Floating Point Numbers

• We need a way to represent a wide range of numbers
  • numbers with fractions, e.g., 3.1416
  • large number:
    \[ 976,000,000,000,000 = 9.76 \times 10^{14} \]
  • small number:
    \[ 0.0000000000000976 = 9.76 \times 10^{-14} \]

• Representation:
  • sign, exponent, significand:
    \[ (-1)^{\text{sign}} \times \text{significand} \times 2^{\text{exponent}} \]
  • more bits for significand gives more accuracy
  • more bits for exponent increases range
Scientific Notation

- **Scientific notation:**
  - \(0.525 \times 10^5 = 5.25 \times 10^4 = 52.5 \times 10^3\)
  - \(5.25 \times 10^4\) is in *normalized* scientific notation.
    - position of decimal point fixed
    - leading digit non-zero

- **Binary numbers**
  - \(5.25 = 101.01 = 1.0101 \times 2^2\)
  - Binary point
    - multiplication by 2 moves the point to the left
    - division by 2 moves the point to the right
  - Known as *floating point format*.
## Binary to Decimal Conversion

<table>
<thead>
<tr>
<th>Conversion Type</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Binary</strong></td>
<td>((-1)^S (1 \cdot b_1 b_2 b_3 b_4) \times 2^E)</td>
</tr>
<tr>
<td><strong>Decimal</strong></td>
<td>((-1)^S \times (1 + b_1 \times 2^{-1} + b_2 \times 2^{-2} + b_3 \times 2^{-3} + b_4 \times 2^{-4}) \times 2^E)</td>
</tr>
</tbody>
</table>

### Example:

\(-1.1100 \times 2^{-2}\) (binary) = \(- (1 + 2^{-1} + 2^{-2}) \times 2^{-2}\)

= \(- (1 + 0.5 + 0.25)/4\)

= \(- 1.75/4\)

= \(- 0.4375\) (decimal)
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 Std. 754 Floating-Point Format

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

**Single-Precision**

- **S**: Sign bit (0 for positive, 1 for negative)
- **E**: 8-bit Exponent (Biased by 127)
- **F**: 23-bit Fraction

**Double-Precision**

- **S**: Sign bit (0 for positive, 1 for negative)
- **E**: 11-bit Exponent (Biased by 1023)
- **F**: 52-bit Fraction + Continuation of 52-bit Fraction (bits 31-0)

**Bits Mapping**

- Bits 30-23: Single-Precision Fraction
- Bits 22-0: Single-Precision Fraction (Continued)
- Bits 30-20: Double-Precision Fraction
- Bits 19-0: Double-Precision Fraction (Continued)
- Bit 31: Sign Bit
IEEE 754 floating-point standard

- Represented value = $(-1)^{\text{sign}} \times (1+F) \times 2^{\text{exponent} - \text{bias}}$

- Exponent is “biased” (excess-K format) to make sorting easier
  - bias of 127 for single precision and 1023 for double precision
  - E values in $[1 .. 254]$ (0 and 255 reserved)
  - Range = $2^{-126}$ to $2^{+127}$ ($10^{38}$ to $10^{+38}$)

- Significand in sign-magnitude, normalized form
  - Significand = $(1 + F) = 1.\ b_1b_2b_3...b_{23}$
  - Suppress storage of leading 1

- Overflow: Exponent requiring more than 8 bits. Number can be positive or negative.

- Underflow: Fraction requiring more than 23 bits. Number can be positive or negative.
IEEE 754 floating-point standard

- Example:
  - Decimal: $-5.75 = -\left( 4 + 1 + \frac{1}{2} + \frac{1}{4} \right)$
  - Binary: $-101.11 = -1.0111 \times 2^2$
  - Floating point: exponent $= 129 = 10000001$

- IEEE single precision:
  $11000000010111000000000000000000$
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: $1011111101000\ldots00$
- Double: $1011111111101000\ldots00$
Floating-Point Example

- What number is represented by the single-precision float
  \[ 110000000101000\ldots00 \]
  - S = 1
  - Fraction = 01000\ldots00_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
Examples

