Section 3 – Arithmetic Logic Unit

3.1 ALU
**Definition**

- Key processing element of a microprocessor that performs arithmetic and logic operations

**Description**

- Directed by Control Unit, ALU performs operations such as ADD, SUB, NOT, OR, AND, XOR

- Data is inputted from and outputted to the Register Array

- Control Signals from Control Unit determine what type of operation is performed

- Input data consists of two operands: operand A and operand B stored in registers and having n bits

- Output data consists of result S

- ALU also outputs Status Signals such as:
  - Zero (when the result of the operation is 0)
  - Negative (when the operation result is < 0)
  - Carry (when the operation results in carry)
  - Overflow (when the result exceeds the number of bits allocated for its storage)
  - Etc.
3.2 Addition

3.2.1 Ripple Carry Adder (RCA)

RCA Goal:

RCA Cell:
RCA Cell Minimisations and Implementation:

<table>
<thead>
<tr>
<th>(B_i)</th>
<th>(A_i)</th>
<th>(C_{i-1})</th>
<th>(S_i)</th>
<th>(C_i)</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>1</td>
<td>0</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>

\[
A_N \backslash B_N C_{N-1} \begin{array}{c|c|c|c|c}
00 & 01 & 11 & 10 \\
\hline
0 & 1 & 0 & 1 \\
1 & 1 & 0 & 0 \\
\end{array}
\]

\[
S_N = \overline{A_N} \overline{B_N} C_{N-1} + \overline{A_N} B_N \overline{C_{N-1}} + A_N \overline{B_N} C_{N-1} + A_N B_N C_{N-1}
\]

\[
A_N \backslash B_N C_{N-1} \begin{array}{c|c|c|c|c}
00 & 01 & 11 & 10 \\
\hline
0 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 \\
\end{array}
\]

\[
C_N = A_N B_N + A_N C_{N-1} + B_N C_{N-1}
\]
Sum Logic

A₀
B₀
Cᵢ
S₀

Carry Logic

Aᵢ
Bᵢ
Aᵢ
Cᵢ₋₁
Bᵢ
Cᵢ₋₁
Cᵢ

RCA Delay:
Using $d$ as delay per carry bit, an $n$-bit RCA has worst case delay of...

$$T_D = n \times d$$

$d$ will depend on technology used, see e.g.
**Example**

An 8-bit RCA Adder is implemented using 2-input NAND technology. The delay of each NAND gate is given as 500 pS. After what time will a valid result be visible at the output?

**NAND Logic**
Use NAND gates to derive other logic gates and functions…

Each Carry Bit will pass through 4 500pS NAND gates

\[ d = 4 \times 500 = 2000 \text{pS} \]

Adder delay: \[ 2000 \times n = 2000 \times 8 \]

\[ => 16000 \text{ pS or 16nS} \]
3.2.2 Carry Look-ahead Adder (CLA)

CLA Rationale:

- RCA sequentially computes carry bits and based on them addition result bits, causing long delays
- Delay can be reduced by examining all inputs simultaneously and producing the carry bits for each next stage

CLA Principle:

We notice that a carry is produced in stage i if:
- Stage i generates a carry
- Stages i-1 generates a carry and stage i propagates it
- Stage i-2 generates a carry and stages i and i-1 propagate it
- Etc.

A carry is generated only if both $A_i$ and $B_i$ are 1:

- $G_i = A_i B_i$

A carry is propagated if carry is not generated and at least one of $A_i$ and $B_i$ is 1:

- $P_i = \overline{A_i} B_i + A_i \overline{B_i}$

Carry for stage i:

- $C_i = G_i + P_i C_{i-1}$
CLA Cell:

CLA Cell Implementation:

\[ C_{-1} = 0 \]

\[ B_N A_N C_{N-1} \]

\[ P_N G_N S_N \]

\[ B_1 A_1 C_0 \]

\[ P_1 G_1 S_1 \]

\[ B_0 A_0 C_{-1} = 0 \]

\[ P_0 G_0 S_0 \]

\[ B_i A_i C_{i-1} \]

\[ P_i G_i S_i \]

\[ A_i \]

\[ B_i \]

\[ C_{i-1} \]

\[ S_i \]

\[ A_i \]

\[ B_i \]

\[ P_i \]

\[ A_i \]

\[ B_i \]

