CS352H: Computer Systems Architecture

Lecture 5: MIPS Integer ALU

September 10, 2009
Integer Addition

Example: 7 + 6

\[ \begin{array}{ccccccc}
\text{Carries} & (0) & (0) & (1) & (1) & (0) & \\
\ldots & 0 & 0 & 0 & 1 & 1 & 1 \\
\ldots & 0 & 0 & 0 & 1 & 1 & 0 \\
\ldots & (0) & (0) & (0) & 1 & (1) & 0 & (0) & 1 \\
\end{array} \]

- Overflow if result out of range
  - Adding +ve and –ve operands, no overflow
  - Adding two +ve operands
    - Overflow if result sign is 1
  - Adding two –ve operands
    - Overflow if result sign is 0
Integer Subtraction

- Add negation of second operand
- Example: $7 - 6 = 7 + (-6)$
  
  
  
  +7: 0000 0000 ... 0000 0111
  
  -6: 1111 1111 ... 1111 1010
  
  +1: 0000 0000 ... 0000 0001

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

- Some languages (e.g., C) ignore overflow
  - Use MIPS `addu`, `addui`, `subu` instructions
- Other languages (e.g., Ada, Fortran) require raising an exception
  - Use MIPS `add`, `addi`, `sub` instructions
  - On overflow, invoke exception handler
    - Save PC in exception program counter (EPC) register
    - Jump to predefined handler address
    - `mfc0` (move from coprocessor reg) instruction can retrieve EPC value, to return after corrective action
Using $c_i$ for $\text{CarryIn}_i$

\[
c_2 = b_1c_1 + a_1c_1 + a_1b_1
\]

and

\[
c_1 = b_0c_0 + a_0c_0 + a_0b_0
\]

Substituting for $c_1$, we get:

\[
c_2 = a_1a_0b_0 + a_1a_0c_0 + a_1b_0c_0 + b_1a_0b_0 + b_1a_0c_0 + b_1b_0c_0 + a_1b_1
\]

Continuing this to 32 bits yields a fast, but unreasonably expensive adder

Just how fast?

Assume all gate delays are the same regardless of fan-in
Carry-Lookahead Adders

- The basic formula can be rewritten:
  - $c_{i+1} = b_i c_i + a_i c_i + a_i b_i$
  - $c_{i+1} = a_i b_i + (a_i + b_i)c_i$

- Applying it to $c_2$, we get:
  - $c_2 = a_1 b_1 + (a_1 + b_1)(a_0 b_0 + (a_0 + b_0)c_0)$

- Define two “signals” or abstractions:
  - Generate: $g_i = a_i \times b_i$
  - Propagate: $p_i = a_i + b_i$

- Redefine $c_{i+1}$ as:
  - $c_{i+1} = g_i + p_i \times c_i$

- So $c_{i+1} = 1$ if
  - $g_i = 1$ (generate) or
  - $p_i = 1$ and $c_i = 1$ (propagate)
Carry-Lookahead Adders

- Our logic equations are simpler:
  - \( c_1 = g_0 + p_0c_0 \)
  - \( c_2 = g_1 + p_1g_0 + p_1p_0c_0 \)
  - \( c_3 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0 \)
  - \( c_4 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0 \)
Carry-Lookahead Adders

\[ a_i \quad b_i \quad c_{i+1} \]

0 0 0  kill
0 1  c_i  propagate
1 0  c_i  propagate
1 1  1  generate

\[
c_0 = g_0 + p_0 c_0
\]

\[
c_1 = g_1 + p_1 g_0 + p_1 p_0 c_0
\]

\[
c_2 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0
\]

\[
c_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0
\]
Carry-Lookahead Adders

- How much better (16-bit adder)?
  - Ripple-carry: \(16 \times T_{\text{add}} = 16 \times 2 = 32\) gate delays
  - Carry-lookahead: \(T_{\text{add}} + \max (p_i, g_i) = 2 + 2 = 4\)
  - Much better, but still too profligate

- What if we apply another level of this abstraction?
  - Use the four-bit adder on the previous slide as a building block
  - Define \(P\) and \(G\) signals
    - \(P_0 = p_3p_2p_1p_0\)
    - \(G_0 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0\)
    - Similarly for \(P_1 - P_3\) and \(G_1 - G_3\)
  - Derive equations for \(C_1 - C_4\)
    - \(C_1 = G_0 + P_0c_0\)
    - \(C_2 = G_1 + P_1G_0 + P_1P_0c_0\), etc.
  - See discussion in Appendix C.6
Carry-Lookahead Adders

