What is an Adder?

An adder is a digital logic circuit in electronics that is extensively used for the addition of numbers. In many computers and other types of processors, adders are even used to calculate addresses and related activities and calculate table indices in the ALU and even utilized in other parts of the processors. These can be built for many numerical representations like excess-3 or binary coded decimal. Adders are basically classified into two types: Half Adder and Full Adder.

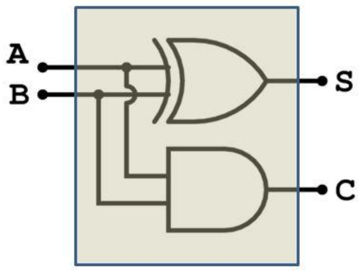
Half Adder

The half adder is a digital circuit that adds two binary digits called as AUGEND & ADDEND and produces two outputs as SUM & CARRY. XOR gate is applied to both inputs to produce SUM and AND gate is applied to both inputs to produce CARRY.

Here, it must be kept in mind that a Half Adder adds only two single digit binary numbers. Thus, we can get a total of four different addition operations which are:

Minuend		Addend		Carry	Sum
0	+	0	=	0	0
0	+	1	=	0	1
1	+	0	=	0	1
1	+	1	=	1	0

Circuit of a Half Adder:



Benefits:

1. **Simplicity:**
    - Half adders are relatively simple and straightforward in terms of design and implementation. They require a small number of logic gates, making them efficient for basic addition operations.
2. **Low Complexity:**
    - The circuit complexity of a half adder is lower compared to a full adder. This makes it suitable for simple addition tasks where only the current bit positions need to be considered.
3. **Fast Operation:**
    - Half adders operate quickly, providing fast results for basic addition operations. This is important in computer architecture where speed is often a critical factor.

Limitations:

1. **No Consideration of Previous Carry:**
    - One of the main limitations of a half adder is that it does not take into account any carry from the previous addition operation. This makes it unsuitable for chaining multiple addition operations in a sequence.
2. **Limited to Single-Bit Addition:**

- A half adder can only add two single-bit binary numbers. For multi-bit addition, you would need to use multiple half adders or upgrade to a full adder.

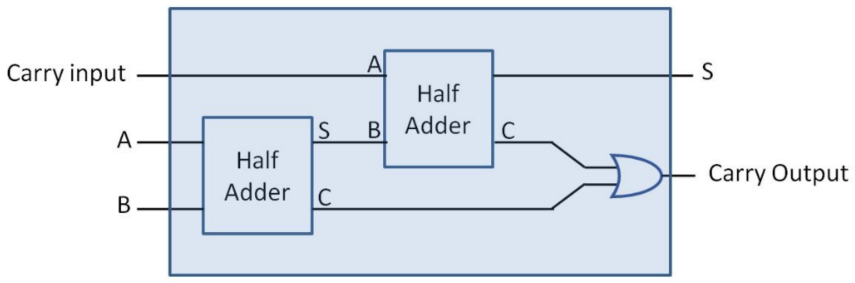
3. **Not Suitable for Arithmetic Overflow:**
    - Half adders do not detect or handle arithmetic overflow. In computer architecture, it's important to manage overflow conditions to prevent errors in calculations, and a half adder alone does not provide this functionality.
4. **Requires Additional Circuitry for Multi-Bit Addition:**
    - When performing multi-bit addition, multiple half adders need to be connected to handle each bit position. This requires additional circuitry and can lead to increased complexity.
5. **Limited Functionality:**
    - While suitable for basic addition, half adders do not support subtraction or other more advanced arithmetic operations. Additional circuitry and logic gates are needed to implement these functionalities.

Full Adder

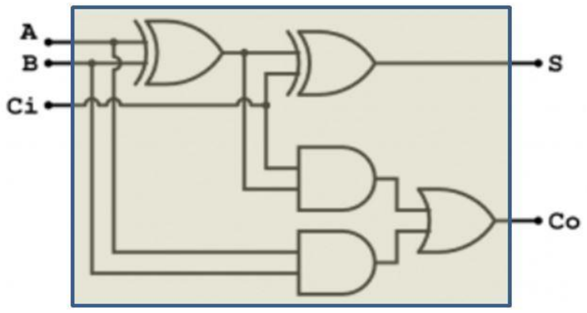
A Full Adder is a digital circuit that not only produces a CARRY; it can even utilize that. Thus a Full adder can be better defined as a digital circuit that inputs three binary bits and adds them to produce a SUM and a CARRY. Among of the three input bits; first is the ADDEND, second is the AUGEND and third is the INPUT CARRY.

