Chapter 3

Floating Point Arithmetic

Review - Multiplication

0 1 1 0 = 6

0 0 0 0 0 1 0 0 = 5
add 0 1 1 0 0 1 0 1
0 0 1 1 0 0 1 0
add 0 0 1 1 0 0 1 0
0 0 0 1 1 0 0 1
add 0 1 1 1 1 0 0 1
0 0 1 1 1 1 0 0
add 0 1 1 1 1 1 0 0
0 0 0 1 1 1 1 0 = 30
**Review - Division**

- **0 0 1 0** = 2
- **32-bit ALU**
- **divisor**
- **dividend**
- **remainder**
- **quotient**
- **subtract**
- **shift left**
- **Control**

\[
\begin{align*}
0 0 0 0 & \quad 0 1 1 0 = 6 \\
0 0 0 0 & \quad 1 1 0 0 \\
\text{sub} & \quad 1 1 1 0 \quad 1 1 0 0 \\
0 0 0 0 & \quad 1 1 0 0 \\
0 0 0 1 & \quad 1 0 0 0 \\
\text{sub} & \quad 1 1 1 1 \quad 1 1 0 0 \\
0 0 0 1 & \quad 1 0 0 0 \\
0 0 1 1 & \quad 0 0 0 0 \\
\text{sub} & \quad 0 0 0 1 \quad 0 0 0 0 \\
0 0 1 0 & \quad 0 0 1 0 \\
\text{sub} & \quad 0 0 0 0 \quad 0 0 1 1 \\
0 0 1 0 & \quad 0 0 1 0 \\
\end{align*}
\]

- rem neg, so 'ient bit = 0
- restore remainder
- rem neg, so 'ient bit = 0
- restore remainder
- rem pos, so 'ient bit = 1
- rem pos, so 'ient bit = 1
- = 3 with 0 remainder

**Floating Point**

- What can be represented in N bits?
  - Unsigned: 0 to \(2^{N-1}\)
  - 2’s Complement: \(-2^{N-1}\) to \(2^{N-1} - 1\)
- But, what about--
  - Very large numbers:
    - 9,349,398,989,787,762,244,859,087,678
    - 1.23 \times 10^{67}
  - Very small numbers:
    - 0.000000000000000000000000045691
    - 2.98 \times 10^{-32}
- Fractional values? 0.35
- Mixed numbers? 10.28
- Irrationals? \(\pi\)
Floating Point

- The essential idea of floating point representation is that a fixed number of bits are used (usually 32 or 64) and that the binary point "floats" to where it is needed. Some of the bits of a floating point representation must be used to say where the binary point lies. The programmer does not need to explicitly keep track of it.
- IEEE (Institute of Electrical and Electronics Engineers) created a standard for floating point. This is the IEEE 754 standard, released in 1985 and since updated in 2008. All "main stream" hardware and software follows this standard.

Floating Point

- Floating Point provides representation for non-integral numbers.
- Like scientific notation, we need to "normalize"
  - $-2.34 \times 10^{56}$
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^9$
- In binary
  - $\pm1.xxxxxxx_2 \times 2^{yyyy}$
- This is the representation for Types `float` and `double` in C.
Sign and Magnitude Representation

<table>
<thead>
<tr>
<th>Sign (1 bit)</th>
<th>Exponent (8 bits)</th>
<th>Fraction (23 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>E</td>
<td>F</td>
</tr>
</tbody>
</table>

- More exponent bits → wider range of numbers
- More fraction bits → higher precision

- For normalized numbers, we are guaranteed that the number is of the form 1.xxxx….
  Hence, in IEEE 754 standard, the 1 is implicit.

