Chapter 3

Arithmetic for Computers
Arithmetic for Computers

- Operations on integers
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow

- Floating-point real numbers
  - Representation and operations
Integer Addition

- Example: 7 + 6

```
\[ \begin{array}{cccccccc}
(0) & (0) & (1) & (1) & (0) & (Carries) \\
\ldots & 0 & 0 & 0 & 1 & 1 & 1 \\
\ldots & 0 & 0 & 0 & 1 & 1 & 0 \\
\ldots (0) & 0 & (0) & 0 & 1 & (1) & 0 & (0) & 1 \\
\end{array} \]
```

- Overflow if result out of range
  - Adding +ve and –ve operands, no overflow
  - Adding two +ve operands
    - Overflow if result sign is 1
  - Adding two –ve operands
    - Overflow if result sign is 0
Integer Subtraction

- Add negation of second operand
- Example: \( 7 - 6 = 7 + (-6) \)
  
  \[
  \begin{array}{c}
  +7: & 0000\,0000\,\ldots\,0000\,0111 \\
  -6: & 1111\,1111\,\ldots\,1111\,1010 \\
  +1: & 0000\,0000\,\ldots\,0000\,0001 \\
  \end{array}
  \]

- Overflow if result out of range
  - Subtracting two +ve or two –ve operands, no overflow
  - Subtracting +ve from –ve operand
    - Overflow if result sign is 0
  - Subtracting –ve from +ve operand
    - Overflow if result sign is 1
Dealing with Overflow

- Some languages (e.g., C) ignore overflow
  - Use MIPS addu, addui, subu instructions

- Other languages (e.g., Ada, Fortran) require raising an exception
  - Use MIPS add, addi, sub instructions
  - On overflow, invoke exception handler
    - Save PC in exception program counter (EPC) register
    - Jump to predefined handler address
    - mfc0 (move from coprocessor reg) instruction can retrieve EPC value, to return after corrective action
Arithmetic for Multimedia

- Graphics and media processing operates on vectors of 8-bit and 16-bit data
  - Use 64-bit adder, with partitioned carry chain
    - Operate on 8×8-bit, 4×16-bit, or 2×32-bit vectors
  - SIMD (single-instruction, multiple-data)

- Saturating operations
  - On overflow, result is largest representable value
    - c.f. 2s-complement modulo arithmetic
  - E.g., clipping in audio, saturation in video
Multiplication

• Start with long-multiplication approach

 multiplicand
  
  1000
  × 1001

  1000
  0000
  0000
  1000

  1001000

multiplier

product

Length of product is the sum of operand lengths

Multiplicand

Shift left

64 bits

Multiplier

Shift right

32 bits

Product

Write

64 bits

Control test

64-bit ALU
Multiplication Hardware

1. Test Multiplier0
   - Multiplier0 = 1
     1a. Add multiplicand to product and place the result in Product register
   - Multiplier0 = 0

2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit

32nd repetition?
   - No: < 32 repetitions
   - Yes: 32 repetitions

Done

64-bit ALU

Control test

Initially 0

Multiplicand

Shift left

64 bits

Multiplier

Shift right

32 bits

Product

Write

64 bits
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**
  - `mult rs, rt` / `multu rs, rt`
    - 64-bit product in HI/LO
  - `mfhi rd` / `mflo rd`
    - Move from HI/LO to rd
    - Can test HI value to see if product overflows 32 bits
  - `mul rd, rs, rt`
    - Least-significant 32 bits of product -> 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

\[
\begin{array}{c}
1000 \underline{1001010} \\
-1000 \\
\hline
1010 \\
-1000 \\
\hline
10
\end{array}
\]

\(1001\) divided by \(1000\), with a remainder of \(10\).
Division Hardware

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

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

2.b. 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.

Remainder ≥ 0 → Test Remainder

Remainder < 0 →

No: < 33 repetitions

Yes: 33 repetitions

33rd repetition?

Done

Initially divisor in left half

Initially dividend

Divisor

Shift right

64 bits

64-bit ALU

Remainder

Write

64 bits

Control test

Quotient

Shift left

32 bits
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

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

- Like scientific notation
  - $-2.34 \times 10^{56}$
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^9$

- In binary
  - $\pm1.xxxxxxxxx_2 \times 2^{yyyy}$

- 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

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

- **Exponents 00000000 and 11111111 reserved**

- **Smallest value**
  - Exponent: 00000001
    - $\Rightarrow$ 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 $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ 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 = \(-1 + \text{Bias}\)
    - Single: \(-1 + 127 = 126 = 01111110_2\)
    - Double: \(-1 + 1023 = 1022 = 0111111110_2\)

- **Single:** \(101111101000\ldots00\)
- **Double:** \(101111111101000\ldots00\)
Floating-Point Example

- What number is represented by the single-precision float
  \[ 11000000101000\ldots00 \]
  - \( S = 1 \)
  - Fraction = \( 01000\ldots00 \)
  - 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

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

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

- Denormal with fraction = 000...0

\[ x = (-1)^S \times (0 + 0) \times 2^{1-\text{Bias}} = \pm0.0 \]

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

Step 2
- Shift smaller number right
- Add

Step 3
- Normalize

Step 4
- Round
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 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:**
  
  ```mips
  f2c:  lwcl  $f16, const5($gp)
  lwcl2 $f18, const9($gp)
  div.s $f16, $f16, $f18
  lwcl1 $f18, const32($gp)
  sub.s $f18, $f12, $f18
  mul.s $f0,  $f16, $f18
  jr     $ra
  ```

Chapter 3 — Arithmetic for Computers — 36
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[][[]]) {
      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  # $4=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>1.0</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>FILD mem/ST(i)</td>
<td>FIADDP 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>FIMULP 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></td>
<td>FPSREM</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 × 128-bit registers
  - Extended to 8 registers in AMD64/EM64T
- Can be used for multiple FP operands
  - 2 × 64-bit double precision
  - 4 × 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., $-5 / 4$
    - $11111011_2 >> 2 = 11111110_2 = -2$
    - Rounds toward $-\infty$
  - c.f. $11111011_2 >>> 2 = 00111110_2 = +62$
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