\[ G_i \]
4-Bit CLA Carry Implementation:

\[ C_0 = G_0 + P_0 C_{-1} \]

\[ C_1 = G_1 + P_1 C_0 \]
\[ = G_1 + P_1 (G_0 + P_0 C_{-1}) \]
\[ = G_1 + P_1 G_0 + P_1 P_0 C_{-1} \]

\[ C_2 = G_2 + P_2 C_1 \]
\[ = G_2 + P_2 (G_1 + P_1 G_0 + P_1 P_0 C_{-1}) \]
\[ = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{-1} \]

\[ C_3 = G_3 + P_3 C_2 \]
\[ = G_3 + P_3 (G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{-1}) \]
\[ = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_{-1} \]

CLA Advantages:
- Any Carry Bit, \( C_i \) requires only 2 gate levels.
- To produce \( G_i \) or \( P_i \) from \( A_i \) and \( B_i \) => 1 level
- Parallelizes operation to improve time.
- Lower delay => faster result

CLA Disadvantages:
- Above \( C_3 \), equations get very complex.
- Gates with high number of inputs slower.
- Additional Logic => Larger Area, More Power
Accumulator

Rationale:

- Many calculations consist of repeated additions and subtractions performed on the result of the previous operation.
- Accumulator (ACC) is a dedicated register used to perform these repeated operations.

Usage:
1. Initialise ACC with base value from memory.
2. Each instruction will then either add or subtract from ACC base value.
3. When finished, Store ACC value to Data Memory.
3.3 Multiplication

3.3.1 Binary Multiplication

Method:
- Multiplication can be performed by treating the multiplier unit as a combinational circuit, multiplicand and multiplier bits as inputs and the bits of the result as outputs
- In general multiplies positive numbers only, however can be extended for any numbers.

<table>
<thead>
<tr>
<th>$A_1$</th>
<th>$A_0$</th>
<th>$B_1$</th>
<th>$B_0$</th>
<th>$P_3$</th>
<th>$P_2$</th>
<th>$P_1$</th>
<th>$P_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</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>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>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
\[ P_0 = A_0 B_0 \]

Homework: minimisation of \( P_1, P_2, P_3 \) and implementation using NAND gates!

\[ P_1 = \overline{A_1 A_0 B_1} + \overline{A_0 B_1 B_0} + \overline{A_1 B_1 B_0} + \overline{A_1 A_0 B_0} \]

\[ P_2 = \overline{A_1 A_0 B_1} + \overline{A_1 B_1 B_0} \]

\[ P_3 = A_1 A_0 B_1 B_0 \]
### 3.3.2 “Add & Shift” Multiplication

**Principle:**
- Multiplication is usually performed by computers by repeating additions
- The principle is based on the “pencil and paper” method that requires the computation of partial results and shifting them before they are added in order to calculate the final result

**“Pencil and Paper” Method:**
- Multiplies positive numbers only
- E.g. $15_{10} \times 13_{10} = 1111_2 \times 1101_2$

| Multiplicand Y | 1 | 1 | 1 | 1 |
| Multiplier X   | 1 | 1 | 0 | 1 |
| Partial results| 0 | 0 | 0 | 0 |
| X*Y            | 1 | 1 | 1 | 1 |

| Partial results | 1 | 1 | 1 | 1 |
| X*Y             | 1 | 1 | 0 | 0 |
| X*Y             | 0 | 0 | 0 | 1 |
| X*Y             | 1 | 1 | 1 | 1 |

- Result verification: $11000011_2 = 195_{10}$
Possible Machine Method:

- Multiplies positive numbers only

- E.g. $15_{10} \times 13_{10} = 1111_2 \times 1101_2$

<table>
<thead>
<tr>
<th>Multiplicand (Y)</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>(+15)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier (X)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>(+13)</td>
</tr>
<tr>
<td>Result (R)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step</th>
<th>C</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Initialisation</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>1 =&gt; ADD (R, Y)</td>
</tr>
<tr>
<td>①</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Right Shift</td>
</tr>
<tr>
<td>②</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Carry</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>③</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Right Shift</td>
</tr>
<tr>
<td>④</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 =&gt; ADD (R, Y)</td>
</tr>
</tbody>
</table>

| Carry | 1 | 1  | 0  | 0  | 0  | 0  | 1  | 1  |                    |

- Result verification: $11000011_2 = 195_{10}$
Hardware for the Machine Method:

