ECE 154A Introduction to Computer Architecture
Fall 2012

Dmitri Strukov

Lecture 9
Computer arithmetic and floating point
Multiplication

• Start with long-multiplication approach

\[
\begin{array}{c}
\text{multiplicand} \\
1000 \\
\times \quad 1001 \\
\hline
1000 \\
0000 \\
0000 \\
1000 \\
\hline
1001000
\end{array}
\]

Length of product is the sum of operand lengths
Multiplication Hardware
Optimized Multiplier

- Perform steps in parallel: add/shift

- One cycle per partial-product addition
  - That’s ok, if frequency of multiplications is low
Faster Multiplier
• Uses multiple adders
  – Cost/performance tradeoff

■ Can be pipelined
  ■ Several multiplication performed in parallel
MIPS Multiplication

• Two 32-bit registers for product
  – HI: most-significant 32 bits
  – LO: least-significant 32-bits

• Instructions
  – \texttt{mult rs, rt} / \texttt{multu rs, rt}
    • 64-bit product in HI/LO
  – \texttt{mfhi rd} / \texttt{mflo rd}
    • Move from HI/LO to rd
    • Can test HI value to see if product overflows 32 bits
  – \texttt{mul rd, rs, rt}
    • Least-significant 32 bits of product $\rightarrow$ rd
Division

- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required

$n$-bit operands yield $n$-bit quotient and remainder
Division Hardware

Initially divisor in left half

Initially dividend

1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register

2a. Shift the Quotient register to the left, setting the new rightmost bit to 1

2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0

3. Shift the Divisor register right 1 bit

33rd repetition? No: < 33 repetitions

Done
Optimized Divider

- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
  - Same hardware can be used for both
Faster Division

• Can’t use parallel hardware as in multiplier
  – Subtraction is conditional on sign of remainder
• Faster dividers (e.g. SRT division) generate multiple quotient bits per step
  – Still require multiple steps
MIPS Division

• Use HI/LO registers for result
  – HI: 32-bit remainder
  – LO: 32-bit quotient

• Instructions
  – `div rs, rt` / `divu rs, rt`
  – No overflow or divide-by-0 checking
    • Software must perform checks if required
  – Use `mfhi`, `mflo` to access result
Floating Point

Partially adapted from Computer Organization and Design, 4th edition, Patterson and Hennessy
Floating Point

• Representation for non-integral numbers
  – Including very small and very large numbers

• Like scientific notation
  – $-2.34 \times 10^{56}$ normalized
  – $+0.002 \times 10^{-4}$ not normalized
  – $+987.02 \times 10^{9}$

• In binary
  – $\pm 1.xxxxxxx_2 \times 2^{yyyy}$

• 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

<table>
<thead>
<tr>
<th></th>
<th>single: 8 bits</th>
<th>single: 23 bits</th>
<th>double: 11 bits</th>
<th>double: 52 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Exponent</td>
<td>Fraction</td>
<td>Exponent</td>
<td>Fraction</td>
</tr>
</tbody>
</table>

\[
x = (-1)^S \times (1+\text{Fraction}) \times 2^{(\text{Exponent-Bias})}
\]

- **S**: sign bit (0 \(\Rightarrow\) non-negative, 1 \(\Rightarrow\) negative)
- **Normalized 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
Single-Precision Range

• Exponents 00000000 and 11111111 reserved
• Smallest value
  – Exponent: 00000001
    \[\Rightarrow\text{actual exponent} = 1 - 127 = -126\]
  – Fraction: 000...00 \[\Rightarrow\text{significand} = 1.0\]
  – \[±1.0 \times 2^{-126} \approx ±1.2 \times 10^{-38}\]
• Largest value
  – exponent: 11111110
    \[\Rightarrow\text{actual exponent} = 254 - 127 = +127\]
  – Fraction: 111...11 \[\Rightarrow\text{significand} \approx 2.0\]
  – \[±2.0 \times 2^{+127} \approx ±3.4 \times 10^{+38}\]
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
  – ±1.0 × 2⁻¹⁰²² ≈ ±2.2 × 10⁻³⁰⁸

• Largest value
  – Exponent: 11111111110
    ⇒ actual exponent = 2046 – 1023 = +1023
  – Fraction: 111...11 ⇒ significand ≈ 2.0
  – ±2.0 × 2⁺¹⁰²³ ≈ ±1.8 × 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 = $1000...00_2$
  
  Exponent = $-1 + \text{Bias}$
  
  • Single: $-1 + 127 = 126 = 01111110_2$
  
  • Double: $-1 + 1023 = 1022 = 01111111110_2$

• Single: $1011111101000...00$

• Double: $10111111111101000...00$
Floating-Point Example

• What number is represented by the single-precision float

\[11000000101000...00\]

  – \( S = 1 \)
  – Fraction = \( 01000...00_2 \)
  – 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
\[ 1/0 = \text{infinity} \]
any positive value / 0 = positive infinity
any negative value / 0 = negative infinity
\[ \text{infinity} \times x = \text{infinity} \]
\[ 1/\text{infinity} = 0 \]

\[ 0/0 = \text{NaN} \]
\[ 0 \times \text{infinity} = \text{NaN} \]
\[ \text{infinity} \times \text{infinity} = \text{NaN} \]
\[ \text{infinity} - \text{infinity} = \text{NaN} \]
\[ \text{infinity}/\text{infinity} = \text{NaN} \]

\[ x + \text{NaN} = \text{NaN} \]
\[ \text{NaN} + x = \text{NaN} \]
\[ x - \text{NaN} = \text{NaN} \]
\[ \text{NaN} - x = \text{NaN} \]
\[ \sqrt{\text{negative value}} = \text{NaN} \]
\[ \text{NaN} \times x = \text{NaN} \]
\[ \text{NaN} \times 0 = \text{NaN} \]
\[ 1/\text{NaN} = \text{NaN} \]

(Any operation on a NaN produces a NaN.)
FIGURE 3.14 IEEE 754 encoding of floating-point numbers. A separate sign bit determines the sign. Denormalized numbers are described in the *Elaboration* on page 270. This information is also found in Column 4 of the MIPS Reference Data Card at the front of this book. Copyright © 2009 Elsevier, Inc. All rights reserved.