- Full Adder is a combinational logic circuit.
- It is used for the purpose of adding two single bit numbers with a carry.
- Thus, full adder has the ability to perform the addition of three bits.
- Full adder contains 3 inputs and 2 outputs (sum and carry) as shown-

A Full Adder can be constructed by connecting two Half Adders serially. Consider the following Block Diagram:



Circuit of a Full Adder:



Truth Table of a Full Adder:

A	B	C_i	C_o	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Benefits:

1. **Binary Addition:**
   - The primary function of a full adder is to add binary numbers. It can add three input bits (A, B, and the carry from the previous stage) and produce a sum output and a carry output.
2. **Arithmetic Operations:**
   - Full adders are crucial components for building arithmetic circuits in computers. They are used in the design of adders, subtractors, and other arithmetic units.
3. **Modularity:**
   - Full adders are modular components that can be easily combined to build larger and more complex circuits. This modularity is essential in designing scalable and efficient systems.
4. **Flexibility:**
   - Full adders can be cascaded to create adders capable of handling larger binary numbers. This flexibility allows for the design of circuits that can perform arithmetic operations on multi-bit binary numbers.
5. **Complementing Operations:**
   - With modifications, full adders can be used to perform other operations such as subtraction and logical operations like XOR and AND.

Limitations:

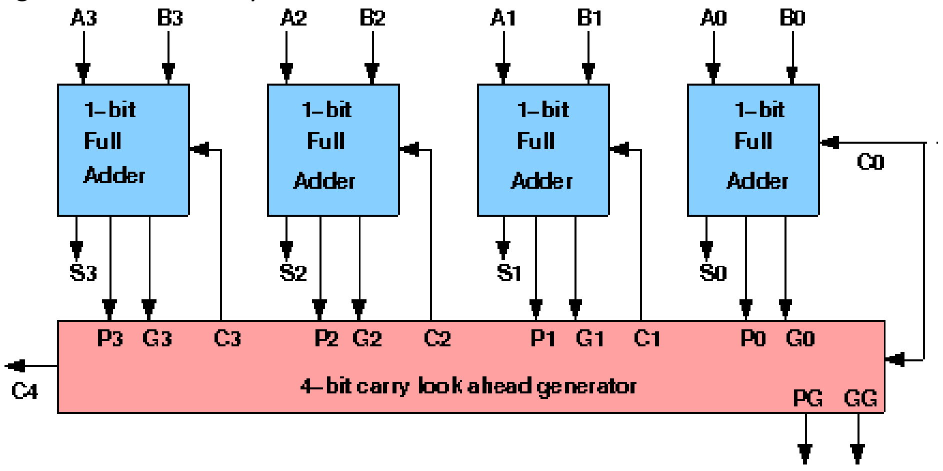
1. **Propagation Delay:**
   - Full adders contribute to the overall propagation delay in a digital circuit. As the number of stages increases, the cumulative delay also increases, affecting the overall performance of the system.
2. **Area and Power Consumption:**
   - Full adders require physical space on a chip, and when numerous adders are needed for complex operations, it can contribute to increased chip area and power consumption.
3. **Complexity in Design:**
   - While a single full adder is relatively simple, designing more complex circuits requires careful consideration of signal paths, timing, and interconnections. This complexity can make the design process challenging.
4. **Limited to Binary Operations:**
   - Full adders are specifically designed for binary addition. While they can be adapted for other operations, they may not be as efficient or optimized for those tasks.

5. **Not Ideal for Floating Point Operations:**

   - Full adders are not designed for floating-point arithmetic, which is commonly used in scientific and engineering applications. Specialized circuits or algorithms are required for floating-point operations.

Carry Lookahead Adders

To reduce the computation time, there are faster ways to add two binary numbers by using carry lookahead adders. They work by creating two signals P and G known to be Carry Propagator and Carry Generator. The carry propagator is propagated to the next level whereas the carry generator is used to generate the output carry, regardless of input carry. The block diagram of a 4-bit Carry Lookahead Adder is shown here below -



The number of gate levels for the carry propagation can be found from the circuit of full adder. The signal from input carry Cin to output carry Cout requires an AND gate and an OR gate, which constitutes two gate levels. So, if there are four full adders in the parallel adder, the output carry C5 would have 2 X 4 = 8 gate levels from C1 to C5. For an n-bit parallel adder, there are 2n gate levels to propagate through.

