MIPS

- 32 bit signed numbers:
  - 0000 0000 0000 0000 0000 0000 0000 0000 = 0	ex{ten}
  - 0000 0000 0000 0000 0000 0000 0000 0001 = + 1	ex{ten}
  - 0000 0000 0000 0000 0000 0000 0000 0010 = + 2	ex{ten}
  - ... 
  - 0111 1111 1111 1111 1111 1111 1111 1110 = + 2,147,483,646	ex{ten}
  - 0111 1111 1111 1111 1111 1111 1111 1111 = + 2,147,483,647	ex{ten}
  - 1000 0000 0000 0000 0000 0000 0000 0000 = \minint
  - 1000 0000 0000 0000 0000 0000 0000 0001 = – \minint
  - 1000 0000 0000 0000 0000 0000 0000 0010 = – \minint
  - ... 
  - 1111 1111 1111 1111 1111 1111 1111 1101 = – 3	ex{ten}
  - 1111 1111 1111 1111 1111 1111 1111 1110 = – 2	ex{ten}
  - 1111 1111 1111 1111 1111 1111 1111 1111 = – 1	ex{ten}

One-Bit Adder

- Takes three input bits and generates two output bits
- Multiple bits can be cascaded

Detecting Overflow

- No overflow when adding a +ve and a -ve number
- No overflow when signs are the same for subtraction
- Overflow occurs when the value affects the sign:
  - overflow when adding two +ves yields a -ve
  - or, adding two -ves gives a +ve
  - or, subtract a -ve from a +ve and get a -ve
  - or, subtract a +ve from a -ve and get a +ve
- Consider the operations A + B, and A – B
  - Can overflow occur if B is 0?
  - Can overflow occur if A is 0?

Effects of Overflow

- An exception (interrupt) occurs
  - Control jumps to predefined address for exception
  - Interrupted address is saved for resumption
- Details based on software system / language
  - example: flight control vs. homework assignment
- Don’t always want to detect overflow
  - new MIPS instructions: addu, addiu, subu
    - note: addiu still sign-extends!
    - note: sltu, sltiu for unsigned comparisons

An ALU (arithmetic logic unit)

- Let’s build an ALU to support
  - andi and ori instructions
  - we’ll just build a 1 bit ALU, and replicate
    - operation
      - op a b res
    - Possible Implementation (sum-of-products):

Different Implementations

- Not easy to decide the “best” way to build something
  - Don’t want too many inputs to a single gate
  - Don’t want to have to go through too many gates
  - for our purposes, ease of comprehension is important
- Let’s look at a 1-bit ALU for addition:
  - How could we build a 1-bit ALU for add, and, or?
  - How could we build a 32-bit ALU?
Building a 32 bit ALU

What about subtraction \((a - b)\)?

- Two's complement approach: just negate \(b\) and add.
- How do we negate?

- A very clever solution:

Tailoring the ALU to the MIPS

- Need to support the set-on-less-than instruction (slt)
  - remember: slt is an arithmetic instruction
  - produces a 1 if \(rs < rt\) and 0 otherwise
  - use subtraction: \((a-b) < 0\) implies \(a < b\)
- Need to support test for equality (beq $t5, $t6, $t7)
  - use subtraction: \((a-b) = 0\) implies \(a = b\)

Supporting slt

- Can we figure out the idea?

A 32-bit ALU

- A Ripple carry ALU
- Two bits decide operation
  - Add/Sub
  - AND
  - OR
  - LESS
- 1 bit decide add/sub operation
- A carry in bit
- Bit 31 generates overflow and set bit

Test for equality

- Notice control lines:
  000 = and
  001 = or
  010 = add
  110 = subtract
  111 = slt

*Note: zero is a 1 when the result is zero!
**Problem:** ripple carry adder is slow

- Is a 32-bit ALU as fast as a 1-bit ALU?
- Is there more than one way to do addition?
  - two extremes: ripple carry and sum-of-products

Can you see the ripple? How could you get rid of it?

\[
\begin{align*}
    c_1 &= b_0c_0 + a_0c_0 + a_0b_0 \\
    c_2 &= b_1c_1 + a_1c_1 + a_1b_1 \\
    c_3 &= b_2c_2 + a_2c_2 + a_2b_2 \\
    c_4 &= b_3c_3 + a_3c_3 + a_3b_3 \\
\end{align*}
\]

Not feasible! Why?

---

**Carry-look-ahead adder**

- An approach in-between our two extremes
- Motivation:
  - If we didn’t know the value of carry-in, what could we do?
  - When would we always generate a carry? \( g_i = a_i \cdot b_i \)
  - When would we propagate the carry? \( p_i = a_i + b_i \)
  - Did we get rid of the ripple?

\[
\begin{align*}
    c_1 &= g_0 + p_0c_0 \\
    c_2 &= g_1 + p_1c_1 \\
    c_3 &= g_2 + p_2c_2 \\
    c_4 &= g_3 + p_3c_3 \\
\end{align*}
\]

Feasible! Why?

---

**A 4-bit carry look-ahead adder**

- Generate g and p term for each bit
- Use g’s, p’s and carry in to generate all C’s
- Also use them to generate block G and P
- CLA principle can be used recursively

---

**Use principle to build bigger adders**

- A 16 bit adder uses four 4-bit adders
- It takes block g and p terms and cin to generate block carry bits out
- Block carries are used to generate bit carries
  - could use ripple carry of 4-bit CLA adders
  - Better: use the CLA principle again!

