## MIPS arithmetic instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add $1,$2,$3</td>
<td>$1 = $2 + $3</td>
<td>3 operands; exception possible</td>
</tr>
<tr>
<td>subtract</td>
<td>sub $1,$2,$3</td>
<td>$1 = $2 – $3</td>
<td>3 operands; exception possible</td>
</tr>
<tr>
<td>add immediate</td>
<td>addi $1,$2,100</td>
<td>$1 = $2 + 100</td>
<td>+ constant; exception possible</td>
</tr>
<tr>
<td>add unsigned</td>
<td>addu $1,$2,$3</td>
<td>$1 = $2 + $3</td>
<td>3 operands; no exceptions</td>
</tr>
<tr>
<td>subtract unsigned</td>
<td>subu $1,$2,$3</td>
<td>$1 = $2 – $3</td>
<td>3 operands; no exceptions</td>
</tr>
<tr>
<td>add imm. unsign.</td>
<td>addiu $1,$2,100</td>
<td>$1 = $2 + 100</td>
<td>+ constant; no exceptions</td>
</tr>
<tr>
<td>multiply</td>
<td>mult $2,$3</td>
<td>Hi, Lo = $2 x $3</td>
<td>64-bit signed product</td>
</tr>
<tr>
<td>multiply unsigned</td>
<td>multu $2,$3</td>
<td>Hi, Lo = $2 x $3</td>
<td>64-bit unsigned product</td>
</tr>
<tr>
<td>divide</td>
<td>div $2,$3</td>
<td>Lo = $2 ÷ $3, Hi = $2 mod $3</td>
<td>Unssigned quotient &amp; remainder</td>
</tr>
<tr>
<td>divide unsigned</td>
<td>divu $2,$3</td>
<td>Lo = $2 ÷ $3, Hi = $2 mod $3</td>
<td>Unssigned quotient &amp; remainder</td>
</tr>
<tr>
<td>Move from Hi</td>
<td>mfhi $1</td>
<td>$1 = Hi</td>
<td>Used to get copy of Hi</td>
</tr>
<tr>
<td>Move from Lo</td>
<td>mflo $1</td>
<td>$1 = Lo</td>
<td>Used to get copy of Lo</td>
</tr>
</tbody>
</table>

## MULTIPLY (unsigned)

Paper and pencil example (unsigned):

**Multiplicand**

<table>
<thead>
<tr>
<th>1000</th>
</tr>
</thead>
</table>

**Multiplier**

<table>
<thead>
<tr>
<th>1001</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
</tr>
<tr>
<td>0000</td>
</tr>
<tr>
<td>0000</td>
</tr>
<tr>
<td>1000</td>
</tr>
</tbody>
</table>

**Product**

| 01101000 |

$m \times n$ bits = $m+n$ bit product

Binary makes it easy:

- 0 => place 0 $(0 \times$ multiplicand)
- 1 => place a copy $(1 \times$ multiplicand)

3 versions of multiply hardware & algorithm:

successive refinement
Unisigned shift-add multiplier (version 1)

- 64-bit Multiplicand reg, 64-bit ALU, 64-bit Product reg, 32-bit multiplier reg

Multiplier = datapath + control

Multiply Algorithm Version 1

Start

Multiplier0 = 1

1. Test Multiplier0

Multiplier0 = 0

1a. Add multiplicand to product & place the result in Product register

2. Shift the Multiplicand register left 1 bit.

3. Shift the Multiplier register right 1 bit.

32nd repetition? No: < 32 repetitions

32nd repetition? Yes: 32 repetitions

Done

<table>
<thead>
<tr>
<th>Product</th>
<th>Multiplier</th>
<th>Multiplicand</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>0011</td>
<td>0000 0010</td>
</tr>
<tr>
<td>0000 0010</td>
<td>0001</td>
<td>0000 0100</td>
</tr>
<tr>
<td>0000 0110</td>
<td>0000</td>
<td>0000 1000</td>
</tr>
<tr>
<td>0000 0110</td>
<td></td>
<td>0000 1010</td>
</tr>
</tbody>
</table>
Observations on Multiply Version 1

• 1/2 bits in multiplicand always 0
  => 64-bit adder is wasted
• 0’s inserted in right of multiplicand as shifted
  => least significant bits of product never changed once formed
• Instead of shifting multiplicand to left, shift product to right?

MULTIPLY HARDWARE Version 2

• 32-bit Multiplicand reg, 32-bit ALU, 64-bit Product reg, 32-bit Multiplier reg
Multiply Algorithm Version 2

Multiplier Multiplicand Product
0011 0010 0000 0000

1. Test Multiplier0
   Multiplier0 = 1
   Multiplier0 = 0

1a. Add multiplicand to the left half of product & place the result in the left half of Product register

Product Multiplier Multiplicand
0000 0000 0110 0010

2. Shift the Product register right 1 bit.

3. Shift the Multiplier register right 1 bit.

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

What's going on?

