Chapter 3

Integer Math

Recap - MIPS Instructions

Consider a comparison instruction:

\[
\text{slt } \$t0, \$t1, \$\text{zero}
\]

and $t1$ contains the 32-bit number \text{1111 01...01}

What gets stored in $t0$?

The result depends on whether $t1$ is a signed or unsigned number – the compiler/programmer must track this and accordingly use either \text{slt} or \text{sltu}

\[
\begin{align*}
\text{slt } &\$t0, \$t1, \$\text{zero} &\text{ stores 1 in } \$t0 \\
\text{sltu } &\$t0, \$t1, \$\text{zero} &\text{ stores 0 in } \$t0
\end{align*}
\]
Recap - Number Representations

- 32-bit signed numbers (2’s complement):
  - \(0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000\) = \(0_{ten}\) = +1\(_{ten}\)
  - \(0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0001\) = \(1_{ten}\) = +2,147,483,647\(_{ten}\)
  - \(1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\) = \(2,147,483,648\) = \(-2\)\(_{ten}\)
  - \(1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\) = \(-2,147,483,649\) = \(-1\)\(_{ten}\)

- Converting < 32-bit values into 32-bit values
  - Copy the most significant bit (the sign bit) into the “empty” bits
    - \(0010 \rightarrow 0000\ 0010\)
    - \(1010 \rightarrow 1111\ 1010\)

- Sign extend versus zero extend

Integer Addition

- Example: 7 + 6
  - Binary addition is similar to decimal addition.
  - For subtraction, simply add the negative number:
    - A - B involves taking the two’s complement of B (negating B’s bits and adding 1) and adding to A.
A MIPS ALU Implementation

- Zero detect (slt, slti, sltiu, sltu, beq, bne)
- Overflow bit for signed arithmetic (add, addi, sub)

Signed Integer Addition

- Example: 7 + 6

<table>
<thead>
<tr>
<th></th>
<th>(0)</th>
<th>(0)</th>
<th>(1)</th>
<th>(1)</th>
<th>(0)</th>
<th>(Carries)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- An Overflow occurs if the result is out of range
  - Adding a + and - operand, no overflow possible.
  - Adding two + operands
    - Overflow if sign bit is 1.
  - Adding two – operands
    - Overflow if sign bit is 0.
Signed Integer Subtraction

- Add negation of second operand.
- Example: $7 - 6 = 7 + (-6)$
  
  +7: \[0000	ext{ ... }0000\ 0111\]
  
  -6: \[1111	ext{ ... }1111\ 1010\]
  
  +1: \[0000	ext{ ... }0000\ 0001\]

- Overflow if result out of range
  - Subtracting two + or two – operands, no overflow.
  - Subtracting + from – operand
    - Overflow if result sign is 0.
  - Subtracting – from + operand
    - Overflow if result sign is 1.

Summary of Overflow Conditions

- Overflow occurs when the result of an operation cannot be represented in 32-bits, i.e., when the sign bit contains a value bit of the result and not the proper sign bit.
- When adding operands with different signs or when subtracting operands with the same sign, overflow can never occur.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operand A</th>
<th>Operand B</th>
<th>Result indicating overflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A + B$</td>
<td>$\geq 0$</td>
<td>$\geq 0$</td>
<td>$&lt; 0$</td>
</tr>
<tr>
<td>$A + B$</td>
<td>$&lt; 0$</td>
<td>$&lt; 0$</td>
<td>$\geq 0$</td>
</tr>
<tr>
<td>$A - B$</td>
<td>$\geq 0$</td>
<td>$&lt; 0$</td>
<td>$&lt; 0$</td>
</tr>
<tr>
<td>$A - B$</td>
<td>$&lt; 0$</td>
<td>$\geq 0$</td>
<td>$\geq 0$</td>
</tr>
</tbody>
</table>
### Summary of Overflow Conditions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operand A</th>
<th>Operand B</th>
<th>Result indicating overflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>A + B</td>
<td>≤ 0</td>
<td>≤ 0</td>
<td>&lt; 0</td>
</tr>
<tr>
<td>A + B</td>
<td>&lt; 0</td>
<td>&lt; 0</td>
<td>≥ 0</td>
</tr>
<tr>
<td>A - B</td>
<td>≥ 0</td>
<td>≥ 0</td>
<td>&lt; 0</td>
</tr>
<tr>
<td>A - B</td>
<td>&lt; 0</td>
<td>≥ 0</td>
<td>≥ 0</td>
</tr>
</tbody>
</table>

