Chapter 3

ALU Design

1-Bit ALU with Add, Or, And

- Multiplexor selects between Add, Or, And operations.
32-bit Ripple Carry Adder

- 1-bit ALUs are connected “in series” with the carry-out of 1 box going into the carry-in of the next box.

Incorporating Subtraction

- Must invert bits of B and add a 1. Include an inverter.
- CarryIn for the first bit is 1.
- The CarryIn signal (for the first bit) can be the same as the Binvert signal.
Incorporating NOR and NAND

Incorporating SLT

- Perform \( a - b \) and check the sign.
- New signal (Less) that is zero for ALU boxes 1-31.
- The 31\(^{st}\) box has a unit to detect overflow and sign – the sign bit serves as the Less signal for the 0\(^{th}\) box.
Incorporating BEQ

- Perform \( a - b \) and confirm that the result is all zero’s.

Control Lines
Control Lines

- What are the values of the control lines and what operations do they correspond to?

<table>
<thead>
<tr>
<th>Ai</th>
<th>Bn</th>
<th>Op</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>00</td>
</tr>
</tbody>
</table>

Speed of Ripple Carry

- The carry propagates through every 1-bit box: each 1-bit box sequentially implements AND and OR – total delay is the time to go through 64 gates!
- You know that any logic equation can be expressed as the sum of products – so it should be possible to compute the result by going through only 2 gates!
- Caveat: need many parallel gates and each gate may have a very large number of inputs – it is difficult to efficiently build such large gates, so we’ll find a compromise:
  - Moderate number of gates.
  - Moderate number of inputs to each gate.
  - Moderate number of sequential gates traversed.
Computing CarryOut

\[
\begin{align*}
\text{CarryIn1} &= b_0.\text{CarryIn0} + a_0.\text{CarryIn0} + a_0.b_0 \\
\text{CarryIn2} &= b_1.\text{CarryIn1} + a_1.\text{CarryIn1} + a_1.b_1 \\
&= b_1.b_0.c_0 + b_1.a_0.c_0 + b_1.a_0.b_0 + a_1.b_0.c_0 + a_1.a_0.c_0 + a_1.a_0.b_0 + a_1.b_1 \\
&\cdots \\
\text{CarryIn32} &= \text{a really large sum of really large products.}
\end{align*}
\]

- Potentially fast implementation as the result is computed by going thru just 2 levels of logic – unfortunately, each gate is enormous and slow.

Generate and Propagate

Equation re-phrased:
\[
\begin{align*}
\text{Ci+1} &= a_i.b_i + a_i.C_i + b_i.C_i \\
&= (a_i.b_i) + (a_i + b_i).C_i
\end{align*}
\]

Stated verbally, the current pair of bits will \textit{generate} a carry if they are both 1 and the current pair of bits will \textit{propagate} a carry if either is 1

Generate signal = \text{ai}.bi \\
Propagate signal = \text{ai} + \text{bi}

Therefore, \text{Ci+1} = \text{Gi} + \text{Pi} \cdot \text{Ci}
Generate and Propagate

c1 = g0 + p0.c0

\[ c2 = g1 + p1.c1 \]
\[ = g1 + p1.g0 + p1.p0.c0 \]

\[ c3 = g2 + p2.g1 + p2.p1.g0 + p2.p1.p0.c0 \]

\[ c4 = g3 + p3.g2 + p3.p2.g1 + p3.p2.p1.g0 + p3.p2.p1.p0.c0 \]

Either,
a carry was just generated, or
a carry was generated in the last step and was propagated, or
a carry was generated two steps back and was propagated by both the next two stages, or
a carry was generated N steps back and was propagated by every single one of the N next stages

Divide and Conquer

- The equations on the previous slide are still difficult to implement as logic functions – for the 32nd bit, we must AND every single propagate bit to determine what becomes of c0 (among other things).
- Hence, the bits are broken into groups (of 4) and each group computes its group-generate and group-propagate.
- For example, to add 32 numbers, you can partition the task as a tree.
P and G for 4-bit Blocks

- Compute P0 and G0 (super-propagate and super-generate) for the first group of 4 bits (and similarly for other groups of 4 bits)
  \[ P_0 = p_0.p_1.p_2.p_3 \]
  \[ G_0 = g_3 + g_2.p_3 + g_1.p_2.p_3 + g_0.p_1.p_2.p_3 \]

- Carry out of the first group of 4 bits is
  \[ C_1 = G_0 + P_0.c_0 \]
  \[ C_2 = G_1 + P_1.G_0 + P_1.P_0.c_0 \]
  ...

- By having a tree of sub-computations, each AND, OR gate has few inputs and logic signals have to travel through a modest set of gates (equal to the height of the tree).

Example

<table>
<thead>
<tr>
<th>Add</th>
<th>A</th>
<th>0001</th>
<th>1010</th>
<th>0011</th>
<th>0011</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>B</td>
<td>1110</td>
<td>0101</td>
<td>1110</td>
<td>1011</td>
</tr>
<tr>
<td>g</td>
<td>0000</td>
<td>0000</td>
<td>0010</td>
<td>0011</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>1111</td>
<td>1111</td>
<td>1111</td>
<td>1011</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>C4 = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[Table showing addition and logic computations for two values A and B, along with corresponding P, G, and C4 values]
Carry Look-Ahead Adder

- 16-bit Ripple-carry takes 32 steps.
- This design takes how many steps?

Review - A MIPS ALU Implementation

- Zero detect (slt, slti, sltiu, sltu, beq, bne)
- Overflow bit setting for signed arithmetic (add, addi, sub)
Implementation Overview

- We need memory
  - to store instructions
  - to store data
  - for now, let’s make them separate units

- We need registers, ALU, and a whole lot of control logic

- CPU operations common to all instructions:
  - use the program counter (PC) to pull instruction out of instruction memory
  - read register values

Big Picture

- What is the role of the Add units?
- Explain the inputs to the data memory unit.
- Explain the inputs to the ALU.
- Explain the inputs to the register unit.

Note: multiplexers not shown.
Clocking Methodology

• Which of the above units need a clock?
• What is being saved (latched) on the rising edge of the clock? Keep in mind that the latched value remains there for an entire cycle.