<table>
<thead>
<tr>
<th>Single precision</th>
<th>Double precision</th>
<th>Object represented</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponent</td>
<td>Exponent</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>Nonzero</td>
<td>± denormalized number</td>
</tr>
<tr>
<td>1–254</td>
<td>Anything</td>
<td>± floating-point number</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>± infinity</td>
</tr>
<tr>
<td>255</td>
<td>Nonzero</td>
<td>NaN (Not a Number)</td>
</tr>
</tbody>
</table>
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
  – $1.000_2 \times 2^{-1} + \neg1.110_2 \times 2^{-2}$ (0.5 + –0.4375)
• 1. Align binary points
  – Shift number with smaller exponent
  – $1.000_2 \times 2^{-1} + \neg0.111_2 \times 2^{-1}$
• 2. Add significands
  – $1.000_2 \times 2^{-1} + \neg0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
• 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
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
  – $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^5$
• 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 × −ve ⇒ −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 Target Label
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:

```mips
f2c:   lw $f16, 0($gp)
lw $f18, 4($gp)
div.s $f16, $f16, $f18
lw $f18, 8($gp)
sub.s $f18, $f12, $f18
mul.s $f0, $f16, $f18
jr $ra
```
FP Example: Array Multiplication

- \( X = X + Y \times Z \)
  - All 32 \( \times \) 32 matrices, 64-bit double-precision elements
- C code:
  ```c
  void mm ( double x[][[]],
            double y[][[]], double z[][[]]) {
    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>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>li $t1, 32</td>
<td># $t1 = 32 (row size/loop end)</td>
<td></td>
</tr>
<tr>
<td>li $s0, 0</td>
<td># i = 0; initialize 1st for loop</td>
<td></td>
</tr>
<tr>
<td>L1: li $s1, 0</td>
<td># j = 0; restart 2nd for loop</td>
<td></td>
</tr>
<tr>
<td>L2: li $s2, 0</td>
<td># k = 0; restart 3rd for loop</td>
<td></td>
</tr>
<tr>
<td>sll $t2, $s0, 5</td>
<td># $t2 = i * 32 (size of row of x)</td>
<td></td>
</tr>
<tr>
<td>addu $t2, $t2, $s1</td>
<td># $t2 = i * size(row) + j</td>
<td></td>
</tr>
<tr>
<td>sll $t2, $t2, 3</td>
<td># $t2 = byte offset of [i][j]</td>
<td></td>
</tr>
<tr>
<td>addu $t2, $a0, $t2</td>
<td># $t2 = byte address of x[i][j]</td>
<td></td>
</tr>
<tr>
<td>l.d $f4, 0($t2)</td>
<td># $f4 = 8 bytes of x[i][j]</td>
<td></td>
</tr>
<tr>
<td>L3: sll $t0, $s2, 5</td>
<td># $t0 = k * 32 (size of row of z)</td>
<td></td>
</tr>
<tr>
<td>addu $t0, $t0, $s1</td>
<td># $t0 = k * size(row) + j</td>
<td></td>
</tr>
<tr>
<td>sll $t0, $t0, 3</td>
<td># $t0 = byte offset of [k][j]</td>
<td></td>
</tr>
<tr>
<td>addu $t0, $a2, $t0</td>
<td># $t0 = byte address of z[k][j]</td>
<td></td>
</tr>
<tr>
<td>l.d $f16, 0($t0)</td>
<td># $f16 = 8 bytes of z[k][j]</td>
<td></td>
</tr>
</tbody>
</table>

...
**FP Example: Array Multiplication**

```
sl $t0, $s0, 5       # $t0 = i*32 (size of row of y)
addu $t0, $t0, $s2    # $t0 = i*size(row) + k
sl $t0, $t0, 3       # $t0 = byte offset of y[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

The BIG Picture

• 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
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*
Numerical Catastrophes

Ariane 5 rocket. [June 4, 1996]
- 10 year, $7 billion ESA project exploded after launch.
- 64 bit float converted to 16 bit signed int.
- Unanticipated overflow.

Vancouver stock exchange. [November, 1983]
- Index undervalued by 44%.
- Recalculated index after each trade by adding change in price.
- 22 months of accumulated truncation error.

Patriot missile accident. [February 25, 1991]
- Failed to track scud; hit Army barracks, killed 28.
- Inaccuracy in measuring time in 1/20 of a second since using 24 bit binary floating point.

After Sedgewick and Wayne
FIGURE 3.23 A sampling of newspaper and magazine articles from November 1994, including the New York Times, San Jose Mercury News, San Francisco Chronicle, and InfoWorld. The Pentium floating-point divide bug even made the “Top 10 List” of the David Letterman Late Show on television. Intel eventually took a $300 million write-off to replace the buggy chips. Copyright © 2009 Elsevier, Inc. All rights reserved.