hardware

Step 1
- Compare exponents
- Shift smaller number right

Step 2
- Add
- Normalize
- Round

Step 3
- Increment or decrement
- Shift left or right

Step 4
Floating-Point Multiplication

- Consider a 4-digit decimal example
  - $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$
- 1. Add exponents
  - For biased exponents, subtract bias from sum
  - New exponent = $10 + (-5) = 5$
- 2. Multiply significands
  - $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^6$
- 3. Normalize result & check for over/underflow
  - $1.0212 \times 10^6$
- 4. Round and renormalize if necessary
  - $1.021 \times 10^6$
- 5. Determine sign of result from signs of operands
  - $+1.021 \times 10^6$
Floating-Point Multiplication

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2} (0.5 \times -0.4375)$

- 1. Add exponents
  - Unbiased: $-1 + -2 = -3$
  - Biased: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$

- 2. Multiply significands
  - $1.000_2 \times 1.110_2 = 1.1102 \Rightarrow 1.110_2 \times 2^{-3}$

- 3. Normalize result & check for over/underflow
  - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow

- 4. Round and renormalize if necessary
  - $1.110_2 \times 2^{-3}$ (no change)

- 5. Determine sign: $+ve \times -ve \Rightarrow -ve$
  - $-1.110_2 \times 2^{-3} = -0.21875$
FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP ↔ integer conversion
- Operations usually takes several cycles
  - Can be pipelined
FP Instructions in MIPS

- FP hardware is coprocessor 1
  - Adjunct processor that extends the ISA
- Separate FP registers
  - 32 single-precision: $f0, f1, \ldots, f31$
  - Paired for double-precision: $f0/f1, f2/f3, \ldots$
    - Release 2 of MIPS ISA supports $32 \times 64$-bit FP reg’s
- FP instructions operate only on FP registers
  - Programs generally don’t do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact
- FP load and store instructions
  - lwc1, ldc1, swc1, sdc1
    - e.g., ldc1 $f8, 32(sp)
FP Instructions in MIPS

- Single-precision arithmetic
  - add.s, sub.s, mul.s, div.s
  - e.g., add.s $f0, $f1, $f6
- Double-precision arithmetic
  - add.d, sub.d, mul.d, div.d
  - e.g., mul.d $f4, $f4, $f6
- Single- and double-precision comparison
  - c.xx.s, c.xx.d (xx is eq, lt, le, …)
  - Sets or clears FP condition-code bit
    - e.g. c.lt.s $f3, $f4
- Branch on FP condition code true or false
  - bc1t, bc1f
    - e.g., bc1t TargetLabel
FP Example: °F to °C

- C code:
  ```c
  float f2c (float fahr) {
    return ((5.0/9.0)*(fahr - 32.0));
  }
  ```
  - `fahr` in $f12$, result in $f0$, literals in global memory space

- Compiled MIPS code:
  ```assembly
  f2c: lwc1  $f16, const5($gp)
    lwc2  $f18, const9($gp)
    div.s $f16, $f16, $f18
    lwc1  $f18, const32($gp)
    sub.s $f18, $f12, $f18
    mul.s $f0,  $f16, $f18
    jr    $ra
  ```
X = X + Y × Z

- All 32 × 32 matrices, 64-bit double-precision elements

C code:
```c
void mm (double x[][],
        double y[][], double z[][]) {
    int i, j, k;
    for (i = 0; i! = 32; i = i + 1)
        for (j = 0; j! = 32; j = j + 1)
            for (k = 0; k! = 32; k = k + 1)
                x[i][j] = x[i][j]
                            + y[i][k] * z[k][j];
}
```

- Addresses of x, y, z in $a0, a1, a2$, and i, j, k in $s0, s1, s2$
FP Example: Array Multiplication

MIPS code:

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>li</td>
<td>$t1, 32</td>
<td>#$t1 = 32 (row size/loop end)</td>
</tr>
<tr>
<td>li</td>
<td>$s0, 0</td>
<td>#i = 0; initialize 1st for loop</td>
</tr>
<tr>
<td>L1:</td>
<td>li</td>
<td>$s1, 0; restart 2nd for loop</td>
</tr>
<tr>
<td>L2:</td>
<td>li</td>
<td>$s2, 0; restart 3rd for loop</td>
</tr>
<tr>
<td>sll</td>
<td>$t2, $s0, 5</td>
<td>#$t2 = i * 32 (size of row of x)</td>
</tr>
<tr>
<td>addu</td>
<td>$t2, $t2, $s1</td>
<td>#$t2 = i * size(row) + j</td>
</tr>
<tr>
<td>sll</td>
<td>$t2, $t2, 3</td>
<td>#$t2 = byte offset of [i][j]</td>
</tr>
<tr>
<td>addu</td>
<td>$t2, $a0, $t2</td>
<td>#$t2 = byte address of x[i][j]</td>
</tr>
<tr>
<td>l.d</td>
<td>$f4, 0($t2)</td>
<td>#$f4 = 8 bytes of x[i][j]</td>
</tr>
<tr>
<td>L3:</td>
<td>sll</td>
<td>$t0, $s2, 5</td>
</tr>
<tr>
<td>addu</td>
<td>$t0, $t0, $s1</td>
<td>#$t0 = k * size(row) + j</td>
</tr>
<tr>
<td>sll</td>
<td>$t0, $t0, 3</td>
<td>#$t0 = byte offset of [k][j]</td>
</tr>
<tr>
<td>addu</td>
<td>$t0, $a2, $t0</td>
<td>#$t0 = byte address of z[k][j]</td>
</tr>
<tr>
<td>l.d</td>
<td>$f16, 0($t0)</td>
<td>#$f16 = 8 bytes of z[k][j]</td>
</tr>
<tr>
<td>…</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FP Example: Array Multiplication

```
...  
 sll $t0, $s0, 5     # $t0 = i*32 (size of row of y)
 addu $t0, $t0, $s2   # $t0 = i*size(row) + k
 sll $t0, $t0, 3      # $t0 = byte offset of [i][k]
 addu $t0, $a1, $t0   # $t0 = byte address of y[i][k]
 l.d $f18, 0($t0)     # $f18 = 8 bytes of y[i][k]
 mul.d $f16, $f18, $f16 # $f16 = y[i][k] * z[k][j]
 add.d $f4, $f4, $f16  # f4=x[i][j] + y[i][k]*z[k][j]
 addiu $s2, $s2, 1   # $k = k + 1
 bne $s2, $t1, L3     # if (k != 32) go to L3
 s.d $f4, 0($t2)      # x[i][j] = $f4
 addiu $s1, $s1, 1    # $j = j + 1
 bne $s1, $t1, L2     # if (j != 32) go to L2
 addiu $s0, $s0, 1    # $i = i + 1
 bne $s0, $t1, L1     # if (i != 32) go to L1
```
Accurate Arithmetic

