Arithmetic Logic Unit (ALU)

ALU operation (2-bit):

00 = and
01 = or
10 = add
What about subtraction \((a - b)\)?

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

**Invert all bits of b**

**Add 1**

**ALU operation (3-bit):**

<table>
<thead>
<tr>
<th>Binvert</th>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>and</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>or</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>add</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>sub</td>
</tr>
</tbody>
</table>
Tailoring the ALU to the MIPS datapath

- Need to support the set-on-less-than instruction
  \[ \text{slt } rd, rs, rt \]
  - \text{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, label)
  - use subtraction: \( (a-b) = 0 \) implies \( a = b \) \( \Rightarrow \) \( \text{Zero}=1 \)
Supporting slt

(a) Operation
(b) Set Overflow detection
Test for equality and complete ALU (3-bit control)

ALU operation:
000 = and
001 = or
010 = add
110 = subtract
111 = slt

Control lines

<table>
<thead>
<tr>
<th>Bnegate</th>
<th>Operation</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>and</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>or</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>add</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>sub</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>slt</td>
</tr>
</tbody>
</table>

• Note: zero is a 1 when the result is zero!
Adding NOR and NAND operations
Final ALU (4-bit control)

<table>
<thead>
<tr>
<th>Ainvert</th>
<th>Bnegate</th>
<th>Operation</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>and</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>or</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>10</td>
<td>add</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>10</td>
<td>sub</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>slt</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>00</td>
<td>nor</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01</td>
<td>nand</td>
</tr>
</tbody>
</table>
Conclusion

- We can build an ALU to support the MIPS instruction set
  - key idea: use multiplexor to select the output we want
  - we can efficiently perform subtraction using two's complement
  - we can replicate a 1-bit ALU to produce a 32-bit ALU
- Important points about hardware
  - all of the gates are always working
  - the speed of a gate is affected by the number of inputs to the gate
  - the speed of a circuit is affected by the number of gates in series
    (on the “critical path” or the “deepest level of logic”)
- Our primary focus: comprehension, however,
  - Clever changes to organization can improve performance
    (similar to using better algorithms in software)
  - we’ll look at two examples for addition and multiplication

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-lookahead 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? \( g_i = a_i b_i \)
  - When would we propagate the carry? \( p_i = a_i + b_i \)
- Did we get rid of the ripple?

\[ c_1 = g_0 + p_0 c_0 \]
\[ c_2 = g_1 + p_1 c_1 \quad \quad \quad \quad c_2 = \]
\[ c_3 = g_2 + p_2 c_2 \quad \quad \quad \quad c_3 = \]
\[ c_4 = g_3 + p_3 c_3 \quad \quad \quad \quad c_4 = \]

Feasible! Why?

Use principle to build bigger adders

- Can’t build a 16 bit adder this way… (too big)
- Could use ripple carry of 4-bit CLA adders
- Better: use the CLA principle again!