Fast Arithmetic

Philipp Koehn

19 October 2016
arithmetic
Addition (Immediate)

- Load immediately one number ($s0 = 2$)
  
  ```
  li $s0, 2
  ```

- Add 4 ($s1 = s0 + 4 = 6$)
  
  ```
  addi $s1, $s0, 4
  ```

- Subtract 3 ($s2 = s1 - 3 = 3$)
  
  ```
  addi $s2, $s1, -3
  ```
Addition (Register)

- Load immediately one number ($s0 = 2$)
  
  ```
  li $s0, 2
  ```

- Add value from $s5 ($s1 = $s0 + $s5)
  
  ```
  add $s1, $s0, $s5
  ```

- Subtract value from $s6 ($s2 = $s1 - $s6)
  
  ```
  sub $s2, $s1, $s6
  ```
Overflow

• Signed integers operations: add, addi, and sub
  – overflow triggers exceptions
  – similar to interrupt
  – register $mfc0$ contains address of exception program

• Unsigned integers operations: addu, addiu, and subu
  – no overflow handling (as in C programming language)
Code for Detecting Overflow

- Overflow for unsigned integers operations can be detected from result

- Actual detection code is a bit intricate

- If you are interested
  → consult Section 3.2 in Patterson/Hennessy textbook
fast addition
Recall: N-Bit Addition

\[
\begin{array}{c}
11 \\
+11 \\
\hline
\end{array}
\]
Recall: N-Bit Addition

\[
\begin{array}{c}
1 & 1 \\
+ & 1 \\
\hline
1 & 0 \\
\end{array}
\]

1+1 = 0, carry the 1
Recall: N-Bit Addition

11
+11
---
11
---
10

1+1+1 = 1, carry the 1
Recall: N-Bit Addition

\[
\begin{array}{c}
11 \\
+11 \\
\hline \\
11 \\
\hline \\
110
\end{array}
\]

copy carry bit
Fast Addition

- We defined n-bit adding as a sequential process
- More bits → addition takes longer
- 32 bit addition gets very slow
- Faster addition: Carry Lookahead
**Problem: Carry Propagation**

- 1+1 addition always causes a carry

  $$1+1 + \text{carry}1 = 1, \text{carry} 1$$
  $$1+1 + \text{carry}0 = 0, \text{carry} 1$$

- 0+0 addition never causes a carry

  $$0+0 + \text{carry}1 = 1, \text{carry} 0$$
  $$0+0 + \text{carry}0 = 0, \text{carry} 0$$

- 0+1 and 1+0 addition may cause a carry

  $$0+1 + \text{carry}1 = 0, \text{carry} 1$$
  $$0+1 + \text{carry}0 = 1, \text{carry} 0$$
Generate and Propagate

- Compute for each bit, if it generates or propagates carry

- Example

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>Generate</th>
<th>Propagate</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operand</td>
<td>0100</td>
<td>0110</td>
<td>0100 0001</td>
<td>0110 1111</td>
<td>1001 111-</td>
</tr>
<tr>
<td>A</td>
<td>0100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0110</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Generate: $a_i \text{ AND } b_i$

- Propagate: $a_i \text{ OR } b_i$

- Carry: ?
4-Bit Adder

- First compute generate and propagate for all bits
  - generate: \( g_i = a_i \text{ AND } b_i \)
  - propagate: \( p_i = a_i \text{ OR } b_i \)

- Compute carries for each bit
  - \( c_1 = g_0 \text{ OR } (p_0 \text{ AND } c_0) \)
  - \( c_2 = g_1 \text{ OR } (p_1 \text{ AND } g_0) \text{ OR } (p_1 \text{ AND } p_0 \text{ AND } c_0) \)
  - \( c_3 = g_2 \text{ OR } (p_2 \text{ AND } g_1) \text{ OR } (p_2 \text{ AND } p_1 \text{ AND } g_1) \text{ OR } (p_2 \text{ AND } p_1 \text{ AND } p_0 \text{ AND } c_0) \)
  - \( c_4 = g_3 \text{ OR } (p_3 \text{ AND } g_2) \text{ OR } (p_3 \text{ AND } p_2 \text{ AND } g_2) \text{ OR } (p_3 \text{ AND } p_2 \text{ AND } p_1 \text{ AND } g_1) \text{ OR } (p_3 \text{ AND } p_2 \text{ AND } p_1 \text{ AND } p_0 \text{ AND } c_0) \)

