Computer Systems Architecture

http://cs.nott.ac.uk/~txa/g51csa/

Thorsten Altenkirch and Liyang Hu

School of Computer Science
University of Nottingham

Lecture 09: Floating Point Arithmetic and the MIPS FPU
Suppose \( f_0 = m_0 \times 2^{e_0} \), \( f_1 = m_1 \times 2^{e_1} \) and \( e_0 \geq e_1 \)

Then \( f_0 + f_1 = (m_0 + m_1 \times 2^{e_1-e_0}) \times 2^{e_0} \)

1. Shift the smaller number right until exponents match
2. Add/subtract the mantissas, depending on sign
3. Normalise the sum by adjusting exponent
4. Check for overflow
5. Round to available bits
6. Result may need further normalisation; if so, goto step 3
Floating Point Multiplication

Suppose \( f_0 = m_0 \times 2^{e_0} \) and \( f_1 = m_1 \times 2^{e_1} \)

Then \( f_0 \times f_1 = m_0 \times m_1 \times 2^{e_0+e_1} \)

1. Add the exponents (be careful, excess-\( n \) encoding!)
2. Multiply the mantissas, setting the sign of the product
3. Normalise the product by adjusting exponent
4. Check for overflow
5. Round to available bits
6. Result may need further normalisation; if so, goto step 3
IEEE 754 Rounding

- Hardware needs two extra bits (round, guard) for rounding
- IEEE 754 defines four rounding modes
  - **Round Up** Always toward \( +\infty \)
  - **Round Down** Always toward \( -\infty \)
  - **Towards Zero** Round down if positive, up if negative
  - **Round to Even** Rounds to nearest even value: in a tie, pick the closest ‘even’ number: e.g. 1.5 rounds to 2.0, but 4.5 rounds to 4.0
- MIPS and Java uses *round to even* by default
Exercise: Rounding

1. Round off the last two digits from the following numbers, interpreting them as 6-bit sign and magnitude:

<table>
<thead>
<tr>
<th>Number</th>
<th>To $+\infty$</th>
<th>To $-\infty$</th>
<th>To Zero</th>
<th>To Even</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+0001.01$</td>
<td>+0010</td>
<td>+0001</td>
<td>+0001</td>
<td>+0001</td>
</tr>
<tr>
<td>$-0001.11$</td>
<td>-0001</td>
<td>-0010</td>
<td>-0001</td>
<td>-0010</td>
</tr>
<tr>
<td>$+0101.10$</td>
<td>+0110</td>
<td>+0101</td>
<td>+0101</td>
<td>+0110</td>
</tr>
<tr>
<td>$+0100.10$</td>
<td>+0101</td>
<td>+0100</td>
<td>+0100</td>
<td>+0100</td>
</tr>
<tr>
<td>$-0011.10$</td>
<td>-0011</td>
<td>-0100</td>
<td>-0011</td>
<td>-0100</td>
</tr>
</tbody>
</table>

2. Give 2.2 to two bits after the binary point: $10.01_2$

3. Round 1.375 and 1.125 to two places: $1.10_2$ and $1.00_2$
IEEE 754 for MIPS

- IEEE operations performed by Floating Point Unit (FPU)
  - MIPS core refers to the FPU as coprocessor 1
  - Previously a separate chip, now usually integrated
- FPU features 32 single precision (32-bit) registers
  - $f0, f1, f2, \ldots, f31$
- Or as 16 pairs of double precision (64-bit) registers
  - $f0, f2, f4, \ldots, f30$ (even registers only!)
  - Here $fi$ actually stands for the pair $fi$ and $f(i + 1)$
- Eight condition code flags for comparison and branching
- FPU instructions do not raise exceptions
  - May need to check for $\pm\infty$ or $NaN$
- MIPS FPU defaults to round to even
MIPS Floating Point Arithmetic

- Single- and double-precision: `mmm.s` and `mmm.d`

```
add.s fdst, fsr0, fsr1  -- addition, single-precision

fdst := fsr0 + fsr1

Example: add.s $f0, $f1, $f2
         $f0 := $f1 + $f2
```

- Double: `add.d $f0, $f2, $f4`
  
  `($f0,$f1) := ($f2,$f3) + ($f4,$f5)`

- Other instructions include: `sub.f`, `mul.f`, `div.f`
  
  where `f` is `s` or `d`

- See *H&P* Appendix A-73 for more
Load / Store for Floating Point

- No encoding for immediate floating-point operands
  - Too many bytes – must be placed in .data segment
  - Assembler directives: .single n or .double n

**l.s fdst, n(src) – load single**
- Load 32-bit word at address src+n into register fdst

**s.d fdst, n(src) – store double**
- Store 64-bit double-word to src+n from register pair fdst
- Address src+n must be double-word aligned!