Design Issues:

The corresponding Boolean expressions are given here to construct a carry lookahead adder. In the carry-lookahead circuit we ned to generate the two signals carry propagator(P) and carry generator(G),

Pi = Ai ⊕ Bi
Gi = Ai · Bi

The output sum and carry can be expressed as

Sumi = Pi ⊕ Ci
Ci+1 = Gi + (Pi · Ci)

Having these we could design the circuit. We can now write the Boolean function for the carry output of each stage and substitute for each Ci its value from the previous equations:

C1 = G0 + P0 · C0
C2 = G1 + P1 · C1 = G1 + P1 · G0 + P1 · P0 · C0
C3 = G2 + P2 · C2 = G2 P2 · G1 + P2 · P1 · G0 + P2 · P1 · P0 · C0
C4 = G3 + P3 · C3 = G3 P3 · G2 P3 · P2 · G1 + P3 · P2 · P1 · G0 + P3 · P2 · P1 · P0 · C0

Carry Lookahead Adder Advantages and Disadvantages

The advantages of CLA are:

- Carry lookahead adder is considered as the fastest adder when compared with other adder systems.
- Here, the propagation delay is minimum because the output carry bit is only based on the first carry bit which is applied at the input stage.

- Using the equations of carry propagate, carry generate and carry bits, CLA devices generate carry-in for every adder in a simultaneous manner.

The disadvantages of CLA are:

- When the number of variables gets increased, the design of carry-lookahead adder becomes more complex.
- So, when the variables get increased and when CLA is integrated with IC, the area is required to increase.
- As the hardware is more, the circuitry cost becomes expensive when compared with the ripple carry adder.

The carry lookahead adder applications are:

- Carry lookahead adders operating with high speed are employed as integrated circuits so that it is simple to integrate adder in many circuits. Also, the increase in the count of gates is even moderate when implemented for higher bits.
- When CLA's are used for high-bit calculations, the device offers more speed whereas the circuit complexity also increases. Usually, these are used for 4-bit modules so that they are integrated together for high-bit computations.
- On a regular basis, carry-lookahead adders are used in Boolean computations.

Binary Multiplication?

Binary multiplication is one of the four basic operations performed on binary numbers that is addition, subtraction, multiplication, and division. This multiplication is similar to decimal multiplication but uses only two digits that are 0 and 1, unlike 0 to 9 in decimal numbers.

Binary Multiplication Table

Since binary numbers make use of only two digits that is 0 and 1, we get to multiply only these binary numbers while performing multiplication. The multiplication table for binary numbers is as follows:

Binary Numbers	**Multiplication Value**
0×0	0
1×0	0
0×1	0
1×1	1

Binary Multiplication Rules

In binary multiplication, we have a multiplier and a multiplicand. The basic rules for the multiplication of binary numbers are:

Multiplicand	**Multiplier**	**Product**
0	0	0×00×0=0
0	1	0×10×1=0
1	0	1×01×0=0
1	1	1×11×1=1

Multiplication of Binary Numbers

As binary numbers comprise of only two values i.e. 0 and 1, the process of multiplication of these numbers becomes easier as compared to decimal numbers. The steps involved in multiplying binary numbers are given below:

Example: Multiply 11101 by 1001.

Step 1: Write the multiplicand 11101 and the multiplier 1001 one below the other in proper columns.

Step 2: Start the multiplication process from the extreme right digit of the multiplier which is 1 in this case, with all the digits of the multiplicand.

Step 3: Add the placeholder 'X' before starting the multiplication with the next digit of the multiplier in the next row.

Step 4: Repeat the same procedure till the leftmost digit in the multiplier is multiplied by all the digits in the multiplicand.

Step 5: The product obtained in each row is called the partial product. Finally, all the partial products are added using the rules for binary addition.

(Rules for binary addition are: 0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1, 1 + 1 = 0, 1 carry).

Let us look at the actual multiplication:

				1	1	1	0	1
				x	**1**	**0**	**0**	**1**
				1	**1**	**1**	**0**	**1**
			0	**0**	**0**	**0**	**0**	**x**
		0	0	0	0	0	x	x
+	1	1	1	0	1	x	x	x
1	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**

Therefore, we can say that the product of 11101 and 1001 is 100000101. We can also check our result by changing binary to decimal numbers. The decimal equivalent of 11101 is 29 and that of 1001 is 9. And the product of 29 and 9 is 261 which is written as 100000101 in binary notation.

Multiplication of Binary Numbers with Decimal Points

Multiplying binary numbers with decimal points is an easy procedure. It is similar to multiplying two binary numbers without decimals. The only difference is, after performing the entire multiplication we need to place the decimal point by counting the decimal places in the multiplier and the multiplicand.

