Arithmetic/Logical Unit (ALU)

Digital Computer Design
Arithmetic Circuits

• Arithmetic circuits are the central building blocks of computers.
• Computers and digital logic perform many arithmetic functions:
  —addition, subtraction, comparisons, shifts, multiplication and division
1-Bit Adders

**Half Adder**

\[
\begin{array}{c c c c}
A & B & C_{out} & S \\
0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 \\
1 & 1 & 1 & 0 \\
\end{array}
\]

\[S = A \oplus B\]

\[C_{out} = AB\]

**Full Adder**

\[
\begin{array}{c c c c c c}
C_{in} & A & B & C_{out} & S \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

\[S = A \oplus B \oplus C_{in}\]

\[C_{out} = AB + AC_{in} + BC_{in}\]
Multibit Adders - Carry Propagate Adders (CPAs)

• An **N-bit adder** sums two *N-bit* inputs A and B, and a carry in $C_{in}$ to produce an *N-bit* result $S$ and a carry out $C_{out}$.

• Multibit adder is commonly called a **carry propagate adder** (CPA)
  – because the carry out of one bit propagates into the next bit.
CPAs: Ripple-Carry Adder

• Chain together N full adders.
  – The $C_{out}$ of one stage acts as the $C_{in}$ of the next stage

• The ripple-carry adder is **slow** when N is large.
  – For an 32-bit adder, $S_{31}$ depends on $C_{30}$, which depends on $C_{29}$, which depends on $C_{28}$, and so forth all the way back to $C_{in}$. 
CPAs: Ripple-Carry Adder

- The fundamental reason that large ripple-carry adders are slow is that the carry signals must propagate through every bit in the adder.

\[
\text{tripple} = N t_{FA}
\]

where \( t_{FA} \) is the delay of a 1-bit full adder
Subtractor

• Subtraction is almost as easy:
  – flip the sign of the second number, then add.

  Symbol | Implementation
  ------ | ---------------
  ![Symbol](image1)  
  ![Symbol](image2)  

• Flipping the sign of a two’ s complement number is done by inverting the bits and adding 1.
Comparator: Equality

- A comparator determines whether two binary numbers are equal or if one is greater or less than the other.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A [4] B [4]</td>
<td>Equal</td>
</tr>
<tr>
<td>Equal</td>
<td>Equal</td>
</tr>
</tbody>
</table>

Ex: \(A_3 B_3\), \(A_2 B_2\), \(A_1 B_1\), \(A_0 B_0\)
Comparator: Less Than

• Compute A – B and looking at the sign (most significant bit) of the result.
  – If the result is negative (i.e., the sign bit is 1), then A is less than B. Otherwise A is greater than or equal to B.

• This comparator, however, functions incorrectly upon overflow.
Arithmetic/Logical Unit

- ALU is one of the main components in the microprocessor.
ALU: Arithmetic/Logical Unit

• An Arithmetic/Logical Unit (ALU) combines a variety of mathematical and logical operations into a single unit.
• The ALU forms the heart of most computer systems.

<table>
<thead>
<tr>
<th>ALUControl&lt;sub&gt;1:0&lt;/sub&gt;</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Add</td>
</tr>
<tr>
<td>01</td>
<td>Subtract</td>
</tr>
<tr>
<td>10</td>
<td>AND</td>
</tr>
<tr>
<td>11</td>
<td>OR</td>
</tr>
</tbody>
</table>

The ALU receives a 2-bit control signal ALUControl that specifies which function to perform.
### ALU Implementation

<table>
<thead>
<tr>
<th>ALUControl&lt;sub&gt;1:0&lt;/sub&gt;</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Add</td>
</tr>
<tr>
<td>01</td>
<td>Subtract</td>
</tr>
<tr>
<td>10</td>
<td>AND</td>
</tr>
<tr>
<td>11</td>
<td>OR</td>
</tr>
</tbody>
</table>

![ALU Diagram]

- **ALUControl**: Signal controlling the ALU operation.
- **A**, **B**: Input operands.
- **N**: Negative input for subtraction.
- **Sum**: Result of addition or subtraction.
- **C<sub>out</sub>**: Carry out signal.
- **Result**: Final output of the ALU.
**ALU Implementation**

<table>
<thead>
<tr>
<th>ALUControl&lt;sub&gt;1:0&lt;/sub&gt;</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Add</td>
</tr>
<tr>
<td>01</td>
<td>Subtract</td>
</tr>
<tr>
<td>10</td>
<td>AND</td>
</tr>
<tr>
<td>11</td>
<td>OR</td>
</tr>
</tbody>
</table>

