NUMERICAL OPERATIONS

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

- This lecture
  - Overflow for addition/subtraction
  - Multiplication
  - Division
Addition/Subtraction Overflow

- **Unsigned Numbers:** overflow happens when the last carry (1) cannot be accommodated.
  - i.e., there is not enough memory bits to represent the number

Example:

\[
\begin{align*}
7_{\text{dec}} : & \quad 0111_{\text{bin}} \\
10_{\text{dec}} : & \quad 1010_{\text{bin}} \\
\hline \\
\text{+} & \quad 0111_{\text{bin}} \\
\text{+} & \quad 0010_{\text{bin}} \\
\hline \\
\end{align*}
\]
Addition/Subtraction Overflow

- **Unsigned Numbers**: overflow happens when the last carry (1) cannot be accommodated.
  - i.e., there is not enough memory bits to represent the number

**Example:**

\[ 7_{\text{dec}}: \quad 0111_{\text{bin}} \]
\[ + \quad 10_{\text{dec}}: \quad 1010_{\text{bin}} \]
\[ \downarrow \]
\[ 1_{\text{dec}}: \quad 0001_{\text{bin}} \]

\[ 7_{\text{dec}}: \quad 0111_{\text{bin}} \]
\[ + \quad 2_{\text{dec}}: \quad 0010_{\text{bin}} \]
\[ \downarrow \]
\[ 9_{\text{dec}}: \quad 1001_{\text{bin}} \]
Addition/Subtraction Overflow

- Signed Numbers: overflow happens if the most significant bit is not the same as every bit to its left
  - e.g., the sum of two positive numbers is a negative result
  - e.g., the sum of two negative numbers is a positive result
- Adding positive and negative numbers will not overflow

Example:

\[ +7_{\text{dec}}: 0111_{\text{bin}} \quad +7_{\text{dec}}: 0111_{\text{bin}} \]
\[ +2_{\text{dec}}: 0010_{\text{bin}} \]
\[ +7_{\text{dec}}: 0111_{\text{bin}} \quad +7_{\text{dec}}: 0111_{\text{bin}} \]
\[ +2_{\text{dec}}: 0010_{\text{bin}} \]
Addition/Subtraction Overflow

- **Signed Numbers:** overflow happens if the most significant bit is not the same as every bit to its left
  - e.g., the sum of two positive numbers is a negative result
  - e.g., the sum of two negative numbers is a positive result
- Adding positive and negative numbers will not overflow

**Example:**

\[ +7_{\text{dec}}: 0111_{\text{bin}} \]
\[ +7_{\text{dec}}: 0111_{\text{bin}} \]

\[ +2_{\text{dec}}: 0010_{\text{bin}} \]
\[ +2_{\text{dec}}: 0010_{\text{bin}} \]

\[ -6_{\text{dec}}: 1010_{\text{bin}} \]
\[ -7_{\text{dec}}: 1001_{\text{bin}} \]

\[ +1_{\text{dec}}: 0001_{\text{bin}} \]
MIPS Instructions

- Instructions `add`, `addi`, and `sub` may cause exceptions on overflow
  - Software needs to handle exceptions
    - More on this later

- MIPS provides the `addu`, `addiu`, and `subu` instructions that work with unsigned integers and never flag an overflow
  - Other instructions may be executed to detect the overflow
Dealing with $>32$-bit

Example 1: Store the unsigned value $6538305685_{\text{dec}}$ in the register file.

$1\,10000101\,10110110\,10100000\,10010101_{\text{bin}}\quad 1\,85B6\,A095_{\text{hex}}$
Dealing with >32-bit

- Example 1: Store the unsigned value $6538305685_{dec}$ in the register file.

```
1 10000101 10110110 10100000 10010101
1 85B6 A095
```

```
lui $t1, 0x85B6
ori $t1, $t1, 0xA095
ori $t0, $zero, 1
```

Two 32-bit registers ($t0$ and $t1$)
Example 2: add the unsigned 64-bit values in ($t0,$t1) and ($t2,$t3). Store the result in ($t4,$t5).

Two 32-bit registers for each value

```
64 bits

$t0$ $t1$

$t2$ $t3$

$t4$ $t5$
```

addu $t5,$t1,$t3
addu $t4,$t0,$t2
Example 2: add the unsigned 64-bit values in ($t0,$t1) and ($t2,$t3). Store the result in ($t4,$t5).