- The carry computations require no recursion
  --- but use a lot of gates

- We may want to stop at 4 bits with this idea
16-Bit Adder

- Combine 4 4-bit adders

- For each 4-bit adder, compute
  - "super" propagate = $P = p_0 \text{ AND } p_1 \text{ AND } p_2 \text{ AND } p_3$
  - "super" generate = $g_3 \text{ OR } (p_3 \text{ AND } g_2) \text{ OR } (p_3 \text{ AND } p_2 \text{ AND } g_1)$
    OR $((p_3 \text{ AND } p_2 \text{ AND } p_1 \text{ AND } g_0)$

- Compute super carry $C_j$ from super propagate $P_j$ and super generate $G_j$

- Use $C_j$ as input carry to the 4-bit adders
1. compute propagate $p_i$ and generate $g_i$

2. compute carry $c_i$
   compute super propagate $P_j$ and super generate $G_j$

3. compute super carry $C_j$

4. carry out all bitwise additions
Trade-Off

• Higher $n$ in $n$-bit adders
  – more gates in circuit
  – faster computation

• Modern CPUs can pack more gates on a chip
  $\Rightarrow$ speed-up at same clock speed
multiplication
Recall Method

• Elementary school multiplication:

\[ \begin{array}{c}
10101 \\
\times 1101 \\
\hline
10101 \\
0 \\
10101 \\
10101 \\
\hline
100010001
\end{array} \]

(in decimal: \(23 \times 13 = 299\))

• Idea

  - shift second operand to right (get last bit)
  - if carry: add second operand to sum
  - rotate first operand to left (multiply with binary 10)
Multiplication in Hardware

- Control unit runs microprogram
  
  loop 32 times:
  
  if lowest bit of multiplyer=1
  
  add multiplicand to product
  
  shift multiplicand left
  
  shift multiplyer right

- Speed
  
  - 32 iterations
  
  - 3 operations each
    
    (add + shift + shift)
  
  \[ \rightarrow \] almost 100 operations

- Note: multiplying 32 bit numbers may result in 64 bit product
Parallelize the 3 Operations

- The 3 operations in each loop affect different registers
  - add: product
  - shift left: multiplicand
  - shift right: multiplier

⇒ These can be executed in parallel
  (note: read is executed before write)
Parallelize the Iterations

- Sum of 32 independently computed values
- More adders $\rightarrow$ some summing can be done in parallel
- Binary tree $\rightarrow \log_2 32 = 5$ cycles
MIPS Instructions

- 32 bit multiplication results in 64 bit product

- Special 64 bit register holds result
  - hi: high word
  - lo: low word

- Low word has to be retrieved by another instruction
  
  ```
  mult $s1, $s2
  mflo $s0
  ```

- Since this is the typical usage, pseudo-instruction
  
  ```
  mul $s0, $s1, $s2
  ```

  More on that later
division
Elementary School Division

\[ 1011 \div 10 = 101 \]

\[
\begin{array}{c}
1011 \\
\text{shift divisor sufficiently to the left} \\
101 \\
\text{check if subtraction is possible} \\
1001 \\
\text{yes} \rightarrow \text{add result bit 1, carry out subtraction} \\
010 \\
\text{no} \rightarrow \text{add result bit 0} \\
011 \\
\text{pull down bit from dividend} \\
110 \\
\text{shift divisor to the right} \\
\text{not possible} \rightarrow \text{done, note remainder} \\
\text{otherwise go to step 2} \\
1 \text{ Remainder}
\end{array}
\]

• Algorithm

1. shift divisor sufficiently to the left
2. check if subtraction is possible
   yes \rightarrow \text{add result bit 1, carry out subtraction}
   no \rightarrow \text{add result bit 0}
3. pull down bit from dividend
4. shift divisor to the right
   not possible \rightarrow \text{done, note remainder}
   otherwise go to step 2
Algorithm Refinement

1. Shift divisor sufficiently to the left
   - hard for machine to determine
     → shift to maximum left
     - 32 bit division: use 64 register, push 32 positions

2. Check if subtraction is possible
   yes → add result bit 1, carry out subtraction
   no → add result bit 0

   - we always carry out subtraction
   - if overflow, do not use result

3. Pull down bit from dividend

4. Shift divisor to the right
   not possible → done, note remainder
   otherwise go to step 2
Division in Hardware

- Operations similar to multiplication
  - shift divisor
  - subtraction
  - indication if subtraction should be accepted

- These operations can be parallelized

- But: iterations cannot be parallelized the same way
  (sophisticated prediction methods guess outcome of subtractions)
MIPS Instructions

• 32 bit division results in 32 bit quotient and 32 bit remainder
  – hi: remainder
  – lo: quotient

• Quotient has to be retrieved by another instruction

  div $s1, $s2
  mflo $s0