- MIPS signals overflow with an exception – an unscheduled procedure call where the Exception Program Counter (EPC) contains the address of the instruction that caused the exception.
- MIPS `addu` and `subu` instructions will not cause an overflow – to detect the overflow, other instructions would have to be executed.

### Detecting Overflow Logically

- When adding two's complement numbers, overflow will only occur if:
  - The numbers being added have the same sign;
  - The sign of the result is different then the sign of the two operands.
- If we perform the addition
  \[
  \begin{align*}
  a_n \ldots a_2 \ldots a_1 \ldots a_0 \\
  + b_n \ldots b_2 \ldots b_1 \ldots b_0 \\
  \hline
  = s_n \ldots s_2 \ldots s_1 \ldots s_0
  \end{align*}
  \]
  - Overflow can be detected as
    \[
    V = a_n \cdot b_n \cdot s_{n-1} + \overline{a_{n-1}} \cdot \overline{b_{n-1}} \cdot s_{n-1}
    \]
  - Overflow can also be detected as
    \[
    V = c_n \oplus c_{n-1}
    \]
    where \( c_{n-1} \) and \( c_n \) are the carry in and carry out of the most significant bit.
MIPS Arithmetic Logic Unit (ALU)

- Must support the Arithmetic/Logic operations of the ISA
  - add, addi, addiu, addu
  - sub, subu
  - mult, multu, div, divu
  - and, andi, nor, or, or1, xor, xori
  - beq, bne, slt, slti, sltiu, sltu

- With special handling for
  - Sign extend – addi, addiu, slti, sltiu
  - Zero extend – andi, or1, xori
  - Overflow detection – add, addi, sub

What about Performance?

- Critical path of n-bit ripple-carry adder is \( n(1\text{-bit delay}) \)

- Solution – throw hardware at it (Carry Lookahead).
Multiply

- Binary multiplication can be just a bunch of right shifts and adds:

![Diagram of binary multiplication]

- Multiplicand
- Multiplier
- Partial product array
- Can be formed in parallel and added in parallel for faster multiplication
- Double precision product

Multiplication Example

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier</td>
<td>x 1001</td>
</tr>
<tr>
<td>--------------</td>
<td>--------</td>
</tr>
<tr>
<td>--------------</td>
<td>--------</td>
</tr>
<tr>
<td>1000</td>
<td>0000</td>
</tr>
<tr>
<td>0000</td>
<td>1000</td>
</tr>
<tr>
<td>--------------</td>
<td>--------</td>
</tr>
<tr>
<td>Product</td>
<td>1001000</td>
</tr>
</tbody>
</table>

In every step:
- Multiplicand is shifted.
- Next bit of multiplier is examined (also a shifting step).
- If this bit is 1, shifted multiplicand is added to the product.
HW Multiplication Hardware

- 32-bit ALU and multiplicand are untouched.
- The sum keeps shifting right.
- At every step, number of bits in product + multiplier = 64, hence, they share a single 64-bit register.

