Chapter 3 & Appendix B (continued)

Arithmetic for Computers

Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - -2.34×10^{56} ← normalized
  - $+0.002 \times 10^{-4}$ ← not normalized
  - $+987.02 \times 10^{9}$ ← not normalized
- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types `float` and `double` in C

§3.5 Floating Point

Floating Point Standard

- Defined by IEEE standard 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

	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction

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

- S: sign bit (0 $\Rightarrow$ non-negative, 1 $\Rightarrow$ negative)
- Normalized significand: 1.0 ≤ |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: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1023

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    $\Rightarrow$ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    $\Rightarrow$ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ 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 $23 \times \log_{10}2 \approx 23 \times 0.3$ ≈ 6 decimal digits of precision
  - Double: approx 2^{-52}
    - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3$ ≈ 16 decimal digits of precision

Binary Refresher

- When we look at a number like 10110_2, we're seeing it as:

  $1(2^4) + 0(2^3) + 1(2^2) + 1(2^1) + 0(2^0)$

  $= 16 + 4 + 2 = 22_{10}$

Binary Decimal Points

- In decimal, 12.63 is the same as

  $1(10^1) + 2(10^0) + 6(10^{-1}) + 3(10^{-2})$

- In binary, 101.01_2 is the same as

  $1(2^2) + 0(2^1) + 1(2^0) + 0(2^{-1}) + 1(2^{-2})$

  $= 4 + 1 + 0.25 = 5.25$

Useful Exponent Identities

- $a^b * a^c = a^{b+c}$
  - Why memorize more than 2^{10} when we can just break them down?
  - $2^{35} = 2^5 * 2^{30}$
    $= 2^5 * 2^{10} * 2^{10} * 2^{10} = 32$ GB
- $a^{-b} = \frac{1}{a^b}$
  - When we have decimal terms, instead of seeing 2^{-1}, 2^{-2}, etc. use $\frac{1}{2^1}, \frac{1}{2^2}$, etc.

More Binary

- Dividing (shifting right) by 2_{10} is the same as moving the decimal point one place to the left.
  - Same reasoning as when we divide by 10 in base 10
- Multiplying (shifting left) works the same way

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: $-1 + 1023 = 1022 = 01111111110_2$
- Single: 1011111101000…00
- Double: 1011111111101000…00

Floating-Point Example

- What number is represented by the single-precision float
  11000000101000…00
  - S = 1
  - Fraction = $01000\ldots00_2$
  - Exponent = $10000001_2 = 129$
- $x = (-1)^1 \times (1 + 0.01_2) \times 2^{(129 - 127)}$
  $= (-1) \times 1.25 \times 2^2$
  $= -5.0$

IEEE 754-1985 Specials

- We reserve all 0s and all 1s in the exponent. This is why:
- 011111111110000...00 = +∞
- 111111111110000...00 = -∞
- X11111111[non-zero] = NaN
  - e.g., square root of a negative number
- X000000000000...00 = 0
  - ...there's actually a positive zero and a negative zero

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
  - 1.0015×10^{2}
- 4. Round and renormalize if necessary
  - 1.002×10^{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 Instructions in MIPS

- FP hardware is coprocessor 1
  - Adjunct processor that extends the ISA
- Separate FP registers
  - 32 single-precision: $f0, $f1, … $f31
  - Paired for double-precision: $f0/$f1, $f2/$f3, …
    - Release 2 of MIPS ISA supports 32 × 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:

```
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:

```
f2c: lwc1   $f16, const5($gp)
     lwc1   $f18, const9($gp)
     div.s  $f16, $f16, $f18
     lwc1   $f18, const32($gp)
     sub.s  $f18, $f12, $f18
     mul.s  $f0,  $f16, $f18
     jr     $ra
```

FP Example: Array Multiplication

- $X = X + Y \times Z$ (initialize X to all zeros...)
  - All 32 × 32 matrices, 64-bit double-precision elements
- C code:

```
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:

```
     li   $t1, 32        # $t1 = 32 (row size/loop end)
     li   $s0, 0         # i = 0; initialize 1st for loop
L1:  li   $s1, 0         # j = 0; restart 2nd for loop
L2:  li   $s2, 0         # k = 0; restart 3rd for loop
     sll  $t2, $s0, 5    # $t2 = i * 32 (size of row of x)
     addu $t2, $t2, $s1  # $t2 = i * size(row) + j
     sll  $t2, $t2, 3    # $t2 = byte offset of [i][j]
     addu $t2, $a0, $t2  # $t2 = byte address of x[i][j]
     l.d  $f4, 0($t2)    # $f4 = 8 bytes of x[i][j]
L3:  sll  $t0, $s2, 5    # $t0 = k * 32 (size of row of z)
     addu $t0, $t0, $s1  # $t0 = k * size(row) + j
     sll  $t0, $t0, 3    # $t0 = byte offset of [k][j]
     addu $t0, $a2, $t0  # $t0 = byte address of z[k][j]
     l.d  $f16, 0($t0)   # $f16 = 8 bytes of z[k][j]
```

…

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

Associativity

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

		(x+y)+z	x+(y+z)
x	-1.50E+38		-1.50E+38
y	1.50E+38	0.00E+00	
z	1.0	1.0	1.50E+38
		1.00E+00	0.00E+00

- 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*