Two 32-bit registers for each value

```
addu $t5, $t1, $t3
addu $t4, $t0, $t2
sltu $t6, $t5, $t1
sltu $t7, $t5, $t3
beq $t6, $zero, OK
beq $t7, $zero, OK
addiu $t4, $t4, 1
OK: ...
```
Multiplication Example

- Multi-step process

- 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

```
Multiplicand  1000_{ten}
Multiplier    x  1001_{ten}
--------------
  1000
  0000
  0000
  1000
--------------
Product      1001000_{ten}
```
**Multiplication Example**

- **Multi-step process**

- **Multiplicand**: \(1000_{\text{ten}}\)
- **Multiplier**: \(1001_{\text{ten}}\)

\[
\begin{array}{c}
\times \\
\hline
1000_{\text{ten}} \\
1001_{\text{ten}} \\
\hline
1001000_{\text{ten}}
\end{array}
\]
Multiplication Algorithm 2

- A more efficient algorithm
  - 32-bit ALU and multiplicand is untouched
  - sum keeps shifting right
    - number of bits in product + multiplier = 64,
    - hence, they share a single 64-bit register
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 disagree.

The product of two 32-bit numbers can be a 64-bit number.

In MIPS, the product is saved in two 32-bit registers.
Signed multiplication (mult)

- `mult $s2, $s3` computes the product and stores it in two “internal” registers that can be referred to as `hi` and `lo`.

- `mfhi $s0` moves the value in `hi` into $s0.
- `mflo $s1` moves the value in `lo` into $s1.

Similarly for unsigned multiplication (multu)

- `multu $s2, $s3` moves the value in `hi` into $s0.
- `mfhi $s0` moves the value in `lo` into $s1.`
Multiplication: Fast Algorithm

- The previous algorithm requires a clock to ensure that the earlier addition has completed before shifting.

- This algorithm can quickly set up most inputs — it then has to wait for the result of each add to propagate down — faster because no clock is involved.

- Note: high transistor cost.
Division Example

- **Multi-step process**
  - Shift divisor right and compare it with current dividend
    - If divisor is larger, shift 0 as the next bit of the quotient
    - If divisor is smaller, subtract to get new dividend and shift 1 as the next bit of the quotient

\[
\begin{array}{c|c}
\text{Divisor} & \text{1000}_{\text{ten}} \\
\hline
\text{Quotient} & \text{1001}_{\text{ten}} \\
\text{Dividend} & \text{1001010}_{\text{ten}} \\
\hline
\text{Remainder} & \text{10}_{\text{ten}}
\end{array}
\]
## Division Example

- Divide $7_{ten} \ (0000 \ 0111_{two})$ by $2_{ten} \ (0010_{two})$

<table>
<thead>
<tr>
<th>Iter</th>
<th>Step</th>
<th>Quot</th>
<th>Divisor</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initial values</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Division Example

- Divide $7_{ten} (0000\ 0111_{two})$ by $2_{ten} (0010_{two})$

<table>
<thead>
<tr>
<th>Iter</th>
<th>Step</th>
<th>Quot</th>
<th>Divisor</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initial values</td>
<td>0000</td>
<td>0010\ 0000</td>
<td>0000\ 0111</td>
</tr>
<tr>
<td>1</td>
<td>Rem = Rem – Div</td>
<td>0000</td>
<td>0010\ 0000</td>
<td>1110\ 0111</td>
</tr>
<tr>
<td></td>
<td>Rem &lt; 0 ⇒ +Div, shift 0 into Q</td>
<td>0000</td>
<td>0010\ 0000</td>
<td>0000\ 0111</td>
</tr>
<tr>
<td></td>
<td>Shift Div right</td>
<td>0000</td>
<td>0001\ 0000</td>
<td>0000\ 0111</td>
</tr>
<tr>
<td>2</td>
<td>Same steps as 1</td>
<td>0000</td>
<td>0001\ 0000</td>
<td>1111\ 0111</td>
</tr>
<tr>
<td>3</td>
<td>Same steps as 1</td>
<td>0000</td>
<td>0000\ 0100</td>
<td>0000\ 0111</td>
</tr>
<tr>
<td>4</td>
<td>Rem = Rem – Div</td>
<td>0000</td>
<td>0000\ 0100</td>
<td>0000\ 0011</td>
</tr>
<tr>
<td></td>
<td>Rem &gt;= 0 ⇒ shift 1 into Q</td>
<td>0001</td>
<td>0000\ 0100</td>
<td>0000\ 0011</td>
</tr>
<tr>
<td></td>
<td>Shift Div right</td>
<td>0001</td>
<td>0000\ 0010</td>
<td>0000\ 0011</td>
</tr>
<tr>
<td>5</td>
<td>Same steps as 4</td>
<td>0011</td>
<td>0000\ 0001</td>
<td>0000\ 0001</td>
</tr>
</tbody>
</table>