Introduction

- **Objective**: To learn what operations are performed by the Arithmetic Logic Unit (ALU) and to learn how the MIPS ALU is implemented.

- **Topics**
  - MIPS logical operations
  - Full Adder
  - 1-Bit ALU
  - The design of the MIPS 32-Bit ALU
  - Overflow and Overflow Detection
  - Carry Lookahead
Addition and Subtraction

- Carry-ripple adder

\[
\begin{array}{c}
0000\ 0111 \\
+\ 0000\ 0110 \\hline
0000\ 1101 \\
\end{array}
\]

\[
\begin{array}{c}
0000\ 0111 \\
-\ 0000\ 0110 \\hline
0000\ 0001 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0000\ 0111 & 0000\ 0111 & 0000\ 0110 & 0000\ 0110 \\
+1111\ 1010 & -0000\ 0111 & +1111\ 1001 & -1111\ 1111 \\hline
0000\ 0001 & 0000\ 0001 & 1111\ 1111 & 1111\ 1111 \\
\end{array}
\]
Overflow Detection

• Overflow occurs in the following situations

<table>
<thead>
<tr>
<th>Operation</th>
<th>A</th>
<th>B</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>A+B</td>
<td>&gt;= 0</td>
<td>&gt;= 0</td>
<td>&lt; 0</td>
</tr>
<tr>
<td>A+B</td>
<td>&lt; 0</td>
<td>&lt; 0</td>
<td>&gt;= 0</td>
</tr>
<tr>
<td>A-B</td>
<td>&gt;= 0</td>
<td>&lt; 0</td>
<td>&lt; 0</td>
</tr>
<tr>
<td>A-B</td>
<td>&lt; 0</td>
<td>&gt;= 0</td>
<td>&gt;= 0</td>
</tr>
</tbody>
</table>
Logical Operations

• Shift left \( \ll \) sll
• Shift right \( \gg \) srl
• Shift right arithmetic sra
• Bitwise and \& and, andi
• Bitwise or | or, ori
• Bitwise complement (not) ~ not (pseudo)
• Exclusive or ^ xor, xori
Representation of Shift Instruction

- Example

```plaintext
sll $t2, $s0, 8      # $t2 = $s0 << 8

$t2 = $10, $s0 = $16
```

```
000000 00000 10000 01010 01000 000000
```

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example use of Logical Operations

```c
Int data;
struct {
    unsigned int ready:
        1;
    unsigned int enable:
        1;
    unsigned int receivedByte:
        8;
} receiver;
...
data = receiver.receivedByte;
receiver.ready = 0;
receiver.enable = 1;
```

- Assume data in $s0
- receiver in $s1

<table>
<thead>
<tr>
<th>receivedByte</th>
<th>enable</th>
<th>ready</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

- sll $s0, $s1, 22
- srl $s0, $s0, 24
- andi $s1, $s1, 0xfffe
- ori $s1, $s1, 0x0002
Building Blocks

1. AND gate (c = a . b)
   \[ c = a \cdot b \]
   \[
   \begin{array}{ccc}
   a & b & c \\
   0 & 0 & 0 \\
   0 & 1 & 0 \\
   1 & 0 & 0 \\
   1 & 1 & 1 \\
   \end{array}
   \]

2. OR gate (c = a + b)
   \[ c = a + b \]
   \[
   \begin{array}{ccc}
   a & b & c \\
   0 & 0 & 0 \\
   0 & 1 & 1 \\
   1 & 0 & 1 \\
   1 & 1 & 1 \\
   \end{array}
   \]

3. Inverter (c = a)
   \[ c = \overline{a} \]
   \[
   \begin{array}{c}
   a \\
   0 \\
   1 \\
   \end{array}
   \]

4. Multiplexor
   (if d = 0, c = a; else c = b)
   \[
   \begin{array}{c}
   d \\
   \hline
   0 & a \\
   1 & b \\
   \end{array}
   \]
Full Adder

