Computer Arithmetic: Floating Points, Subword Parallelism

Dr. Vincent C. Emeakaroha

08-02-2017

vc.emeakaroha@cs.ucc.ie

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 = $01000\ldots00_2$
  - Fxponent = 10000001_2 = 129
- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

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

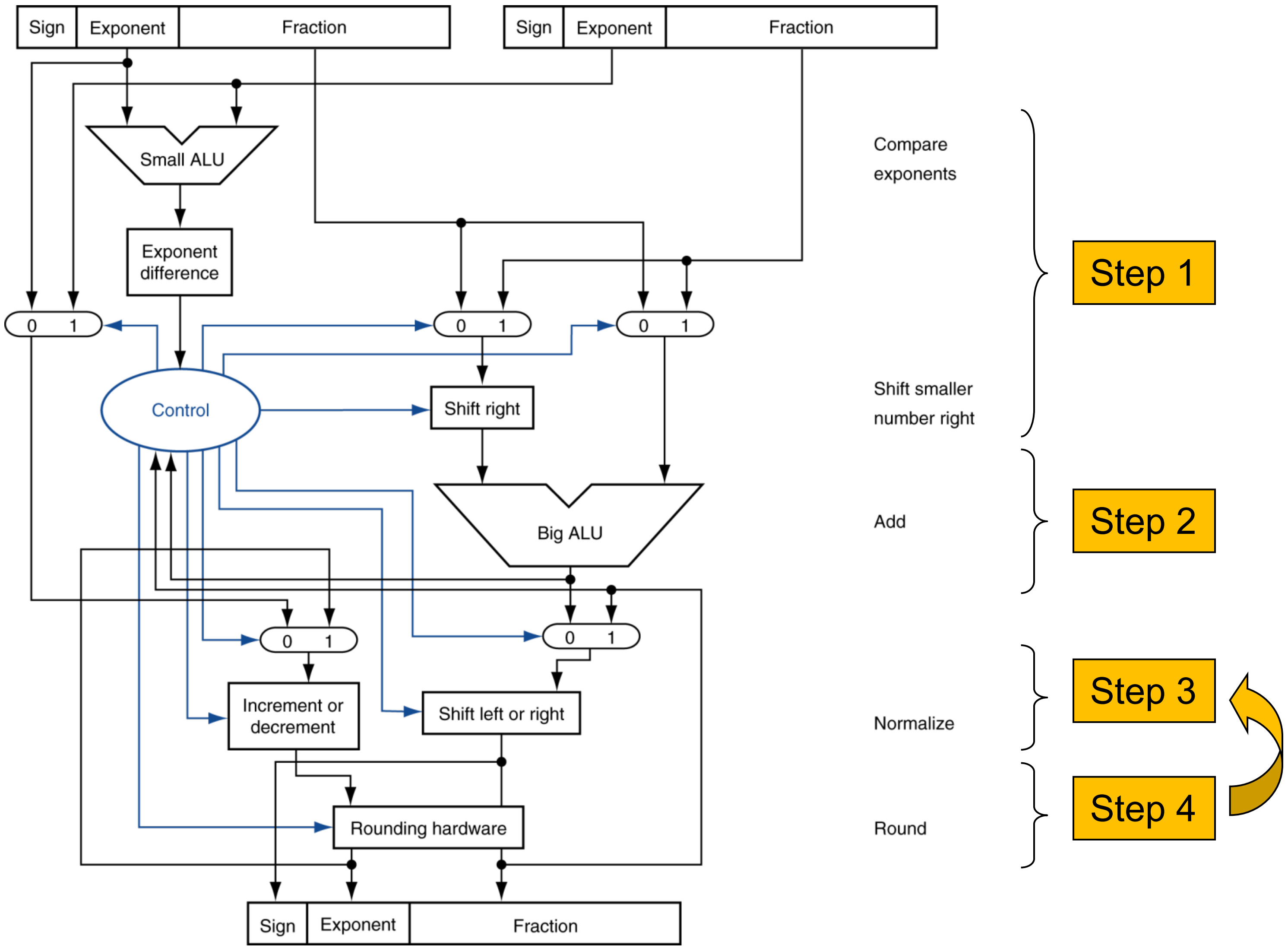
Floating-Point Addition

- Now consider a 4-digit binary example
  - Add 0.5_{10} and -0.4375_{10}
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$
- 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) $= 0.0625_{10}$

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



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
  - 1.0212×10^6
- 4. Round and renormalize if necessary
  - 1.021×10^6
- 5. Determine sign of result from signs of operands
  - $+1.021 \times 10^6$

Floating-Point Multiplication

- Now consider a 4-digit binary example
  - Multiply 0.5_{10} and -0.4375_{10}
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$
- 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 $\Rightarrow$ –ve
  - $-1.110_2 \times 2^{-3} = -0.21875_{10}$

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 $\leftrightarrow$ integer conversion
- Operations usually takes several cycles
  - Can be pipelined

FP Instructions in MIPS

- 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)
     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
```

FP Example: Array Multiplication

- $X = X + Y \times Z$
  - 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
- Computer arithmetic is almost always an approximation of real number
- Different between computer number and number in real world
  - Computer numbers have limited size and hence limited precision
  - Programmers must remember these limits and write programs accordingly