1 The ALU

- ALU includes *combinational* logic.
  - Combinational logic → a change in inputs directly causes a change in output, after a characteristic delay.
  - Different from *sequential logic* (later section) which only changes on the *clock*.

- Two major components of combinational logic are – multiplexors & decoders.

- 2-input multiplexor (or selector) is implemented with gates below

![Symbol and gate implementation](image-url)
**Multiplexors (MUXes)**

Multiplexors can have any number of inputs (in theory)

Multiplexors can apply to buses → multiplied for many lines.

\[ 3 \times 8 \text{ multiplexor} \]

Example: 1 x 2 multiplexor on 32 bits bus.
Decoders

Each combination of the inputs enables exactly one output.
The ALU

- The ALU provides the basic logical and arithmetic functions: AND, OR plus addition.
- Subtraction in 2's complement → invert +1.
- Shift, multiplication and division are usually outside the basic ALU.

Logical operations

1 bit logical unit for AND/OR operations
1 bit FULL adder (3,2)

```
<table>
<thead>
<tr>
<th>inputs</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

\[
\text{sum} = (\overline{a} \cdot \overline{b} \cdot \overline{Cin}) + (a \cdot \overline{b} \cdot \overline{Cin}) + (a \cdot \overline{b} \cdot \overline{Cin}) + (a \cdot b \cdot Cin) = a \oplus b \oplus Cin
\]

\[
\text{Cout} = (b \cdot Cin) + (a \cdot Cin) + (a \cdot b)
\]

= ((a \oplus b) \cdot Cin) + (a \cdot b)
Full Adder from Half Adders

Half adder

Full adder from 2 half adders + or gate
1 Bit Simple ALU

1 bit simple ALU for logical / arithmetic operations

![1 Bit Simple ALU Diagram]
1 Bit Enhanced ALU

Enhanced for subtraction

2's complement: use \( \text{Cin} = 1 \)
subtraction: \( a + \overline{b} + 1 = a + (\overline{b} + 1) = a + (-b) = a - b \)
Ripple Carry Type Adder

32 bit ADDER with ripple carry:

- **To produce a 32 bit result, we connect 32 single bit units together.**

- **This type of ALU adder is called a *ripple* adder**
  - Carry bits are generated in sequence.
  - Bit 31 result and Cout is not correct until it receives Cin from previous unit, which is not correct until it receives Cin from its previous unit, etc.
  - Total operation time is proportional to word size (here 32).
Carry Lookahead

- Ripple arithmetic operations are too slow for high performance.
- We can calculate all carries in 2-level logic, avoiding the ripple. We know that any logical function can be represented in canonical form (sum of products) but it requires more gates ⇒ too expensive.
  - Carry bit $i$ has two possibilities: either $a \cdot b$ and no carry in, or $a \oplus b$ and carry in. But carry in itself is the same combination of the bits previous to those that created it. Hence two level logic has $2^n$ terms for $n$ bits.

- Practical adders use carry lookahead.
  - Factors out two basic functions which give us the carries:
    - Generate - does bit $i$ create a carry by itself?
    - Propagate - does bit $i$ send a carry ahead to the next position?

$$
g_i = (a_i \cdot b_i) \\
p_i = (a_i \oplus b_i) \\
c_{i+1} = g_i + p_i \cdot c_i$$
Illustration of Carry Lookahead for 4 Bit Adder

\[ c_1 = g_0 + (p_0 \cdot c_0) \]
\[ c_2 = g_1 + (p_1 \cdot g_0) + (p_1 \cdot p_0 \cdot c_0) \]
\[ c_3 = g_2 + (p_2 \cdot g_1) + (p_2 \cdot p_1 \cdot g_0) + (p_2 \cdot p_1 \cdot p_0 \cdot c_0) \]
\[ c_4 = g_3 + (p_3 \cdot g_2) + (p_3 \cdot p_2 \cdot g_1) + (p_3 \cdot p_2 \cdot p_1 \cdot g_0) + (p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0) \]
Carry Lookahead Concept Generalized to Higher Levels

\[ P_i = p_{i0} \cdot p_{i1} \cdot p_{i2} \cdot p_{i3} \]
\[ G_i = g_{i3} + g_{i2} \cdot p_{i3} + g_{i1} \cdot p_{i2} \cdot p_{i3} + g_{i0} \cdot p_{i1} \cdot p_{i2} \cdot p_{i3} \]
\[ C_{i+1} = G_i + P_i \cdot C_i \]

Carry Lookahead on 4-bit units (provides logarithmic complexity)

\[ \text{Cin = C0} \]