Biased exponent (0-255), bias 127 (01111111) to be subtracted

\[ 1.1010001 \times 2^{10100} = 0 \ 10010011 \ 101001000000000000000000 = 1.6328125 \times 2^{20} \]

\[ -1.1010001 \times 2^{10100} = 1 \ 10010011 \ 101001000000000000000000 = -1.6328125 \times 2^{20} \]

\[ 1.1010001 \times 2^{-10100} = 0 \ 01101011 \ 101001000000000000000000 = 1.6328125 \times 2^{-20} \]

\[ -1.1010001 \times 2^{-10100} = 1 \ 01101011 \ 101001000000000000000000 = -1.6328125 \times 2^{-20} \]

0.5
0.125
0.0078125
0.6328125
Numbers in 32-bit Formats

• Two’s complement integers

• Floating point numbers


ELEC 5200/6200 - From P-H slides
IEEE 754 Special Codes

Zero: $S \ 00000000 \ 00000000000000000000000000000000$

- \pm 1.0 \times 2^{-127}
- Smallest positive number in single-precision IEEE 754 standard.
- Interpreted as positive/negative zero.
- Exponent less than -127 is positive underflow (regard as zero).

Infinity: $S \ 11111111 \ 00000000000000000000000000000000$

- \pm 1.0 \times 2^{128}
- Largest positive number in single-precision IEEE 754 standard.
- Interpreted as \pm \infty
- If true exponent = 128 and fraction \neq 0, then the number is greater than \infty. It is called “not a number” or NaN (interpret as \infty).
Addition and Subtraction

- Addition/subtraction of two floating-point numbers:

Example: \(2 \times 10^3\)  
Align mantissas: \(0.2 \times 10^4\)  
\[+ 3 \times 10^4\]  
\[+ 3 \times 10^4\]  
\[3.2 \times 10^4\]

- General Case:

\[m_1 \times 2^{e_1} \pm m_2 \times 2^{e_2} = (m_1 \pm m_2 \times 2^{e_2-e_1}) \times 2^{e_1} \text{ for } e_1 > e_2\]

\[= (m_1 \times 2^{e_1-e_2} \pm m_2) \times 2^{e_2} \text{ for } e_2 > e_1\]

- Shift smaller mantissa right by \(|e_1 - e_2|\) bits to align the mantissas.
Addition/Subtraction Algorithm

0. Zero check
   - Change the sign of subtrahend
   - If either operand is 0, the other is the result

1. Significand alignment: right shift smaller significand until two exponents are identical.

2. Addition: add significands and report exception if overflow occurs.

3. Normalization
   - Shift significand bits to normalize.
   - Report overflow or underflow if exponent goes out of range.

4. Rounding
FP Add/Subtract  (PH Text Figs. 3.16/17)
Example

- Subtraction: $0.5_{\text{ten}} - 0.4375_{\text{ten}}$

- Step 0: Floating point numbers to be added
  
  $1.000_{\text{two}} \times 2^{-1}$ and $-1.110_{\text{two}} \times 2^{-2}$

- Step 1: Significand of lesser exponent shifted right until exponents match
  
  $-1.110_{\text{two}} \times 2^{-2} \rightarrow -0.111_{\text{two}} \times 2^{-1}$

- Step 2: Subtract significands, $1.000_{\text{two}} + (-0.111_{\text{two}})$

  Result is $0.001_{\text{two}} \times 2^{-1}$

- Step 3: Normalize, $1.000_{\text{two}} \times 2^{-4}$

  No overflow/underflow since $127 \geq \text{exponent} \geq -126$

- Step 4: Rounding, no change since the sum fits in 4 bits.