Multiplication Example

\[
\begin{array}{c}
\text{multiplicand} \\
0 1 1 0 = 6 \\
\end{array}
\]

\[
\begin{array}{c}
\text{32-bit ALU} \\
\end{array}
\]

\[
\begin{array}{c}
\text{product} \\
0 0 0 0 0 1 0 \circledast = 5 \\
\text{ multiplier} \\
\end{array}
\]

\[
\begin{array}{c}
\text{add} \\
0 1 1 0 \rightarrow 0 1 0 \circledast \\
0 0 1 1 \rightarrow 0 0 1 \circledast \\
0 0 1 1 \rightarrow 0 1 0 \circledast \\
0 0 1 1 \rightarrow 1 0 0 \circledast \\
0 0 1 1 \rightarrow 1 1 0 \circledast \\
0 0 1 1 \rightarrow 1 1 0 \circledast \\
\end{array}
\]

\[
\begin{array}{c}
\text{Control} \\
\end{array}
\]

\[
\begin{array}{c}
0 0 0 1 1 1 0 \rightarrow 1 1 1 0 \circledast = 30 \\
0 0 0 1 \rightarrow 1 1 1 0 \circledast = 30 \\
\end{array}
\]
Notes

- The previous algorithm also works for signed numbers (negative numbers in 2’s complement form).

- We can also convert negative numbers to positive, multiply the magnitudes, and convert to negative if signs are different.

- The product of two 32-bit numbers can be a 64-bit number -- hence, in MIPS, the product is saved in two 32-bit registers – HI and LO.

Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoffs
    - A clock is not required.
    - Much higher hardware cost.

- Can be pipelined
  - Several multiplications performed in parallel.
**MIPS Multiplication Instructions**

- Two 32-bit registers for product
  - HI: most-significant 32 bits.
  - LO: least-significant 32-bits.
- Instructions
  - `mul rs, rt / multu rs, rt`
    - 64-bit product in HI/LO
  - `mfhi rd / mflo rd`
    - Move from HI/LO to rd
    - Can test HI value to see if product overflows 32 bits
  - `mul rd, rs, rt`
    - Least-significant 32 bits of product -> rd

**Division**

- Division is just a bunch of quotient digit guesses and left shifts and subtracts.

\[
\text{dividend} = \text{quotient} \times \text{divisor} + \text{remainder}
\]

\[
\begin{array}{cccccc}
\text{divisor} & \text{quotient} & \text{dividend} & \text{partial remainder array} & \text{remainder} \\
\hdashline
0 & 0 & 0 & 0 & n \\
0 & 0 & 0 & 0 & n \\
0 & 0 & 0 & 0 & n \\
0 & 0 & 0 & 0 & n \\
0 & 0 & 0 & 0 & n \\
0 & 0 & 0 & 0 & n \\
\end{array}
\]
## Division

74/8 = 9 rem 2

- Check for 0 divisor.
- Long division approach
  - If divisor bits ≤ dividend bits
    - 1 bit in quotient, subtract.
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit.
- Restoring division
  - Always do the subtract, and if remainder is < 0, add divisor back.
- Signed division
  - Divide using absolute values.
  - Adjust sign of quotient and remainder as required.

n-bit operands yield n-bit quotient and n-bit remainder.

### Division Hardware - Left Shift and Subtract

0 0 1 0 = 2

0 0 0 0 0 0 0 = 6

sub 1 0 0 1 0 1 0 rem neg, so 'ient bit = 0
sub 0 0 0 1 1 1 1 rem neg, so 'ient bit = 0
sub 0 0 0 1 0 0 0 rem pos, so 'ient bit = 1

0 0 1 0 = 3 with 0 remainder
MIPS Divide Instruction

- Divide (`div` and `divu`) generates the remainder in `hi` and the quotient in `lo`.
  ```
  div $s0, $s1  # lo = $s0 / $s1
  # hi = $s0 mod $s1
  ```

- Instructions `mfhi rd` and `mflo rd` are provided to move the quotient and remainder to registers in the register file.

Faster Division

- Can’t use parallel hardware as in multiplier.
  - Subtraction is conditional on sign of remainder.
- Faster dividers (e.g. SRT division) generate multiple quotient bits per step (4 on today’s high-end processors).
  - Uses table lookup.
  - Still requires multiple steps.
Next Week - 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: $\pi$