C335
Computer Structures

ALU Design (I)

Dr. David R. Surma

Department of Computer and Information Sciences
Adapted from Morgan Kaufmann, Dr. L. Zhang and others
The Design is to Represent

(1) **Functional Specification**

**Inputs:** 2 x 16 bit operands- A, B; 1 bit carry input- Cin, 3 bit mode/function.

**Outputs:** 1 x 16 bit result- S; 1 bit carry output- Co.

**Operations:** SLT, ADD (A plus B plus Cin), SUB (A minus B minus Cin), AND, XOR, OR, COMPARE (equality)

**Performance:** left unspecified for now!

(2) **Block Diagram**

Understand the data and control flows
Review: Two’s Complement Arithmetic

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Decimal</th>
<th>2’s Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>-1</td>
<td>1111</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>-2</td>
<td>1110</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>-3</td>
<td>1101</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>-5</td>
<td>1011</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>-6</td>
<td>1010</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>-7</td>
<td>1001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-8</td>
<td>1000</td>
</tr>
</tbody>
</table>

- **Examples:**
  - $7 - 6 = 7 + (-6) = 1$
  - $3 - 5 = 3 + (-5) = -2$
Block Diagram of the ALU

- ALU Control Lines (ALUop)  Function
  - 000  And
  - 001  Or
  - 010  Add
  - 110  Subtract
  - 111  Set-on-less-than
4 Hardware Building Blocks

- **AND gate** \( (c = a \cdot b) \)

  \[
  \begin{array}{ccc}
  a & b & c = a \cdot b \\
  0 & 0 & 0 \\
  1 & 0 & 0 \\
  0 & 1 & 0 \\
  1 & 1 & 1 \\
  \end{array}
  \]

- **OR gate** \( (c = a + b) \)

  \[
  \begin{array}{ccc}
  a & b & c = a + b \\
  0 & 0 & 0 \\
  1 & 0 & 1 \\
  0 & 1 & 1 \\
  1 & 1 & 1 \\
  \end{array}
  \]

- **Inverter** \( (c = a') \)

  \[
  \begin{array}{c}
  a \rightarrow 0 \rightarrow c
  \end{array}
  \]

- **Multiplexer**
  
  if \( d == 0 \), \( c = a \);
  
  otherwise \( c = b \)

  \[
  \begin{array}{ccc}
  d & c \\
  0 & a \\
  1 & b \\
  \end{array}
  \]
This 1-bit ALU will perform AND, OR, and ADD.
Review: A One-bit Full Adder

- **1 bit full adder**: a switching circuit which add together two binary digits (bits), and a third bit called a *CarryIn* bit which may have come from a previous full adder.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong></td>
<td><strong>B</strong></td>
<td><strong>CarryIn</strong></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Review: A One-bit Full Adder

- Sum = A ⊕ B ⊕ CarryIn
- CarryOut = B • CarryIn + A • CarryIn + A • B
A 4-bit ALU

1-bit ALU

B

A

1-bit Full Adder

CarryOut

ALUop

Mux

Result

4-bit ALU

A0

B0

A1

B1

A2

B2

A3

B3

1-bit ALU

Result0

1-bit ALU

Result1

1-bit ALU

Result2

1-bit ALU

Result3
How About Subtraction?

- Keep in mind the followings:
  - $(A - B)$ is the same as $A + (-B)$
  - 2’s Complement: Take the inverse of every bit and add 1

- Bit-wise inverse of $B$ is $B'$:
  - $A + B' + 1 = A + (B' + 1) = A + (-B) = A - B$

![Diagram of ALU with 2x1 Mux and Subtract function]
### Overflow

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Decimal</th>
<th>2’s Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>-1</td>
<td>1111</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>-2</td>
<td>1110</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>-3</td>
<td>1101</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>-5</td>
<td>1011</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>-6</td>
<td>1010</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>-7</td>
<td>1001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-8</td>
<td>1000</td>
</tr>
</tbody>
</table>

- Examples: $7 + 3 = 10$ but ...

- $-4 - 5 = -9$ but ...

- Table showing decimal and binary representations:

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Decimal</th>
<th>2’s Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>-1</td>
<td>1111</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>-2</td>
<td>1110</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>-3</td>
<td>1101</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>-5</td>
<td>1011</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>-6</td>
<td>1010</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>-7</td>
<td>1001</td>
</tr>
</tbody>
</table>

- Diagram showing addition and subtraction examples.
Overflow Detection

- Overflow: the result is too large (or too small) to represent properly
  - Example: -8 ≤ 4-bit binary number ≤ 7

- Can overflow happen when adding operands with different signs?