  $1.000_{\text{two}} \times 2^{-4} = (1+0)/16 = 0.0625_{\text{ten}}$
FP Multiplication: Basic Idea

\[(m_1 \times 2^{e_1}) \times (m_2 \times 2^{e_2}) = (m_1 \times m_2) \times 2^{e_1+e_2}\]

- Separate signs
- Add exponents
- Multiply significands
- Normalize, round, check overflow
- Replace sign
FP Mult. Illustration

• Multiply \(0.5_{\text{ten}}\) and \(-0.4375_{\text{ten}}\) (answer = \(-0.21875_{\text{ten}}\)) or
• Multiply \(1.000_{\text{two}} \times 2^{-1}\) and \(-1.110_{\text{two}} \times 2^{-2}\)
• Step 1: Add exponents

\[-1 + (-2) = -3\]

• Step 2: Multiply significands

```
  1.000
x 1.110
   0000
   1000
   1000
   1000
  1110000 Product is 1.110000
```

* Step 3: Normalization: If necessary, shift significand right and increment exponent.

Normalized product is \(1.110000 \times 2^{-3}\)
Check overflow/underflow: \(127 \geq \text{exponent} \geq -126\)

* Step 4: Rounding: \(1.110 \times 2^{-3}\)

* Step 5: Sign: Operands have opposite signs, Product is \(-1.110 \times 2^{-3}\)

Decimal value = \(- (1+0.5+0.25)/8 = -0.21875_{\text{ten}}\)
FP Division: Basic Idea

