Computer Systems Architecture
http://cs.nott.ac.uk/~txa/g51csa/

Thorsten Altenkirch and Liyang Hu

School of Computer Science
University of Nottingham

Lecture 07: Signedness, Overflow, Multiplication and Division
MIPS can interpret words as signed or unsigned
Many instructions have signed and unsigned variants

**slt vs sltu dst, src₀, src₁ – Set on Less-Than**

- `slt` interprets `src₀` and `src₁` as *signed* integers
- While `sltu` interprets `src₀` and `src₁` as *unsigned*

What is the result from each of the following instructions?

<table>
<thead>
<tr>
<th>$s₁$</th>
<th>FFFFFFFF₁₆</th>
<th><code>slt $s₀, $s₁, $s₂</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>$s₂$</td>
<td>00000001₁₆</td>
<td><code>sltu $s₀, $s₁, $s₂</code></td>
</tr>
</tbody>
</table>

- `slt $s₀ = 1` because $-1 < 1$
- `sltu $s₀ = 0` because $2^{32} - 1 \not< 1$
What about signed bytes and half-words (in memory)?

Run: `lbu $s0, ($a0)` with \[M[a0] = 123456FF_{16}\]

- Loads the byte \(FF_{16} = -1_{10}\) into \(s0\)
- But now \(s0\) contains \(000000FF_{16} = 255_{10}\)!
- How do we preserve the intended value?

### Sign Extension

- For a signed byte, copy the MSB (bit 7) 24 times:
  
  \[
  \begin{array}{ccc}
  x & \cdots & \leftarrow & \cdots & x & xyxy \ \text{yyyy} \\
  \leftarrow & 24 \text{ bits} & \rightarrow & \leftarrow & 8 \text{ bits} & \rightarrow \\
  \end{array}
  \]

- e.g. \(D6_{16} = -42_{10}\) is sign-extended to \(FFFFFFFD6_{16}\)

- `lb` and `lh` performs *sign extension*; `lbu` and `lhu` does not

- Usually use bytes for characters, so `lbu` used more often
Signed Overflow

- add/addi/addu/addiu use the same addition circuits
  But constants in addi/addiu are always sign-extended
- However, add and addi also check for overflow
  Overflow when result exceeds \(-2^{31} \leq x < 2^{31}\)
  On overflow, trigger an error exception
  Try `li $t0, 0x7fffffff` in SPIM!
  `addi $t0, $t0, 1`
- addu/addiu doesn’t check for overflow
  We can check for ourselves: avoid triggering exception
- There is also subu, subtraction without overflow checking
Checking for Overflow

\[ dst = src_0 + src_1 \]

if (sign(src_0) != sign(src_1))
    goto no_overflow;
if (sign(dst) == sign(src_0))
    goto no_overflow;
    # we have overflow!
no_overflow:
    # rest of program

- xor dst, src_0, src_1 returns
  - a positive dst when the sign bit of src_0 and src_1 match
  - a negative dst when the sign bit of src_0 and src_1 differ

\[ \text{sign}(x) = x \& 0x8000 \] for 32 bit numbers.
Multiplication

- Product of \(m\)- and \(n\)-digit numbers requires \(m + n\) digits
- Multiplying 4-digit numbers needs 8 digits
- Binary case needs only multiply by 0 or 1, and addition

### Long Multiplication in Decimal

\[
\begin{array}{cccc}
6 & 2 & 9 & 5 \\
\times & 2 & 8 & 1 & 7 \\
\hline
& 4 & 4 & 0 & 6 & 5 \\
& 6 & 2 & 9 & 5 \\
& 5 & 0 & 3 & 6 & 0 \\
\hline
\end{array}
\]

\[
\begin{array}{cccc}
+ & 1 & 2 & 5 & 9 & 0 \\
\hline
\end{array}
\]

\[
= 1 & 7 & 7 & 3 & 3 & 0 & 1 & 5
\]
Product of \( m \)- and \( n \)-digit numbers requires \( m + n \) digits