Requires:

- Binary adder to add multiplicand to the shifter partial product if multiplier digit is 1
- Shift register to shift partial product to the right
- Shift register to shift multiplier to the right in order to parse all its digits

Observation:

- If the Multiplicand has Md bits and Multiplier has Mp bits, the result will have Md+Mp bits
- The result will be stored in a Md+Mp bit register and will be initialised with 0s
- As repeated additions and shifts are performed on partial results, the result register can be the accumulator (A)
- As the Multiplicand is added to the left-most Md bits of A and A shifts its content Mp times to the right, the right-most Mp bits of A can store the Multiplier
- As A shifts its content to the right, the Multiplier bits are also shifted, enabling its bit that decides whether to add the Multiplicand to the partial result to be always the last bit in A
- This solution saves a register to store and shift the Multiplier
Question: What signal do you think is missing from the diagram?
3.3.3 Multiplication of Negative Numbers

Rationale:
- The already presented multiplication methods deal with positive numbers only
- There is a need to find a solution to multiply numbers regardless of their sign

Sign-Magnitude Representation:
- Includes the positive number in binary (Magnitude) and a sign-bit that can indicate:
  - 0 - positive number
  - 1 - negative number

Principle:
- The product magnitude is the result of the product of the magnitudes of the multiplicand and the multiplier
- The sign of the product ($P_{sb}$) is:
  - Positive if the sign of the multiplier ($X_{sb}$) and multiplicand ($Y_{sb}$) are the same
  - Negative if their signs are different

Implementation:

Disadvantage:
- As numbers are represented in two-complement, they require conversion into sign-magnitude, prior to multiplication and then back to two-complement
- The largest negative numbers cannot be multiplied
3.3.4 Booth Algorithm for Multiplication [2’s comp]

Principle:

- Performs additions and subtractions of the Multiplicand, based on the value of the Multiplier bits
- The algorithm looks at two adjacent bits in the Multiplier in order to decide the operation to be performed
- The Multiplier bits are considered from the least significant bit (right-most) to the most significant bit; by default a 0 will be considered at the right of the least significant bit of the Multiplier
- If Multiplicand has Md bits and Multiplier has Mp bits, the result will be stored in a Md+Mp bit register and will be initialised with 0s
- As repeated operations and shifts are performed on partial results, the result register is the accumulator (A)

Algorithm:

a) Initialise the result register with 0; this will store both partial products and the final result
b) If the Multiplier’s bits to be tested are “10”, SUBTRACT the Multiplicand from the partial product
c) If the Multiplier’s bits to be tested are “01”, ADD the Multiplicand from the partial product
d) If the Multiplier’s bits to be tested are “00” or “11”, DO NOTHING
e) Arithmetically RIGHT SHIFT the partial product
f) Sense the next set of adjacent bits of the Multiplier
g) If there are still Multiplier bits to be sensed continue the algorithm at b)

Example:
- E.g. \(+11\)\(_{10}\) * \(-3\)\(_{10}\) = 01011\(_{2}\) * 11101\(_{2}\)

<table>
<thead>
<tr>
<th>Multiplicand (Y)</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>(+11(_{10}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier (X)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initialisation</td>
</tr>
<tr>
<td>0</td>
<td>0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>1</td>
<td>1 0 1 0 1 0 0 0 0</td>
</tr>
<tr>
<td>①</td>
<td>1 1 0 1 0 1 0 0 0</td>
</tr>
<tr>
<td>②</td>
<td>0 0 0 0 1 0 1 1 0 0</td>
</tr>
<tr>
<td>③</td>
<td>1 0 1 1 1 1 0 0 0 0</td>
</tr>
<tr>
<td>④</td>
<td>1 1 0 1 1 1 1 1 0 0</td>
</tr>
<tr>
<td>⑤</td>
<td>1 0 1 1 1 1 0 0 0 0</td>
</tr>
<tr>
<td>⑥</td>
<td>1 1 0 1 1 1 1 1 0 0</td>
</tr>
<tr>
<td>⑦</td>
<td>1 1 0 1 1 1 1 1 1 0</td>
</tr>
<tr>
<td>⑧</td>
<td>1 1 0 1 1 1 1 1 1 0</td>
</tr>
<tr>
<td>⑨</td>
<td>1 1 1 1 0 1 1 1 1 1</td>
</tr>
</tbody>
</table>

- Result verification: \(1111011111\)\(_{2}\) = \(-33\)\(_{10}\)

Homework: Build the block-level hardware diagram.
3.4 Division

3.4.1 Combinational Circuit-based Division

Homework: Build the combinational circuit that divides 4-bit positive numbers by 2-bit positive numbers.