- Others instructions: l.d and s.s
Floating Point I/O

- How do we input/output floating point numbers?
- Complete list in *Hennessey and Patterson*, Appendix A-44

<table>
<thead>
<tr>
<th>syscall</th>
<th>$v0</th>
<th>Arguments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_float</td>
<td>2</td>
<td>$f12</td>
<td>none</td>
</tr>
<tr>
<td>print_double</td>
<td>3</td>
<td>($f12, $f13)</td>
<td>none</td>
</tr>
<tr>
<td>read_float</td>
<td>6</td>
<td>none</td>
<td>$f0</td>
</tr>
<tr>
<td>read_double</td>
<td>7</td>
<td>none</td>
<td>($f0, $f1)</td>
</tr>
</tbody>
</table>
Example: Area of a Circle

```assembly
.data
pi:    .double 3.141592653589793
.text
.globl main
main:  li $v0, 7   # read_double
       syscall       # radius <- user input
       la $a0, pi
       l.d $f12, 0($a0)  # a := pi
       mul.d $f12, $f12, $f0 # a := a * r
       mul.d $f12, $f12, $f0 # a := a * r
       li $v0, 3   # print_double
       syscall       # print area
       j $ra
```
Floating Point Comparison

- Eight independent condition code (cc) flags, from 0 to 7

```c
c.eq.d cc fsr_{c0}, fsr_{c1} \text{ – compare double for equality}
```

- Flag \( cc := fsr_{c0} == fsr_{c1} \) ? true : false

- General form: \( c.\text{rel.f} \ cc fsr_{c0}, fsr_{c1} \)

<table>
<thead>
<tr>
<th>Relation</th>
<th>Name</th>
<th>Abbr. rel</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>equals</td>
<td>eq</td>
</tr>
<tr>
<td>( \leq )</td>
<td>\text{less than or equals}</td>
<td>le</td>
</tr>
<tr>
<td>&lt;</td>
<td>\text{less than}</td>
<td>lt</td>
</tr>
</tbody>
</table>

- Example: \( c.\text{le.s} 4 \ $f0, \ $f1 \)

  set flag 4 if \( f0 \leq f1 \)
Branching on FPU Flags

- **bc1t** cc label – branch on coprocessor 1 true
  - if (flag cc true) then goto label

- Similarly, there’s **bc1f** – branch on coprocessor 1 false
  - With this we can implement ≠, > and ≥ comparisons

- Remember 0.1 * 0.1 != 0.01?

- One final useful instruction: **abs.f** – absolute value

- **abs.d** fdst, fsr – single precision absolute value
  - fdst := fsr < 0 ? -fsr : fsr or fdst := |fsr|
## Floating Point ↔ Integers Conversion

**round.w.f fdst, fsrc — round to nearest word**

- Round `fsrc` to nearest 32-bit integer
- `fdst` receives bit pattern of a two’s complement integer

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cvt.d.s fdst, fsrc</td>
<td>Convert to double from single</td>
</tr>
<tr>
<td>cvt.s.d fdst, fsrc</td>
<td>Convert to single from double</td>
</tr>
<tr>
<td>cvt.w.f fdst, fsrc</td>
<td>Round to integer, towards zero</td>
</tr>
<tr>
<td>ceil.w.f fdst, fsrc</td>
<td>Round to integer, towards $+\infty$</td>
</tr>
<tr>
<td>floor.w.f fdst, fsrc</td>
<td>Round to integer, towards $-\infty$</td>
</tr>
<tr>
<td>round.w.f fdst, fsrc</td>
<td>Round to nearest integer (not even)</td>
</tr>
</tbody>
</table>

- FPU does not understand two’s complement integers
  - Must move to CPU for processing
### FPU ↔ CPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>mfc1 dst, fsrc</strong> – move from coprocessor 1</td>
<td></td>
</tr>
<tr>
<td>( dst := fsrc )</td>
<td></td>
</tr>
<tr>
<td><strong>mtc1 dst, fsrc</strong> – move to coprocessor 1</td>
<td></td>
</tr>
<tr>
<td>( fsrc := dst )</td>
<td></td>
</tr>
</tbody>
</table>

- Words can be transferred between the FPU and CPU
  - e.g. set \( $f12 := 0 \) using **mtc1** \( $zero, $f12 \)
  - But only the bit pattern, not the value!
- Can be manipulated or stored like any other data
  - e.g. to flip the sign of the single precision \( $f7 \):
    - **mfc1** \( $t0, f7 \)
    - **xor** \( $t0, $t0, 0x80000000 \)
    - **mtc1** \( $t0, f7 \)
Example: Approximately Equal

```
.text
la $a0, tenth
la $a1, hundredth
la $a2, epsilon
l.s $f0, ($a0)
l.s $f1, ($a1)
l.s $f2, ($a2)
mul.s $f0, $f0, $f0  # $f0 := 0.1 * 0.1
sub.s $f3, $f0, $f1  # $f3 := (0.1 * 0.1) - 0.01
abs.s $f3, $f3      # $f3 := |(0.1 * 0.1) - 0.01|
c.lt.s 6 $f3, $f2    # flag 6 = $f3 < 1.0e-7 ?
bc1f 6 not_quite    # if(not flag 6) goto not_quite
                      # approximately equal!

not_quite:
```

```
.data
 tenth:    .float 0.1
 hundredth: .float 0.01
 epsilon:  .float 1.0e-7
```

Floating Point Arithmetic
IEEE 754 for MIPS