- Multiplying 4-digit numbers needs 8 digits
- Binary case needs only multiply by 0 or 1, and addition

**Long Multiplication in Binary**

\[
\begin{array}{cccccc}
& 1 & 1 & 0 & 1 & a = 13_{10} \\
\times & 1 & 0 & 1 & 1 & b = 11_{10} \\
\hline
& 1 & 1 & 0 & 1 \\
1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 \\
+ & 1 & 1 & 0 & 1 \\
\hline
= & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & c = 143_{10}
\end{array}
\]
n-Bit Binary Multiplication

- Given $a$ and $b$, calculates $c = a \times b$
- Optimisation: exit as soon as $b == 0$
- But doesn’t work for signed numbers!
  - Take magnitude, multiply, then fix sign?
- Equivalent C code:
  ```
c = 0;
for(i = 1; i < n; i = i + 1) {
    if(b & 0x01 != 0)
        c = c + a;
    b = b >> 1;
    a = a << 1;
}
```
Signed Binary Multiplication

- Signed numbers can be infinitely sign-extended
  - Positive numbers prefixed by an ‘infinite’ number of 0s
  - Negative numbers prefixed by an ‘infinite’ number of 1s
- Sign-extend \( a \) and \( b \) to \( 2n \) digits for multiplication
  - Must loop \( 2n \) times as a result
  - But can still exit early when \( b == 0 \)

**Example:** \(-3 \times 5\)

\[
\begin{array}{cccccccccc}
\ldots & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & a = -3_{10} \\
\times \ldots & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & b = 5_{10} \\
\hline \\
\ldots & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
\ldots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
+ \ldots & 1 & 1 & 1 & 1 & 0 & 1 & & & \text{Partial Sums} \\
\hline \\
= \ldots & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & c = 15_{10}
\end{array}
\]
Division

Long Division in Decimal

\[
\begin{array}{ccccccc}
\text{Divisor} & 0 & 0 & 2 & 1 & 3 & \text{Quotient} \\
\div & 8 & 1 & 1 & 7 & 3 & 3 & 0 \\
\text{Dividend} & & & & & & \\
\hline
& 8 & 1 & 0 & 0 & 0 & 0 \times 0 \\
& 8 & 1 & 0 & 0 & 0 & \times 0 \\
& 8 & 1 & 0 & 0 & \times 2 \\
- & 1 & 6 & 2 & 0 & 0 & \\
\hline
& 1 & 1 & 3 & 0 \\
& 8 & 1 & 0 & \times 1 \\
- & 8 & 1 & 0 & \\
\hline
& 3 & 2 & 0 \\
& 8 & 1 & \times 3 \\
- & 2 & 4 & 3 & \\
\hline
& 7 & 7 & \text{Remainder} \\
\end{array}
\]
## Division

### Long Division in Binary

<table>
<thead>
<tr>
<th>Divisor</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dividend</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- 1 0 1 0 0 0 0 0 0 × 0

- 1 0 1 0 0 0 0 0 0 × 0

- 1 0 1 0 0 0 0 0 0 × 1

- 1 0 0 0 1 0 0

- 1 0 0 0 1 0 0 0 0 × 0

- 1 0 1 0 1 0 0 0 × 1

- 1 0 0 0 0 0 1

- 1 0 1 0 1 0 0 0 0 0 0 × 1

- 1 1 1

- 1 0 1 0 1 1 1 1 1 1 1

- 1 0

---

Remainder
n-Bit Binary Division

- Given dividend $a$ and divisor $b$
  - Calculates their quotient $d = a / b$
  - Leaves the remainder in $a = a \% b$

- Equivalent C code:
  ```c
  d = 0;
  b = b << n;
  for(i = 0; i < n; i++) {
    b = b >> 1;
    d = d << 1;
    if(a >= b) {
      a = a - b;
      d = d + 1;
    }
  }
  ```
Mul\(^n\) produces a 64-bit word; div\(^n\) two 32-bit results
- No way to encode two destination registers...
- Slow compared to e.g. addition; takes many cycles
  - Would stall the next instructions in the pipeline
- Independent unit (from main ALU) for mul\(^n\) and div\(^n\)
  - Source operands come from the usual register file
  - Results written to two special registers HI and LO