Hint: See Binary multiplication using combinational circuits.

3.4.2 “Subtract and Shift” Division

Algorithm:

a) Initialise the result register (Accumulator - A) with 0; this will store both the Remainder and the Quotient resulted after division.

b) Copy Dividend in the least significant part of A.

c) If the most significant part of the partial result (AH) is greater or equal to the Divisor, SUBTRACT the Divisor from AH and set the next Quotient bit to 1.

d) If the most significant part of the partial result (AH) is smaller than the Divisor, DO NOTHING and set the next Quotient bit to 0.

e) LEFT SHIFT the partial result, introducing in the least significant position the Quotient bit just determined.

f) If we have performed this algorithm less times than the number of bits the Divisor has, continue the algorithm at b)
Example:

- E.g. $35_{10} / 7_{10} = 100011_2 / 000111_2$

Dividend (A): 100011 (35_{10})
Divisor (B): 000111 (7_{10})

<table>
<thead>
<tr>
<th>Dividend (A)</th>
<th>Remainder (R)</th>
<th>Quotient (Q)</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH / Remainder</td>
<td>AL / Q</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 0 0</td>
<td>1 0 0 0 0 1 1</td>
<td>Left Shift</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 0 1</td>
<td>0 0 0 1 1 0 0</td>
<td>AH&lt;B =&gt; Q_0=0</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 0 1</td>
<td>0 0 0 1 1 0 0</td>
<td>Nop, Left Shift</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 1 0 0</td>
<td>0 0 1 1 0 0 0</td>
<td>AH&lt;B =&gt; Q_0=0</td>
<td></td>
</tr>
<tr>
<td>0 0 0 1 0 0 0</td>
<td>0 1 1 0 0 0 0</td>
<td>Nop, Left Shift</td>
<td></td>
</tr>
<tr>
<td>0 0 1 0 0 0 0</td>
<td>1 1 0 0 0 0 0</td>
<td>AH≥B =&gt; Q_0=1</td>
<td></td>
</tr>
<tr>
<td>0 0 0 1 1 1</td>
<td>SUB(AH, B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 0 1</td>
<td>1 1 1 0 0 0 0</td>
<td>Left Shift</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 1 1</td>
<td>0 0 0 0 0 0 1</td>
<td>AH&lt;B =&gt; Q_0=0</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 1 1 1</td>
<td>0 0 0 0 0 0 0</td>
<td>Nop, Left Shift</td>
<td></td>
</tr>
<tr>
<td>0 0 0 1 1 1</td>
<td>0 0 0 0 0 1 0</td>
<td>AH≥B =&gt; Q_0=1</td>
<td></td>
</tr>
<tr>
<td>0 0 0 1 1 1</td>
<td>SUB(AH, B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 1 0</td>
<td>Left Shift</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 0 0</td>
<td>0 0 0 1 0 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Quotient (Q): 000101 (5_{10})
Remainder (R): 000000 (0_{10})
3.4.3 Multiplication with Reciprocal

\[ \text{Quotient} = \frac{1}{\text{Divisor}} \times \text{Dividend} \]

Hint: Computation of the Reciprocal can be performed using combinational circuits.
3.5 Operations with Floating Point Binary Numbers

3.5.1 Floating Point Representation

Principle:
- Any real number can be represented in the following form:
  \[ A \times 2^a \]
- Where A – mantissa and a – exponent
- Floating point representation requires that the floating point binary numbers are normalised

Normalisation:
- A normalised two-complement number has the following requirements:
  \[-1 \leq A < -\frac{1}{2} \quad \text{or} \quad A = 0 \quad \text{or} \quad \frac{1}{2} \leq A < 1\]

Example:

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0111</td>
<td>0.111*2^0011</td>
</tr>
<tr>
<td>-7</td>
<td>1001</td>
<td>1.001*2^0011</td>
</tr>
<tr>
<td>7/32</td>
<td>0.00111</td>
<td>0.111*2^1110</td>
</tr>
</tbody>
</table>

