FLOATING POINT OPERATIONS

Mahdi Nazm Bojnordi
Assistant Professor
School of Computing
University of Utah
Overview

- Notes
  - Homework 6 will be posted tonight
    - Deadline: Mar. 5th

- This lecture
  - Floating point operations
  - Basics of logic design
Recall: Floating Point Addition

- Numbers maintain only 4 decimal digits and 2 exponent digits
  - $9.999 \times 10^1 + 1.610 \times 10^{-1}$

- Convert to the larger exponent
  - $9.999 \times 10^1 + 0.016 \times 10^1$

- Add
  - $10.015 \times 10^1$

- Normalize
  - $1.0015 \times 10^2$

- Check for overflow/underflow

- Round
  - $1.002 \times 10^2$

- Re-normalize
Recall: Floating Point Addition

- Numbers maintain only 4 decimal digits and 2 exponent digits
  - \( 9.999 \times 10^1 + 1.610 \times 10^{-1} \)

- Convert to the larger exponent
  - \( 9.999 \times 10^1 + 0.016 \times 10^1 \)

- Add
  - \( 10.015 \times 10^1 \)

- Normalize
  - \( 1.0015 \times 10^2 \)

- Check for overflow/underflow

- Round
  - \( 1.002 \times 10^2 \)

- Re-normalize

If we had more fraction bits, these errors would be minimized
Floating Point Addition

- Numbers maintain only 4 binary digits and 2 exponent digits
  - $1.010 \times 2^1 + 1.100 \times 2^3$

- Convert to the larger exponent
  - $0.0101 \times 2^3 + 1.100 \times 2^3$

- Add
  - $1.1101 \times 2^3$

- Normalize
  - $1.1101 \times 2^3$

- Check for overflow/underflow
Floating Point Addition

- Numbers maintain only 4 binary digits and 2 exponent digits
  - $1.010 \times 2^1 + 1.100 \times 2^3$

- Convert to the larger exponent
  - $0.0101 \times 2^3 + 1.100 \times 2^3$

- Add
  - $1.1101 \times 2^3$

- Normalize
  - $1.1101 \times 2^3$

- Check for overflow/underflow

- IEEE 754 format (32-bit)
  
  0 10000010 1101000000000000000000000000000
Floating Point Addition

Example: add the following two single-precision floating point numbers.

A: 0100000001100000000000000000000000000

B: 0100000110100110000000000000000000000

Steps:

1. Convert to larger exponent
2. Add
3. Normalize
4. Round
### Floating Point Addition

**Example:** add the following two single-precision floating point numbers.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E_A = 128 )</td>
<td>( E_B = 131 )</td>
</tr>
<tr>
<td>( M_A = 1.11_{\text{two}} )</td>
<td>( M_B = 1.010011_{\text{two}} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E_A = 131 )</td>
<td>( E_B = 131 )</td>
</tr>
<tr>
<td>( M_A = 0.00111_{\text{two}} )</td>
<td>( M_B = 1.010011_{\text{two}} )</td>
</tr>
</tbody>
</table>

\[ E_A = E_B = 131 \quad M_A + M_B = 0.00111_{\text{two}} + 1.010011_{\text{two}} = 1.100011_{\text{two}} \]

\[ A + B: \quad 0100000111000010000000000000000000000000 \]
Floating Point Multiplication

- Similar steps are required for multiplication
  - Compute exponent
    - Need to remove bias
  - Multiply significands
    - May end up unnormalized
  - Normalize
    - Shift the point
  - Round
    - Fit in the number of bits
  - Assign sign
    - Compute sign
Floating Point Multiplication

Example: add the following two single-precision floating point numbers.

A: \[ \begin{array}{ccccccccccccccccccc} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array} \]

B: \[ \begin{array}{cccccccccccccccccccccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array} \]

Steps:

1. Compute exponent
2. Multiply significands
3. Normalize
4. Round
5. Compute sign
Example: add the following two single-precision floating point numbers.

A:

\[
E_A = 128 \quad M_A = 1.11_{\text{two}}
\]

B:

\[
E_B = 131 \quad M_B = 1.010011_{\text{two}}
\]

\[
E_{AxB} = 128 + 131 - 127 = 132 \quad M_{AxB} = 10.01000101_{\text{two}}
\]

\[
E_{AxB} = 133 \quad M_{AxB} = 1.001000101_{\text{two}}
\]

A x B:

\[
11000001010010001010000000000000000
\]
MIPS employs separate registers for floating point
- 32-bit registers: $f0, f1, …, f31.
- Each register represents a single-precision number
- Register pairs are used for double-precision
  - Example: $f0$ refers to \{f0, f1\}

<table>
<thead>
<tr>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>add.s</td>
<td>$f2 = f4 + f6$</td>
<td>FP add (single precision)</td>
</tr>
<tr>
<td>sub.s</td>
<td>$f2 = f4 - f6$</td>
<td>FP sub (single precision)</td>
</tr>
<tr>
<td>mul.s</td>
<td>$f2 = f4 \times f6$</td>
<td>FP multiply (single precision)</td>
</tr>
<tr>
<td>div.s</td>
<td>$f2 = f4 / f6$</td>
<td>FP divide (single precision)</td>
</tr>
<tr>
<td>add.d</td>
<td>$f2 = f4 + f6$</td>
<td>FP add (double precision)</td>
</tr>
<tr>
<td>sub.d</td>
<td>$f2 = f4 - f6$</td>
<td>FP sub (double precision)</td>
</tr>
<tr>
<td>mul.d</td>
<td>$f2 = f4 \times f6$</td>
<td>FP multiply (double precision)</td>
</tr>
<tr>
<td>div.d</td>
<td>$f2 = f4 / f6$</td>
<td>FP divide (double precision)</td>
</tr>
</tbody>
</table>
Floating Point Instructions

- Load/Store instructions by coprocessor 1 (c1)
  - Still use integer registers for address computation

- Comparison instructions
  - Set an internal bit (cond) to be inspected by branch instructions

<table>
<thead>
<tr>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>lwcl</td>
<td>$f1,100($s2)</td>
<td>$f1 = Memory[$s2 + 100]</td>
</tr>
<tr>
<td>swcl</td>
<td>$f1,100($s2)</td>
<td>Memory[$s2 + 100] = $f1</td>
</tr>
<tr>
<td>bclt</td>
<td>25</td>
<td>if (cond == 1) go to PC + 4 + 100</td>
</tr>
<tr>
<td>bclf</td>
<td>25</td>
<td>if (cond == 0) go to PC + 4 + 100</td>
</tr>
<tr>
<td>c.lt.s</td>
<td>$f2,$f4</td>
<td>if ($f2 &lt; $f4) cond = 1; else cond = 0</td>
</tr>
<tr>
<td>c.lt.d</td>
<td>$f2,$f4</td>
<td>if ($f2 &lt; $f4) cond = 1; else cond = 0</td>
</tr>
</tbody>
</table>
Code Example

- Convert a temperature in Fahrenheit to Celsius

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

- Assume that constants are stored in global memory
Code Example

- Convert a temperature in Fahrenheit to Celsius

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

- Assume that constants are stored in global memory
Convert a temperature in Fahrenheit to Celsius

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

Assume that constants are stored in global memory

```assembly
f2c:  mtcl  $a0, $f12
      lwcl  $f16, const5($gp)
      lwcl  $f18, const9($gp)
      div.s $f16, $f16, $f18
      lwcl  $f18, const32($gp)
      sub.s $f18, $f12, $f18
      mul.s $f0, $f16, $f18
      jr      $ra
```