16-bit adder performance = $T_{\text{add}} + \max (P_i, G_i) = 2 + 2 + 1 = 5$
(with thrifty hardware)
Arithmetic for Multimedia

- Graphics and media processing operates on vectors of 8-bit and 16-bit data
  - Use 64-bit adder, with partitioned carry chain
  - Operate on $8 \times 8$-bit, $4 \times 16$-bit, or $2 \times 32$-bit vectors
  - SIMD (single-instruction, multiple-data)
- Saturating operations
  - On overflow, result is largest representable value
    - c.f. 2s-complement modulo arithmetic
  - E.g., clipping in audio, saturation in video
Shifters

- Two kinds:
  - Logical: value shifted in is always “0”
  
  ![Logical Shift Diagram]

- Arithmetic: sign-extend on right shifts

  ![Arithmetic Shift Diagram]

- What about n-bit, rather than 1-bit, shifts?
  
  Want a fast shifter
Combinatorial Shifter from MUXes

Basic Building Block

\[
\text{sel} \quad 1 \quad 0 \\
A \quad B \\
D
\]

- What comes in the MSBs?
- How many levels for 32-bit shifter?
- What if we use 4-1 Muxes?

8-bit right shifter

\[
\begin{array}{ccccccccc}
A_7 & A_6 & A_5 & A_4 & A_3 & A_2 & A_1 & A_0 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
R_7 & R_6 & R_5 & R_4 & R_3 & R_2 & R_1 & R_0 \\
S_2 & S_1 & S_0
\end{array}
\]
### Unsigned Multiplication

#### Paper and pencil example (unsigned):

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>1 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier</td>
<td>1 0 0 1</td>
</tr>
<tr>
<td></td>
<td>1 0 0 0</td>
</tr>
<tr>
<td></td>
<td>0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>1 0 0 0</td>
</tr>
<tr>
<td></td>
<td>0 1 0 0 1 0 0 0</td>
</tr>
</tbody>
</table>

- $m$ bits $\times n$ bits = $m+n$ bit product
- Binary makes it easy:
  - 0: place 0  
    - (0 x multiplicand)
  - 1: place copy  
    - (1 x multiplicand)
Unsigned Combinatorial Multiplier

Stage $i$ accumulates $A \times 2^i$ if $B_i = 1$
How Does it Work?

- At each stage shift A left (x2)
- Use next bit of B to determine whether to add in shifted multiplicand
- Accumulate 2n bit partial product at each stage
Sequential Multiplication Hardware

Initially 0

1. Test Multiplier0
   - Multiplier0 = 1
     1a. Add multiplicand to product and place the result in Product register
   - Multiplier0 = 0

2. Shift the Multiplicand register left 1 bit

3. Shift the Multiplier register right 1 bit

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

Done

Multiply

Shift left

64 bits

64-bit ALU

Product

Write

Control test

Initially 0

Multiplier

Shift right

32 bits
Observations

- One clock per multiply cycle
  - ~32 clock cycles per integer multiply
  - Vs. one cycle for an add/subtract
- Half of the bits in the multiplicand are always zero
  - 64-bit adder is wasted
- Zeros inserted in left of multiplicand as shifted
  - Least significant bits of product unchanged once formed

- Instead of shifting multiplicand to left, shift product to right!
Optimized Multiplier

- Perform steps in parallel: add/shift

- One cycle per partial-product addition
  - That’s ok, if frequency of multiplications is low
Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoff

- Can be pipelined
  - Several multiplication performed in parallel
MIPS Multiplication

- Two 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits

- Instructions
  - `mult 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

- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 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 remainder
Division Hardware

1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register

2. Test Remainder
   - If Remainder ≥ 0, proceed with the next step
   - If Remainder < 0, shift the Divisor register to the left, setting the new rightmost bit to 1

3. Shift the Divisor register right 1 bit

No: < 33 repetitions
   - Shift right 64 bits
   - 64-bit ALU
   - Write 64 bits
   - Control test
   - Quotient Shift left 32 bits

Yes: 33 repetitions
   - Done

Initially divisor in left half

Initially dividend
Optimized Divider

- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
  - Same hardware can be used for both
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
  - Still require multiple steps
MIPS Division

- Use HI/LO registers for result
  - HI: 32-bit remainder
  - LO: 32-bit quotient

- Instructions
  - `div rs, rt` / `divu rs, rt`
  - No overflow or divide-by-0 checking
    - Software must perform checks if required
  - Use `mfhi`, `mflo` to access result
Next Lecture

- Floating point
  - Rest of Chapter 3