- IEEE Std 754 specifies additional rounding control
  - Extra bits of precision (guard, round, sticky)
  - Choice of rounding modes
  - Allows programmer to fine-tune numerical behavior of a computation
- Not all FP units implement all options
  - Most programming languages and FP libraries just use defaults
- Trade-off between hardware complexity, performance, and market requirements
Interpretation of Data

- Bits have no inherent meaning
  - Interpretation depends on the instructions applied
- Computer representations of numbers
  - Finite range and precision
  - Need to account for this in programs
Associativity

- Parallel programs may interleave operations in unexpected orders
  - Assumptions of associativity may fail

<table>
<thead>
<tr>
<th></th>
<th>(x+y)+z</th>
<th>x+(y+z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>-1.50E+38</td>
<td>-1.50E+38</td>
</tr>
<tr>
<td>y</td>
<td>1.50E+38</td>
<td>0.00E+00</td>
</tr>
<tr>
<td>z</td>
<td>1.0</td>
<td>1.50E+38</td>
</tr>
<tr>
<td></td>
<td>1.00E+00</td>
<td>0.00E+00</td>
</tr>
</tbody>
</table>

- Need to validate parallel programs under varying degrees of parallelism
x86 FP Architecture

- Originally based on 8087 FP coprocessor
  - 8 × 80-bit extended-precision registers
  - Used as a push-down stack
  - Registers indexed from TOS: ST(0), ST(1), …
- FP values are 32-bit or 64 in memory
  - Converted on load/store of memory operand
  - Integer operands can also be converted on load/store
- Very difficult to generate and optimize code
  - Result: poor FP performance
# x86 FP Instructions

<table>
<thead>
<tr>
<th>Data transfer</th>
<th>Arithmetic</th>
<th>Compare</th>
<th>Transcendental</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD mem/ST(i)</td>
<td>FADDP mem/ST(i)</td>
<td>FICOMP</td>
<td>FPATAN</td>
</tr>
<tr>
<td>FISTP mem/ST(i)</td>
<td>FISUBRP mem/ST(i)</td>
<td>FIUCOMP</td>
<td>F2XMI</td>
</tr>
<tr>
<td>FLDPI</td>
<td>FMULP mem/ST(i)</td>
<td></td>
<td>FCOS</td>
</tr>
<tr>
<td>FLD1</td>
<td>FIDIVRP mem/ST(i)</td>
<td></td>
<td>FPTAN</td>
</tr>
<tr>
<td>FLDZ</td>
<td>FSQRT</td>
<td>FSTSW AX/mem</td>
<td>FPREM</td>
</tr>
<tr>
<td></td>
<td>FABS</td>
<td></td>
<td>FPSIN</td>
</tr>
<tr>
<td></td>
<td>FRNDINT</td>
<td></td>
<td>FYL2X</td>
</tr>
</tbody>
</table>

- **Optional variations**
  - **I**: integer operand
  - **P**: pop operand from stack
  - **R**: reverse operand order
  - But not all combinations allowed
Streaming SIMD Extension 2 (SSE2)

- Adds $4 \times 128$-bit registers
  - Extended to 8 registers in AMD64/EM64T
- Can be used for multiple FP operands
  - $2 \times 64$-bit double precision
  - $4 \times 32$-bit double precision
  - Instructions operate on them simultaneously
    - Single-Instruction Multiple-Data
Right Shift and Division

- Left shift by \( i \) places multiplies an integer by \( 2^i \)
- Right shift divides by \( 2^i \)?
  - Only for unsigned integers
- For signed integers
  - Arithmetic right shift: replicate the sign bit
  - e.g., \(-5 / 4\)
    - \( 1111011_2 \gg 2 = 1111110_2 = -2 \)
    - Rounds toward \(-\infty\)
  - c.f. \( 1111011_2 \gg 2 = 00111110_2 = +62 \)
Who Cares About FP Accuracy?

- Important for scientific code
  - But for everyday consumer use?
    - “My bank balance is out by 0.0002¢!” 😁

- The Intel Pentium FDIV bug
  - The market expects accuracy
  - See Colwell, *The Pentium Chronicles*
Concluding Remarks

- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow and underflow
- MIPS ISA
  - Core instructions: 54 most frequently used
    - 100% of SPECINT, 97% of SPECFP
  - Other instructions: less frequent
Next Lecture

- Performance evaluation
- Micro-architecture introduction
  - Chapter 4.1 – 4.4