- Separate sign.
- Check for zeros and infinity.
- Subtract exponents.
- Divide significands.
- Normalize/overflow/underflow.
- Rounding.
- Replace sign.
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
  - \texttt{lwc1, ldc1, swc1, sdc1}
    - e.g., \texttt{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:  lw   $f16, const5($gp)
        lw   $f18, const9($gp)
        div.s $f16, $f16, $f18
        lw   $f18, const32($gp)
        sub.s $f18, $f12, $f18
        mul.s $f0, $f16, $f18
        jr   $ra
  ```
### MIPS Floating Point Instructions

#### MIPS floating-point operands

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 floating-point registers</td>
<td>$f0, f1, f2, ..., f31</td>
<td>MIPS floating-point registers are used in pairs for double precision numbers.</td>
</tr>
<tr>
<td>$2^{30}$ memory words</td>
<td>Memory[0], Memory[4], ..., Memory[4294967292]</td>
<td>Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.</td>
</tr>
</tbody>
</table>

#### MIPS floating-point assembly language

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>Add single</td>
<td>add.s $f2,$f4,$f6</td>
<td>$f2 = f4 + f6</td>
<td>FP add (single precision)</td>
</tr>
<tr>
<td></td>
<td>Subtract single</td>
<td>sub.s $f2,$f4,$f6</td>
<td>$f2 = f4 - f6</td>
<td>FP sub (single precision)</td>
</tr>
<tr>
<td></td>
<td>Multiply single</td>
<td>mul.s $f2,$f4,$f6</td>
<td>$f2 = f4 \times f6</td>
<td>FP multiply (single precision)</td>
</tr>
<tr>
<td></td>
<td>Divide single</td>
<td>div.s $f2,$f4,$f6</td>
<td>$f2 = f4 / f6</td>
<td>FP divide (single precision)</td>
</tr>
<tr>
<td></td>
<td>Add double</td>
<td>add.d $f2,$f4,$f6</td>
<td>$f2 = f4 + f6</td>
<td>FP add (double precision)</td>
</tr>
<tr>
<td></td>
<td>Subtract double</td>
<td>sub.d $f2,$f4,$f6</td>
<td>$f2 = f4 - f6</td>
<td>FP sub (double precision)</td>
</tr>
<tr>
<td></td>
<td>Multiply double</td>
<td>mul.d $f2,$f4,$f6</td>
<td>$f2 = f4 \times f6</td>
<td>FP multiply (double precision)</td>
</tr>
<tr>
<td></td>
<td>Divide double</td>
<td>div.d $f2,$f4,$f6</td>
<td>$f2 = f4 / f6</td>
<td>FP divide (double precision)</td>
</tr>
<tr>
<td>Data transfer</td>
<td>Load word congen. 1</td>
<td>lwcl $f1,100($s2)</td>
<td>$f1 = Memory[$s2 + 100]</td>
<td>32-bit data to FP register</td>
</tr>
<tr>
<td></td>
<td>Store word congen.</td>
<td>swcl $f1,100($s2)</td>
<td>Memory[$s2 + 100] = $f1</td>
<td>32-bit data to memory</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>Branch on FP true</td>
<td>bclt 25</td>
<td>if (cond == 1) go to PC + 4 = 100</td>
<td>PC-relative branch if FP cond.</td>
</tr>
<tr>
<td></td>
<td>Branch on FP false</td>
<td>bclf 25</td>
<td>if (cond == 0) go to PC + 4 = 100</td>
<td>PC-relative branch if not cond.</td>
</tr>
<tr>
<td></td>
<td>Compare single (eq, ne, lt, le, gt, ge)</td>
<td>c.lt.s $f2,$f4</td>
<td>if ($f2 &lt; $f4) cond = 1; else cond = 0</td>
<td>FP compare less than single precision</td>
</tr>
<tr>
<td></td>
<td>Compare double (eq, ne, lt, le, gt, ge)</td>
<td>c.lt.d $f2,$f4</td>
<td>if ($f2 &lt; $f4) cond = 1; else cond = 0</td>
<td>FP compare less than double precision</td>
</tr>
</tbody>
</table>

#### MIPS floating-point machine language

<table>
<thead>
<tr>
<th>Name</th>
<th>Format</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>add.s</td>
<td>R</td>
<td>17 16 6 4 2 0</td>
<td>add.s $f2,$f4,$f6</td>
</tr>
<tr>
<td>sub.s</td>
<td>R</td>
<td>17 16 6 4 2 2</td>
<td>sub.s $f2,$f4,$f6</td>
</tr>
<tr>
<td>mul.s</td>
<td>R</td>
<td>17 16 6 4 2 3</td>
<td>mul.s $f2,$f4,$f6</td>
</tr>
<tr>
<td>div.s</td>
<td>R</td>
<td>17 16 6 4 2 2</td>
<td>div.s $f2,$f4,$f6</td>
</tr>
<tr>
<td>add.d</td>
<td>R</td>
<td>17 17 6 4 2 2</td>
<td>add.d $f2,$f4,$f6</td>
</tr>
<tr>
<td>sub.d</td>
<td>R</td>
<td>17 17 6 4 2 2</td>
<td>sub.d $f2,$f4,$f6</td>
</tr>
<tr>
<td>mul.d</td>
<td>R</td>
<td>17 17 6 4 2 2</td>
<td>mul.d $f2,$f4,$f6</td>
</tr>
<tr>
<td>div.d</td>
<td>R</td>
<td>17 17 6 4 2 2</td>
<td>div.d $f2,$f4,$f6</td>
</tr>
<tr>
<td>lwcl</td>
<td>I</td>
<td>49 20 2 100</td>
<td>lwcl $f2,100($s4)</td>
</tr>
<tr>
<td>swcl</td>
<td>I</td>
<td>57 20 2 100</td>
<td>swcl $f2,100($s4)</td>
</tr>
<tr>
<td>bclt</td>
<td>I</td>
<td>17 8 1 25</td>
<td>bclt 25</td>
</tr>
<tr>
<td>bclf</td>
<td>I</td>