- Non-normalised numbers:
  \[ 0.001\times2^{0011} \quad 101.11\times2^{1011} \]
3.5.2 Floating Point Addition

Addition Algorithm:
- Lets assume that $A \times 2^a$ and $B \times 2^b$ will be added
- In order for the addition to be performed, the exponents of the two numbers have to be equalised and their mantissas shifted accordingly:
  - If $a > b$, $B$ is right shifted $a - b$ places, obtaining $B'$
  - If $a < b$, $A$ is right shifted $b - a$ places, obtaining $A'$
  - If $a = b$, no shift is required
- The mantissa of the result is obtained:
  - If $a > b$ by adding $A$ and $B'$
  - If $a < b$ by adding $A'$ and $B$
  - If $a = b$ by adding $A$ and $B$
- The exponent of the result is:
  - If $a > b$ is $a$
  - If $a < b$ is $b$
  - If $a = b$ is either $a$ or $b$
- Normalisation of the result may be needed

Example:

$$0.100 \times 2^{0001} + 1.000 \times 2^{0011}$$

$a = 0001$, $b = 0011$;

$a < b$  =>  $A$ right shifted $b - a = 2$ places

$$C = A' + B = 0.001 + 1.000 = 1.001; \quad c = b$$

Result: $C \times 2^c = 1.001 \times 2^{0011}$
Normalisation Algorithm:

- The mantissa has to be transformed such as it is either
  - Between 0.100000 and 0.111111 for positive numbers
  - or
  - Between 1.011111 and 1.000000 for negative numbers

- If the result $C*2^c$ is written as $C_n C_{n-1}... C_1 C_0 . C_{-1} C_{-2}...$, then $C$ is above range and mantissa is repeatedly right shifted one place at a time and the exponent is incremented each time by one until $C_0 \oplus C_{-1} = 1$

- If the result $C*2^c$ can be written as $C_0 . C_{-1} C_{-2}...$, and $C_0 \oplus C_{-1} = 0$, then $C$ is below the range and mantissa is repeatedly left shifted one place at a time and the exponent is decremented each time by one until $C_0 \oplus C_{-1} = 1$

Example 1:

$$1010.100*2^{0100}$$

$c=0101$, $C=1010.100$
C right shifted 3 places
$c$ incremented by 1 three times
Result: $C*2^c = 1.010100*2^{0111}$

Example 2:

$$0.00100*2^{0110}$$

$c=0110$, $C=0.00100$
C left shifted 2 places
$c$ decremented by 1 two times
Result: $C*2^c = 0.10000*2^{0100}$
3.5.3 Floating Point Multiplication

Multiplication Algorithm:

- Lets assume that $A \times 2^a$ and $B \times 2^b$ will be multiplied
- The multiplication result will have:
  - Mantissa resulted from the multiplication of the two mantissas ($C = A \times B$)
  - Exponent resulted from the addition of the two exponents ($c = a + b$)

Example:

$$0.100 \times 2^{0010} \quad * \quad 0.101 \times 2^{0011}$$

$a = 0010, \quad b = 0011$;
$A = 0.100, \quad B = 0.101$;

$C = A \times B = 0.010100$
$c = a + b = 0101$

Normalisation: left shift $C$ 1 place and decrement $c$ by 1

Result: $0.101 \times 2^{0100}$

Overflow:

- The result becomes so large that the exponent cannot be represented in the number of bits allocated
3.5.4 Floating Point Division

Division Algorithm:

- Lets assume that \( A \times 2^a \) will be divided by \( B \times 2^b \)
- The division result will have:
  - Mantissa resulted from the division of the two mantissas \( (C = A/B) \)
  - Exponent resulted from the subtraction of the two exponents \( (c = a-b) \)

Example:

\[
0.101 \times 2^{0100} \div 0.100 \times 2^{0010}
\]

\( a=0100, \quad b=0010; \)

\( A=0.101, \quad B=0.100; \)

\[
C = A / B = 01.010 \\
c = a - b = 0010
\]

Normalisation: right shift \( C \) 1 place and increment \( c \) by 1

Result: \( 0.101 \times 2^{0011} \)

Underflow:

- The result becomes so small that it cannot be represented accurately in the number of bits allocated