• Multiplicand stays still and product moves right

Start

A0 A1 A2 A3 A0 A1 A2 A3 A0 A1 A2 A3 A0 A1 A2 A3

B0 B1 B2 B3

P7 P6 P5 P4 P3 P2 P1 P0
Multiply Algorithm Version 2

1. Test Multiplier0
   - Multiplier0 = 1
   - Multiplier0 = 0

1a. Add multiplicand to the left half of product & place the result in the left half of Product register

2. Shift the Product register right 1 bit.

3. Shift the Multiplier register right 1 bit.

<table>
<thead>
<tr>
<th>Product</th>
<th>Multiplier</th>
<th>Multiplicand</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>0011</td>
<td>0010</td>
</tr>
<tr>
<td>0010 0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001 0000</td>
<td>0001</td>
<td>0010</td>
</tr>
<tr>
<td>0011 0000</td>
<td>0001</td>
<td>0010</td>
</tr>
<tr>
<td>0001 1000</td>
<td>0000</td>
<td>0010</td>
</tr>
<tr>
<td>0000 1100</td>
<td>0000</td>
<td>0010</td>
</tr>
<tr>
<td>0000 0110</td>
<td>0000</td>
<td>0010</td>
</tr>
</tbody>
</table>

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

Done

Observations on Multiply Version 2

- Product register wastes space that exactly matches size of multiplier
- => combine Multiplier register and Product register
MULTIPLY HARDWARE Version 3

- 32-bit Multiplicand reg, 32-bit ALU, 64-bit Product reg, (0-bit Multiplier reg)

Multiply Algorithm Version 3

1. Test Product
   - Product0 = 1
   - Product0 = 0

1a. Add multiplicand to the left half of product & place the result in the left half of Product register

2. Shift the Product register right 1 bit.

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

Done
Observations on Multiply Version 3

- 2 steps per bit because Multiplier & Product combined
- MIPS registers Hi and Lo are left and right half of Product
- Gives us MIPS instruction MultU
- How can you make it faster?
- What about signed multiplication?
  - easiest solution is to make both positive & remember whether to complement product when done (leave out the sign bit, run for 31 steps)
  - apply definition of 2’s complement
    - need to sign-extend partial products and subtract at the end
  - Booth’s Algorithm is elegant way to multiply signed numbers using same hardware as before and save cycles
    - can handle multiple bits at a time

Floating Point Arithmetic
Recall IEEE 754 Standard

Representation of floating point numbers in IEEE 754 standard:

<table>
<thead>
<tr>
<th>single precision</th>
<th>1</th>
<th>8</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sign</td>
<td>E</td>
<td>M</td>
</tr>
</tbody>
</table>

**Exponent:**
- bias 127
- binary integer

**Mantissa:**
- sign + magnitude, normalized
- binary significand w/ hidden integer bit: 1.M

Actual exponent is:
- \( e = E - 127 \)

Magnitude of numbers that can be represented is in the range:

\[
N = (-1)^s \times 2^{E-127} \times (1.M)  \\
0 = 0 \times 0 \cdots 0  \\
-1.5 = 1 \times 01111111 \cdots 0
\]

\[
2^{-126} (1.0) \quad \text{to} \quad 2^{127} (2 - 2^{-23})
\]

which is approximately:

\[
1.8 \times 10^{-38} \quad \text{to} \quad 3.40 \times 10^{38}
\]

Floating Point Complexities

- Operations are somewhat more complicated
- 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!
Floating Point Addition Example

Example: Add $9.999 \times 10^1$ and $1.610 \times 10^{-1}$ assuming 4 decimal digits

1. Align decimal point of number with smaller exponent
   
   $1.610 \times 10^{-1} = 0.161 \times 10^0 = 0.0161 \times 10^1$

   Shift smaller number to right

2. Add significands
   
   $9.999$
   
   $0.016$
   
   $10.015 \Rightarrow \text{SUM} = 10.015 \times 10^1$

   NOTE: One digit of precision lost during shifting. Also sum is not normalized

3. Shift sum to put it in normalized form $1.0015 \times 10^2$

4. Since significand only has 4 digits, we need to round the sum
   
   $\text{SUM} = 1.002 \times 10^2$

   NOTE: normalization maybe needed again after rounding, e.g. rounding 9.9999 you get 10.000
Accurate Arithmetic – Guard & Round bits

- IEEE 754 standard specifies the use of 2 extra bits on the right during intermediate calculations – Guard bit and Round bit
- Example: Add $2.56 \times 10^3$ and $2.34 \times 10^2$ assuming 3 significant digits and without guard and round bits
  \[2.56 \times 10^3 = 0.0256 \times 10^2 \]
  \[
  \begin{align*}
  2.34 \\
  0.02 \\
  \hline
  2.36 \times 10^2
  \end{align*}
  \]
- With guard and round bits
  \[
  \begin{align*}
  2.34 \\
  0.0256 \\
  \hline
  2.3656 \times 10^2
  \end{align*}
  \]
  ROUND $\Rightarrow 2.37 \times 10^0$
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 (i.e., algorithms and implementation).
• Next class: we are ready to move on (and implement the processor)