- **Sum** = \( \text{parity}(a, b, \text{CarryIn}) \)
  - \( a \oplus b \oplus c + a \cdot b \cdot c \equiv a \oplus b \oplus c \)
- **CarryOut** = \( \text{majority}(a, b, \text{CarryIn}) \)
  - \( b \cdot \text{CarryIn} + a \cdot \text{CarryIn} + a \cdot b + a \cdot b \cdot \text{CarryIn} \equiv \)
  - \( b \cdot \text{CarryIn} + a \cdot \text{CarryIn} + a \cdot b \)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>CarryIn</th>
<th>Sum</th>
<th>CarryOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
One-Bit ALU

Operation

CarryIn

Result

CarryOut

a

b

0

1

2

0

1

2

+
Building a 32-Bit ALU

- Chain 32 1-Bit ALUs
Supporting Subtraction

- Subtraction is equivalent to adding the inverse
  - In two’s complement $a + \overline{b} + 1$
Overflow and SLT

- **Modify last 1-Bit ALU**
  - SLT set if \( a < b \) \( \iff \) \( a - b < 0 \)
  - Check sign bit after subtraction
  - Check overflow in last 1-Bit ALU
  - Need to take overflow into account for SLT
Overflow and SLT

- **Overflow?**
  - How do we check for overflow?
  - What do we do with slt if we encounter overflow?
Overflow and SLT

• When adding two positive numbers and a final carry out is produced, is this overflow?
  – Yes! This is intuitive.
• When adding two negative numbers and a final carry out is produced, is this overflow?
  – No! Recall 2’s complement sign extend proof and the \((x + -x)\) example.
  – A final carry out is expected.
• Notice the carry in to the most significant (sign) bit ALU in each case and derive an overflow formula that will work in all cases.
• When overflow occurs, what happens to the SLT signal? Derive this as well.
32-Bit ALU with Sub and Slt

```
<table>
<thead>
<tr>
<th>Binvert</th>
<th>CarryIn</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0</td>
<td>CarryIn</td>
<td>ALU0</td>
</tr>
<tr>
<td>b0</td>
<td></td>
<td>Less</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CarryOut</td>
</tr>
<tr>
<td>a1</td>
<td>CarryIn</td>
<td>ALU1</td>
</tr>
<tr>
<td>b1</td>
<td></td>
<td>Less</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>CarryOut</td>
</tr>
<tr>
<td>a2</td>
<td>CarryIn</td>
<td>ALU2</td>
</tr>
<tr>
<td>b2</td>
<td></td>
<td>Less</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>CarryOut</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>a31</td>
<td>CarryIn</td>
<td>ALU31</td>
</tr>
<tr>
<td>b31</td>
<td></td>
<td>Less</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Result0: ALU0 Result
Result1: ALU1 Result
Result2: ALU2 Result
Result31: ALU31 Result
Set: Overflow
Overflow: CarryIn
Less: CarryIn
```
Support Beq

- \( a = b \iff a - b = 0 \)

- Zero = (Result31 + \cdots + Result0)
Final 32-Bit ALU

Control Function
000 and
001 or
010 add
110 sub
111 slt
Carry Lookahead Adder

- Why is it easy to add 12345 + 87654 in your head? Why is it harder to add 65345 + 89298?
- If there are no carries (or if we can figure them out a priori), we can compute the additions themselves in parallel.
- As it stands now, we must ripple the carries through each ALU, causing a linear time addition. This extra time is spent propagating the carry out to the next carry in.
- By computing the carry in advance, we can improve addition to logarithmic time.
Carry Lookahead Adder

• In binary addition there are only a few possible cases:
  – $1 + 1$ ($a = b = 1$)
  – $0 + 0$ ($a = b = 0$)
  – $0 + 1$ or $1 + 0$ ($a \text{xor} b$)