\textbf{mfhi \textit{dst} / mflo \textit{dst} — move from HI/LO}:
- \texttt{mfhi \textit{dst} — \textit{dst} := HI}
- \texttt{mflo \textit{dst} — \textit{dst} := LO}
MIPS Multiplication

\[ \text{mult} \ src_0, \ src_1 / \ \text{multu} \ src_0, \ src_1 \quad \text{— multiplication} \]

- HI := upper 32 bits of \( src_0 \times src_1 \)
- LO := lower 32 bits of \( src_0 \times src_1 \)
- mult treats \( src_0/src_1 \) as signed; multu as unsigned

\[ \text{mul} \ dst, \ src_0, \ src_1 \quad \text{— multiplication (no overflow check)} \]

- \( dst \) := LO := lower 32 bits of \( src_0 \times src_1 \)
- Single instruction equivalent of \( \text{mult} \ src_0, \ src_1 \)
- \( \text{mflo} \ dst \)
- No mulu – same result signed or unsigned
Multiplication Overflow

- Pseudoinstructions `mulo` and `mulou` check for overflow.
  - Result too large for a 32-bit signed/unsigned word.
  - How do these pseudoinstructions work?
- Replace `break $0` with your own error-handling code.

**mulo** $dst$, $src_0$, $src_1$

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mult $src_0$, $src_1</code></td>
<td>Multiply $src_0$ by $src_1$</td>
</tr>
<tr>
<td><code>mfhi $at</code></td>
<td>Store high half of $dst$ in $at$</td>
</tr>
<tr>
<td><code>beq $at, $0, no_overflow</code></td>
<td>Check if $at$ is zero</td>
</tr>
<tr>
<td></td>
<td>- Break $0$ if not zero</td>
</tr>
<tr>
<td><code>no_overflow:</code></td>
<td></td>
</tr>
<tr>
<td><code>mflo $dst</code></td>
<td>Store low half of $dst$ in $dst$</td>
</tr>
</tbody>
</table>

**mulou** $dst$, $src_0$, $src_1$

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mult $src_0$, $src_1</code></td>
<td>Multiply $src_0$ by $src_1$</td>
</tr>
<tr>
<td><code>mfhi $at</code></td>
<td>Store high half of $dst$ in $at$</td>
</tr>
<tr>
<td><code>mflo $dst</code></td>
<td>Store low half of $dst$ in $dst$</td>
</tr>
<tr>
<td><code>sra $dst$, $dst$, 31</code></td>
<td>Shift right $dst$ by 31 bits</td>
</tr>
<tr>
<td><code>beq $at, $dst, no_overflow</code></td>
<td>Check if $at$ is equal to $dst$</td>
</tr>
<tr>
<td></td>
<td>- Break $0$ if not equal</td>
</tr>
<tr>
<td><code>no_overflow:</code></td>
<td></td>
</tr>
<tr>
<td><code>mflo $dst</code></td>
<td>Store low half of $dst$ in $dst$</td>
</tr>
</tbody>
</table>
MIPS Division

\[ \text{div } src_0, src_1 \quad / \quad \text{divu } src_0, src_1 \quad \text{— division} \]

- \( HI := src_0 \mod src_1 \)
- \( LO := src_0 \div src_1 \)
- \( \text{div} \) treats \( src_0/src_1 \) as signed; \( \text{divu} \) as unsigned

\[ \text{div } dst, src_0, src_1 \quad / \quad \text{divu } dst, src_0, src_1 \quad \text{— division} \]

- Three argument pseudoinstruction version of \( \text{div}/\text{divu} \)
- Expands to \( \text{div } src_0, src_1 \) or \( \text{divu } src_0, src_1 \)
  \[
  \text{mflo } dst \quad \text{mflo } dst
  \]