- Overflow occurs when adding:
  - 2 positive numbers and the sum is negative
  - 2 negative numbers and the sum is positive

- Exercise: Prove you can detect overflow by:
  - Carry into MSB ≠ Carry out of MSB
Overflow Detection Logic

- **Carry into MSB ! = Carry out of MSB**
  - For a N-bit ALU: Overflow = CarryIn[N - 1] XOR CarryOut[N - 1]

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X XOR Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

![Diagram of 1-bit ALUs and XOR gate for overflow detection]
Zero Detection Logic

- A = B is the same as A - B = 0
- Zero Detection Logic is just a one BIG NOR gate
  - Any non-zero input to the NOR gate will cause its output to be zero

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c = a NOR b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

![Diagram of Zero Detection Logic]

1-bit ALU

- A0 → B0
- CarryIn0 → Result0
- CarryIn1 → CarryOut0

1-bit ALU

- A1 → B1
- CarryIn1 → Result1
- CarryIn2 → CarryOut1

1-bit ALU

- A2 → B2
- CarryIn2 → Result2
- CarryIn3 → CarryOut2

1-bit ALU

- A3 → B3
- CarryIn3 → Result3
- CarryOut3 → Zero
The Disadvantage of Ripple Carry

The adder we just built is called a “Ripple Carry Adder”

- The carry bit may have to propagate from LSB to MSB
- Worst case delay for a N-bit adder: 2N-gate delay
Consider building a 8-bit ALU

- Simple: connects two 4-bit ALUs in series
Consider building a 8-bit ALU

- Expensive but faster: uses three 4-bit ALUs
The Theory Behind Carry Lookahead

Recall: CarryOut = (B • CarryIn) + (A • CarryIn) + (A • B)
- Cin2 = Cout1 = (B1 • Cin1) + (A1 • Cin1) + (A1 • B1)
- Cin1 = Cout0 = (B0 • Cin0) + (A0 • Cin0) + (A0 • B0)

Substituting Cin1 into Cin2:
- Cin2 = (A1 • A0 • B0) + (A1 • A0 • Cin0) + (A1 • B0 • Cin0) + (B1 • A0 • B0) + (B1 • B0 • Cin0) + (B1 • A0 • Cin0) + (A1 • B1)

Now define two new terms:
- Generate Carry at Bit i \( g_i = A_i \cdot B_i \)
- Propagate Carry via Bit i \( p_i = A_i + B_i \)
The Theory Behind Carry Lookahead (Continue)

- Using the two new terms we just defined:
  - Generate Carry at Bit i \( g_i = A_i \cdot B_i \)
  - Propagate Carry via Bit i \( p_i = A_i + B_i \)

- We can rewrite:
  - \( C_{in1} = g_0 + (p_0 \cdot C_{in0}) \)
  - \( C_{in2} = g_1 + (p_1 \cdot g_0) + (p_1 \cdot p_0 \cdot C_{in0}) \)
  - \( C_{in3} = g_2 + (p_2 \cdot g_1) + (p_2 \cdot p_1 \cdot g_0) + (p_2 \cdot p_1 \cdot p_0 \cdot C_{in0}) \)

- Carry going into bit 3 is 1 if
  - We generate a carry at bit 2 (\( g_2 \))
  - Or we generate a carry at bit 1 (\( g_1 \)) and bit 2 allows it to propagate (\( p_2 \& g_1 \))
  - Or we generate a carry at bit 0 (\( g_0 \)); and bit 1 as well as bit 2 allows it to propagate (\( p_2 \& p_1 \& g_0 \))
  - Or we have a carry input at bit 0 (\( C_{in0} \)); and bit 0, 1, and 2 all allow it to propagate (\( p_2 \& p_1 \& p_0 \& C_{in0} \))
The Theory Behind Carry Lookahead (Continue)
A Partial Carry Lookahead Adder

- It is very expensive to build a “full” carry lookahead adder
  - Just imagine the length of the equation for Cin31

Common practices:
- Connects several N-bit Lookahead Adders to form a big adder
- Example: connects four 8-bit carry lookahead adders to form a 32-bit partial carry lookahead adder
Summary

- An Overview of the Design Process
  - Design is an iterative process—successive refinement
  - Do NOT wait until you know everything before you start

- Binary Arithmetics
  - If you use 2’s complement representation, subtract is easy.

- ALU Design
  - Designing a Simple 4-bit ALU
    - How to implement SLT operation?
  - Other ALU Construction Techniques