<td>17 8 0 25</td>
<td>bclf 25</td>
</tr>
<tr>
<td>c.lt.s</td>
<td>R</td>
<td>17 16 4 2 0 60</td>
<td>c.lt.s $f2,$f4</td>
</tr>
<tr>
<td>c.lt.d</td>
<td>R</td>
<td>17 17 4 2 0 60</td>
<td>c.lt.d $f2,$f4</td>
</tr>
</tbody>
</table>

Field size: 6 bits, 5 bits, 5 bits, 5 bits, 5 bits, 6 bits. All MIPS instructions 32 bits.
Floating Point Complexities

- Operations are somewhat more complicated (see text)
- In addition to overflow we can have “underflow”
- Accuracy can be a big problem
  - IEEE 754 keeps two extra bits, guard and round
  - four rounding modes
  - positive divided by zero yields “infinity”
  - zero divide by zero yields “not a number”
  - other complexities
- Implementing the standard can be tricky
- Not using the standard can be even worse
  - see text for description of 80x86 and Pentium bug!
## x86 FP Instructions

### Data transfer
- `FLD`: memory to `ST(i)`
- `FSTP`: memory to `ST(i)`
- `FLDPI`
- `FLD1`
- `FLDZ`

### Arithmetic
- `FLADD`: memory to `ST(i)`
- `FLSUBRP`: memory to `ST(i)`
- `FLMULP`: memory to `ST(i)`
- `FLDIVRP`: memory to `ST(i)`
- `FSQRT`
- `FABS`
- `FRNDINT`

### Compare
- `FLCOMP`
- `FLUCOMP`
- `FSTSW AX/ mem`

### Transcendental
- `FPATAN`
- `F2XM`
- `FCOS`
- `FPTAN`
- `FPREM`
- `FPSI N`
- `FYL2X`

### Optional variations
- **I**: integer operand
- **P**: pop operand from stack
- **R**: reverse operand order
- But not all combinations allowed

---

**FSTSW**: Move FP status to integer unit for conditional jump.

---

Chapter 3 — Arithmetic for Computers — 27
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>FADD</td>
<td></td>
<td>Both operands in stack; result replaces top of stack.</td>
</tr>
<tr>
<td>FADD</td>
<td>ST(i)</td>
<td>One source operand is $i$th register below the top of stack; result replaces the top of stack.</td>
</tr>
<tr>
<td>FADD</td>
<td>ST(i), ST</td>
<td>One source operand is the top of stack; result replaces $i$th register below the top of stack.</td>
</tr>
<tr>
<td>FADD</td>
<td>mem32</td>
<td>One source operand is a 32-bit location in memory; result replaces the top of stack.</td>
</tr>
<tr>
<td>FADD</td>
<td>mem64</td>
<td>One source operand is a 64-bit location in memory; result replaces the top of stack.</td>
</tr>
</tbody>
</table>

**FIGURE 3.21 The variations of operands for floating-point add in the x86.** Copyright © 2009 Elsevier, Inc. All rights reserved.
Streaming SIMD Extension 2 (SSE2)

- Adds $4 \times 128$-bit registers
  - Extended to 8 registers in AMD64/EM64T
- Can be used for multiple FP operands
  - $2 \times 64$-bit double precision
  - $4 \times 32$-bit single precision
- Instructions operate on them simultaneously
  - Single-Instruction Multiple-Data
FIGURE 3.22 The SSE/SSE2 floating-point instructions of the x86. xmm means one operand is a 128-bit SSE2 register, and mem/xmm means the other operand is either in memory or it is an SSE2 register. We use the curly brackets {} to show optional variations of the basic operations: {SS} stands for Scalar Single precision floating point, or one 32-bit operand in a 128-bit register; {PS} stands for Packed Single precision floating point, or four 32-bit operands in a 128-bit register; {SD} stands for Scalar Double precision floating point, or one 64-bit operand in a 128-bit register; {PD} stands for Packed Double precision floating point, or two 64-bit operands in a 128-bit register; {A} means the 128-bit operand is aligned in memory; {U} means the 128-bit operand is unaligned in memory; {H} means move the high half of the 128-bit operand; and {L} means move the low half of the 128-bit operand. Copyright © 2009 Elsevier, Inc. All rights reserved.
Chapter Three Summary

- Computer arithmetic is constrained by limited precision
- Bit patterns have no inherent meaning but standards do exist
  - two’s complement
  - IEEE 754 floating point
- Computer instructions determine “meaning” of the bit patterns
- Performance and accuracy are important so there are many complexities in real machines
- Algorithm choice is important and may lead to hardware optimizations for both space and time (e.g., multiplication)
- MIPS ISA
  - Core instructions: 54 most frequently used
    - 100% of SPECINT, 97% of SPECFP
  - Other instructions: less frequent