---

**Delays in carry look-ahead adders**

- 4-Bit case
  - Generation of g and p: 1 gate delay
  - Generation of carries (and G and P): 2 gate delay
  - Generation of sum: 1 more gate delay
- 16-Bit case
  - Generation of g and p: 1 gate delay
  - Generation of block G and P: 2 more gate delay
  - Generation of block carries: 2 more gate delay
  - Generation of bit carries: 2 more gate delay
  - Generation of sum: 1 more gate delay
- 64-Bit case
  - 12 gate delays

---

**What is Realistic Delay**

- Can we use carry look ahead for all sizes
- Probably not due to large sizes of gate required
- What about 64 bit adders
- Use 8 bit blocks
- Eight blocks will make 64 bits
- What about 32 bits?
- Compare design using 4 bit and 8 bit blocks
- Any creative thinking?
Multiplication

- More complicated than addition - accomplished via shifting and addition
- More time and more area
- Let's look at 3 versions based on grade school algorithm

\[
\begin{align*}
\text{Multiplicand:} & \quad 01010010 \\
\times \text{Multiplier:} & \quad 01101101
\end{align*}
\]

- Negative numbers: convert and multiply
- Use other better techniques like Booth's encoding

Multiplication: Implementation

Second Version

Final Version

Multiplication Example
Signed Multiplication

• Let Multiplier be \( Q^{n-1:0} \), multiplicand be \( M^{n-1:0} \)
• Let \( F = 0 \) (shift flag)
• Let result \( A^{n-1:0} = 0\ldots00 \)
• For \( n-1 \) steps do
  – \( A^{n-1:0} = A^{n-1:0} + M^{n-1:0} \times Q^0 \) /* add partial product */
  – \( F \leq F .or. (M^{n-1} .and. Q^0) \) /* determine shift bit */

Booth’s Encoding

• Numbers represented using three symbols, 1, 0, & -1
• Let us consider -1 in 8 bits
  – One representation is 1 1 1 1 1 1 1 1
  – Another possible one 0 0 0 0 0 0 0 -1

Carry-Save Addition

• Consider adding six set of numbers (4 bits each in the example)
• The numbers are 1001, 0110, 1111, 0111, 1010, 0110 (all positive)
• One way is to add them pair wise, getting three results, and then adding them again

Other method is add them three at a time by saving carry

Booth’s algorithm (Neg. multiplier)

<table>
<thead>
<tr>
<th>Itera-tion</th>
<th>multiplier</th>
<th>Booth’s algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0010</td>
<td>Initial values</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>1c: 10 &amp; prod + Prod - Mxand</td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>2: Shift right Prod</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>1b: 00 &amp; prod + Prod - Mxand</td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>2: Shift right Prod</td>
</tr>
<tr>
<td>3</td>
<td>0010</td>
<td>1c: 10 &amp; prod + Prod - Mxand</td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>2: Shift right Prod</td>
</tr>
<tr>
<td>4</td>
<td>0010</td>
<td>1d: 11 &amp; no operation</td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>2: Shift right Prod</td>
</tr>
</tbody>
</table>

Carry-Save Addition for Multiplication

• n-bit carry-save adder take 1FA time for any n
• For \( n \times n \) bit multiplication, \( n \) or \( n/2 \) (for 2 bit at time Booth’s encoding) partial products can be generated
• For \( n \) partial products, need \( n/3 \) n-bit carry save adders
• This yields \( 2n/3 \) partial results
• Repeat this operation until only 2 partial results remain
• Add them using a regular adder to obtain 2n bits
• For \( n = 32 \), you need 30 carry save adders in eight stages taking 8T time where T is time for one-bit full adder
• You need one carry-propagate/carry-look-ahead adder
Even more complicated can be accomplished via shifting and addition/subtraction. More time and more area. We will look at 3 versions based on grade school algorithm.

Division

Division, First Version

Division, Second Version

Division, Final Version

Restoring Division

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Divisor</th>
<th>Divide algorithm</th>
<th>Step</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0010</td>
<td>Initial values</td>
<td></td>
<td>0000 0111</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>2. Shift Rem left</td>
<td></td>
<td>1110 1110</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2b. Rem = Rem - Div</td>
<td>1</td>
<td>0001 1100</td>
</tr>
<tr>
<td>3</td>
<td>0010</td>
<td>2b. Rem = Rem - Div</td>
<td>1111 1100</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0010</td>
<td>2. Shift Rem left</td>
<td></td>
<td>1111 1100</td>
</tr>
<tr>
<td>Done</td>
<td>0010</td>
<td>Shift left half of Rem right</td>
<td></td>
<td>0001 0111</td>
</tr>
</tbody>
</table>

Non-Restoring Division

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Divisor</th>
<th>Divide algorithm</th>
<th>Step</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0010</td>
<td>Initial values</td>
<td></td>
<td>0000 1110</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>2b. Rem = Rem - Div</td>
<td>1</td>
<td>1111 1100</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2b. Rem = Rem + Div</td>
<td>0001 1000</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0010</td>
<td>2a. Rem = Rem - Div</td>
<td>0011 0001</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0010</td>
<td>2a. Rem = Rem - Div</td>
<td>1101 0001</td>
<td></td>
</tr>
<tr>
<td>Done</td>
<td>0010</td>
<td>Shift left half of Rem right</td>
<td>0011 0001</td>
<td></td>
</tr>
</tbody>
</table>