• Notice!
  – The $a = b = 1$ case will always produce a carry out ($C_{out} = 1$), regardless of the carry in
  – The $a = b = 0$ case will never produce a carry out ($C_{out} = 0$), regardless of the carry in
  – The $a \text{xor} b$ case will result in $C_{out} = C_{in}$

• The result is the carry status (generate, kill, propagate)
• Carry status signals can be combined (for example, generate followed by a kill is a kill)
Carry Lookahead Adder

- if \((a_{i-1} = b_{i-1} = 0)\) then \(c_i = 0\) "kill"
- if \((a_{i-1} = b_{i-1} = 1)\) then \(c_i = 1\) "generate"
- if \((a_{i-1} \neq b_{i-1})\) then \(c_i = c_{i-1}\) "propagate"

<table>
<thead>
<tr>
<th>(a_{i-1})</th>
<th>(b_{i-1})</th>
<th>Carry Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0 kill</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1 propagate</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0 propagate</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1 generate</td>
</tr>
</tbody>
</table>
Carry Lookahead Adder

- We can logically define these carry status signals as:
  - Generate Signal $G_i = A_i \text{ AND } B_i$
  - Propagate Signal $P_i = A_i \text{ XOR } B_i$
- Now how do we compute Carry Out?
Carry Lookahead Adder

- \( C_{out_i} = G_i \ OR \ (P_i \ AND \ C_{in_i}) \)
- But recall that \( C_{in_i} \) is really the previous \( C_{out_{i-1}} \), and we can compute that as \( G_{i-1} \ OR \ (P_{i-1} \ AND \ C_{in_{i-1}}) \)
- How did we define \( G \) and \( P \)? Can you write the recursive formula in terms of \( A \) and \( B \) only?
- If so, we can compute this fast.
Combined Carry Status

- $x_i = \text{kill}$ if $(a_{i-1} = b_{i-1})$
- $x_i = \text{generate}$ if $(a_{i-1} = b_{i-1})$
- $x_i = \text{propagate}$ if $(a_{i-1} \neq b_{i-1})$

\[
FA_i
\]

\[
FA_{i-1}
\]

\[
\begin{array}{cccc}
\text{otimes} & \text{kill} & \text{propagate} & \text{generate} \\
\text{kill} & \text{kill} & \text{kill} & \text{generate} \\
\text{propagate} & \text{kill} & \text{propagate} & \text{generate} \\
\text{generate} & \text{kill} & \text{generate} & \text{generate} \\
\end{array}
\]

- $y_i = y_{i-1} \otimes x_i = x_1 \otimes \cdots \otimes x_i$
Calculating Carry from Carry Status

Lemma:

1. $y_i = \text{kill} \implies c_i = 0$
2. $y_i = \text{generate} \implies c_i = 1$
3. $y_i = \text{propagate}$ does not occur

Proof:

1. $y_i = \text{kill} \implies x_i = \text{kill} \implies c_i = 0$ or
   - $x_i = \text{propagate}$ and $y_{i-1} = \text{kill}$ and $c_i = \text{majority}(c_{i-1}, a_i, b_i) = c_{i-1} = \text{kill}$ (by induction)
2. $y_i = \text{generate} \implies x_i = \text{generate} \implies c_i = 1$ or
   - $x_i = \text{propagate}$ and $y_{i-1} = \text{generate}$ and $c_i = \text{majority}(c_{i-1}, a_i, b_i) = 1$ (by induction)
3. $y_i = \text{propagate} \implies x_i = \text{propagate}$ and $y_{i-1} = \text{propagate}$, which by induction leads to a contradiction
Characterize Carry in Parallel

- Fast (parallel computation of $y_i$ in log time)

```
x0  x1  x2  x3  x4  x5  x6  x7
[0,1] [1,2] [2,3] [3,4] [4,5] [5,6] [6,7]
[0,2] [0,3] [1,4] [2,5] [3,6] [4,7]
[0,0] [0,1] [0,2] [0,3] [0,4] [0,5] [0,6] [0,7]
```