**Example:** Perform $A$ OR $B$

$ALU\text{Control}_{1:0} = 11$

Mux selects output of OR gate as $Result$

$Result = A$ OR $B$
**ALU Implementation**

<table>
<thead>
<tr>
<th>ALUControl&lt;sub&gt;1:0&lt;/sub&gt;</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Add</td>
</tr>
<tr>
<td>01</td>
<td>Subtract</td>
</tr>
<tr>
<td>10</td>
<td>AND</td>
</tr>
<tr>
<td>11</td>
<td>OR</td>
</tr>
</tbody>
</table>

**Example: Perform A + B**

\( ALUControl_{1:0} = 00 \)

\( C_{\text{in}} \) to adder = 0

2\textsuperscript{nd} input to adder is \( B \)

Mux selects \textit{Sum} as Result

\( Result = A + B \)
### ALU with Status Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>Result is <strong>Negative</strong></td>
</tr>
<tr>
<td>$Z$</td>
<td>Result is <strong>Zero</strong></td>
</tr>
<tr>
<td>$C$</td>
<td>Adder produces <strong>Carry out</strong></td>
</tr>
<tr>
<td>$V$</td>
<td>Adder <strong>Overflowed</strong></td>
</tr>
</tbody>
</table>

Some ALUs produce extra outputs, called **flags**, that indicate **information** about the ALU output.
ALU with Status Flags
ALU with Status Flags: **Negative**

\[ N = 1 \text{ if:} \]

Result is **negative**

So, \( N \) is connected to most significant bit of Result
ALU with Status Flags: **Zero**

\[ Z = 1 \text{ if: all of the bits of } Result \text{ are 0} \]
ALU with Status Flags: Carry

\[ C = 1 \text{ if:} \]

\[ C_{out} \text{ of Adder is 1 AND} \]

ALU is adding or subtracting (ALUControl is 00 or 01)
ALU with Status Flags: overflow

V = 1 if:
The addition of 2 same-signed numbers produces a result with the opposite sign. (the result is too big to fit in the available digits.)
ALU with Status Flags: \textbf{Overflow}

\textbf{V} = 1 \textbf{if:}

ALU is performing addition or subtraction \((ALUControl_{1} = 0)\)
ALU with Status Flags: **Overflow**

\[ V = 1 \text{ if:} \]

- ALU is performing addition or subtraction \((ALU_{Control_1} = 0)\)
- \(AND\)
- A and Sum have opposite signs
ALU with Status Flags: oVerflow

\[ V = 1 \text { if:} \]
\[ \text{ALU is performing addition or subtraction} \quad (ALUControl_1 = 0) \]
\[ \text{AND} \]
\[ A \text{ and Sum have opposite signs} \]
\[ \text{AND} \]
\[ A \text{ and } B \text{ have same signs upon addition} \quad (ALUControl_0 = 0) \]
\[ \text{OR} \]
\[ A \text{ and } B \text{ have different signs upon subtraction} \quad (ALUControl_0 = 1) \]
Shifters/Rotators

• Shifters and rotators move bits and multiply or divide by powers of 2.
• As the name implies, a shifter shifts a binary number left or right by a specified number of positions.

**Logical shifter:** shifts value to left or right and fills empty spaces with 0’s

— Ex: \(11001 \gg 2 = 00110\)
— Ex: \(11001 \ll 2 = 00100\)
Shifters/Rotators

**Arithmetic shifter:** right shift, fills empty spaces with the **old most significant bit** (msb)
- Ex: \(11001 \ggg 2 = 11110\)
- Ex: \(11001 \lll 2 = 00100\)

**Rotator:** rotates bits in a circle, such that bits shifted off one end are shifted into the other end
- Ex: \(11001 \text{ ROR } 2 = 01110\)
- Ex: \(11001 \text{ ROL } 2 = 00111\)
Example: Logical Shift Right Implementation

- An **N-bit shifter** can be built from $N\ N:1$ multiplexers.
- The input is shifted by 0 to $N-1$ bits, depending on the value of the $\log_2 N$-bit select lines.

Depending on the value of the 2-bit **shift amount** $\text{shamt}_{1:0}$, the output $Y$ receives the input $A$ shifted by 0 to 3 bits.
Further Reading

• You can read Chapter 5 of your book
  – From Section 5.1 to 5.2.5