IEEE Floating-Point Format

single: 8 bits  double: 11 bits
single: 23 bits  double: 52 bits

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

- 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. This bit is referred to as the “hidden” bit.
  - Significand is Fraction with the “1.” restored.
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned.
  - Single: Bias = 127; Double: Bias = 1023
Single-Precision Range

- Exponents 00000000 and 11111111 are reserved for exceptions.
- 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
  - ±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 Example

- What number is represented by the single-precision float 11000000101000...00?
  - S = 1
  - Exponent = 10000001_2 = 129
  - Fraction = 01000...00_2
  - x = (-1)^1 × (1 + 01_2) × 2^{129 - 127}
    = (-1) × 1.25 × 2^2
    = -5.0

Floating-Point Example

- What is the FP bit representation for -0.75_{10}?
  - Represent -0.75
    - -0.75 = (-1)^1 × 1.1_2 × 2^{-1}
    - S = 1
    - Fraction = 1000...00_2
    - Exponent = -1 + Bias
      - Single: -1 + 127 = 126 = 01111110_2
      - Double: -1 + 1023 = 1022 = 0111111110_2
  - Single: 1011111101000...00
  - Double: 1011111111101000...00
Code Example – Degree Conversion

- MIPS has a second set of 32 32-bit registers reserved for floating point operations.

```c
float f2c (float fahr)
{
    return ((5.0/9.0) * (fahr - 32.0));
}
```

(argument fahr is stored in $f12)
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

Floating-Point Addition

- Addition (and subtraction)

\[(\pm F_1 \times 2^{E_1}) + (\pm F_2 \times 2^{E_2}) = \pm F_3 \times 2^{E_3}\]

- Step 0: Restore the hidden bit in F1 and in F2.
- Step 1: Align fractions by right shifting F2 by \(E_1 - E_2\) positions (assuming \(E_1 \geq E_2\)) keeping track of the lower (or higher) order bits shifted out.
- Step 2: Add the resulting F2 to F1 to form F3.
- Step 3: Normalize F3 (so it is in the form 1.XXXXX …)
  - If F1 and F2 have the same sign, shift F3 and increment E3 (check for overflow).
  - If F1 and F2 have different signs, F3 may require many left shifts each time decrementing E3 (check for underflow).
- Step 4: Round F3 and possibly normalize again.
- Step 5: Rehide the most significant bit of F3 before storing the result.
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
  - $1.002 \times 10^2$

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 × 10^{10} × 9.200 × 10^{-5}
- 1. Add exponents
  - For biased exponents, subtract bias from sum.
  - New exponent = 10 + -5 = 5
- 2. Multiply significands
  - 1.110 × 9.200 = 10.212 \Rightarrow 10.212 × 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}
Denormal Numbers

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

- Denormal numbers are smaller than normal numbers.
- Denormal with fraction = 000...0
  \[ x = (-1)^S \times (0 + 0) \times 2^{-\text{Bias}} = \pm 0.0 \]

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.
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.
- Rounding (except for truncation) requires the hardware to include extra bits during calculations
  - Guard bit – used to provide one bit when shifting left to normalize a result (e.g., when normalizing after division or subtraction).
  - Round bit – used to improve rounding accuracy.
  - Sticky bit – used to support \textit{Round to nearest even}; is set to a 1 whenever a 1 bit shifts (right) through it (e.g., when aligning during addition/subtraction).

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>F:COMP</td>
<td>FPATAN</td>
</tr>
<tr>
<td>FISTP mem/ST(i)</td>
<td>FISUBRP mem/ST(i)</td>
<td>F:UCOMP</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>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.
Subword Parallelism

- ALUs are typically designed to perform 64-bit or 128-bit arithmetic.

- Some data types are much smaller, e.g., bytes for pixel RGB values, half-words for audio samples.

- Partitioning the carry-chains within the ALU can convert the 64-bit adder into 4 16-bit adders or 8 8-bit adders.

- A single load can fetch multiple values, and a single add instruction can perform multiple parallel additions, referred to as subword parallelism.

Concluding Remarks

- 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.

- ISA’s support arithmetic
  - Signed and unsigned integers.
  - Floating-point approximation to real numbers.

- Bounded range and precision
  - Operations can overflow and underflow.

- MIPS ISA strongly supports IEEE-754.