Let us understand this with an example:

Example: Multiply: 1011.01 and 110.1

Solution: We will perform simple binary multiplication and insert a decimal point in the final answer:

				1	0	1	1	0	1
					x	1	1	0	1
				1	0	1	1	0	1
			0	0	0	0	0	0	x
		1	0	1	1	0	1	x	x

+	1	0	1	1	0	1	x	x	x
1	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**1**

The answer obtained by multiplying 101101 and 1101 is 1001001001. Now as we have to multiply 1011.01 and 110.1, the final answer is 1001001.001.

Signed Binary Multiplication

Signed Binary Multiplication is also known as 2's complement multiplication. We can perform this multiplication by simply multiplying the magnitudes of the two numbers and then extending it to the original sign bit of the number.

It is to be noted that unlike addition when we multiply an n-bit number with an m-bit number, it results in an n+m-bit number.

Let us understand this signed multiplication using an example:

Example: Multiply -5 and 7 in signed binary multiplication.

Solution: We know that in binary numbers -5 is written as 1011 and 7 is written as 0111.

In order to perform signed multiplication, we simply need to perform binary multiplication using simple rules, that is 1×0 = 01×0 = 0, 0×0 = 00×0 = 0, and 1×1 = 11×1 = 1.

After final multiplication, we have to extend each row to the number of sign bits, in this case, 8-bit. Once all the rows are extended, we can add the rows together using rules of addition, and give the result in an 8-bit representation.

						1	0	1	1
					x	0	1	1	1
		1	1	1	1	1	0	1	1
		1	1	1	1	0	1	1	x
		1	1	1	0	1	1	x	x
	+	0	0	0	0	0	x	x	x
1	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**0**	**1**

The final result is represented in 8-bit ignoring the extra two digits in the front.

So, the signed multiplication result for 1011 and 0111 is 11011101.

Unsigned Binary Multiplication

Solving unsigned binary multiplication is an easy process. This multiplication can be solved like any other decimal multiplication.

Let us check a solved example for better understanding:

Example: Multiply 13 and 9 in binary digits.

Solution: 13 in binary can be written as 1101 and 9 in binary is denoted as 1001.

Performing unsigned multiplication:

			1	1	0	1
		x	1	0	0	1
			1	1	0	1

		0	0	0	0	x
	0	0	0	0	x	x
1	1	0	1	x	x	x
1	**1**	**1**	**0**	**1**	**0**	**1**

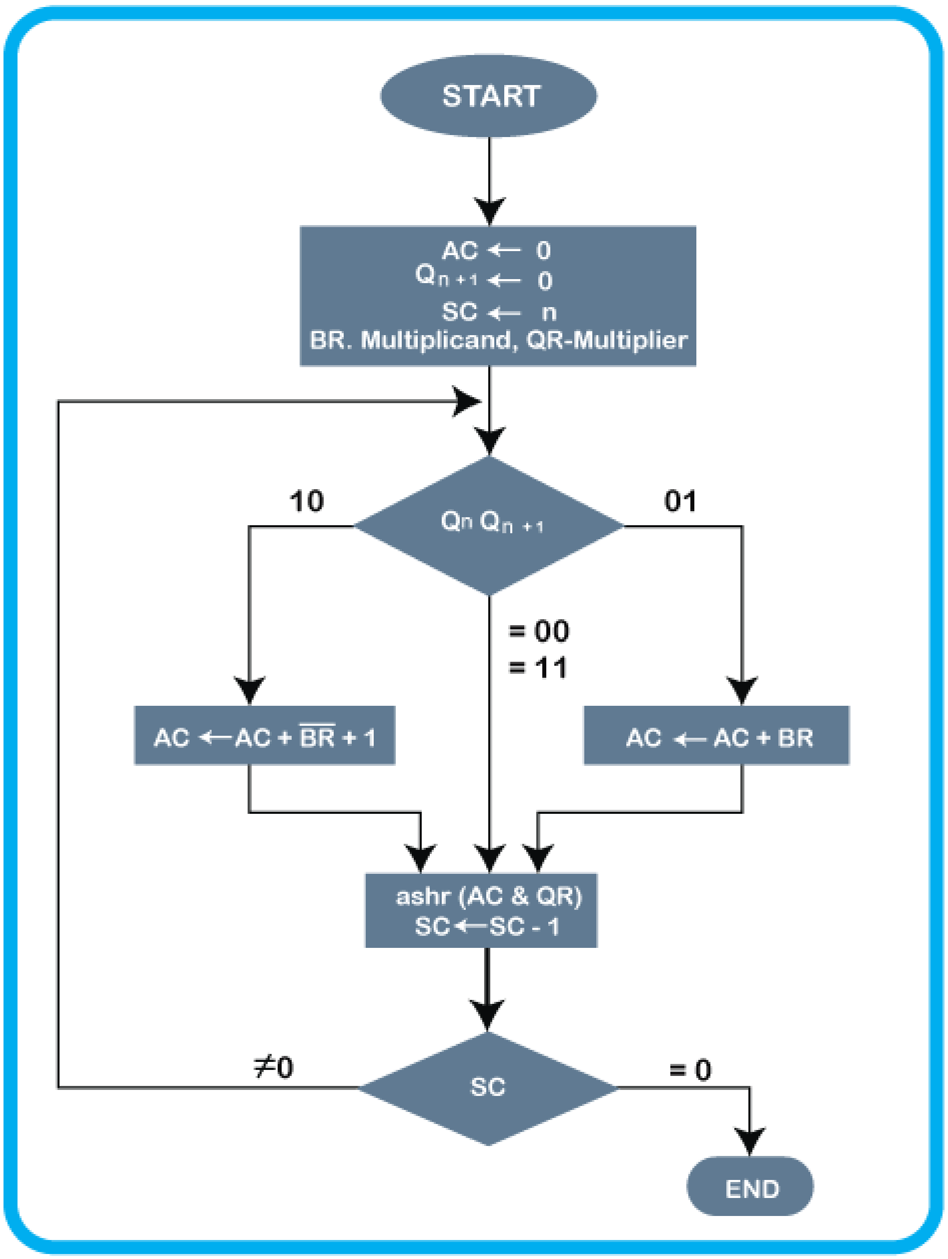
So, unsigned multiplication for 1101 and 1001 is 1110101. That is 13 multiplied by 9 gives 117.

Booth's Algorithm

Andrew Donald Booth's Algorithm, introduced in 1951, revolutionized binary multiplication by reducing the number of additions and shifts required. This algorithm capitalizes on the concept of signed-digit representation, where digits are encoded as either -1, 0, or 1. Booth's Algorithm is particularly advantageous when multiplying numbers with repeated patterns of 1s or 0s.

How Booth's Algorithm Works?

The algorithm operates by identifying sequences of consecutive 1s and 0s in the multiplier, efficiently adjusting the partial products based on these patterns. It involves three basic steps:



Booth's Algorithm is a clever technique used to perform binary multiplication more efficiently, particularly when dealing with numbers that have repeated patterns of 1s or 0s in their binary representation. This algorithm reduces the number of additions and shifts required compared to traditional multiplication methods, making it a valuable tool in computer organization and hardware implementations. Let's dive into the detailed workings of Booth's Algorithm step by step:

Step 1: Initialization

1. Input: Booth's Algorithm takes two binary numbers as input: the multiplicand (M) and the multiplier (Q). The length of the multiplier determines the number of iterations in the algorithm.

2. Initialization: Create three registers:
A (Accumulator): Initialize to zero, representing the result of the multiplication.
Q (Multiplier): Load the binary multiplier into this register.
Q_-1 (Previous Qubit): Initialize to 0, representing the least significant bit (LSB) of the multiplier before the start of the algorithm.

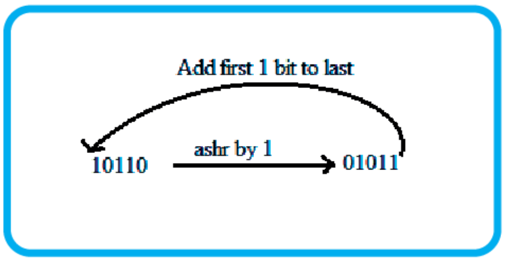
Step 2: Iteration

The algorithm iterates through the bits of the multiplier. For each bit of the multiplier, it performs the following actions:

1. Identify Patterns: Examine two consecutive bits of the multiplier along with the previous bit (Q*i, Q*-1, Q_i+1). These bits form a three-bit pattern. There are four possible patterns: 00, 01, 10, and 11.

2. Pattern-based Addition or Subtraction:
If the pattern is 00 or 11, no action is taken, and the algorithm proceeds to the next iteration.
If the pattern is 01, subtract the multiplicand (M) from the accumulator (A) and store the result in the accumulator.
If the pattern is 10, add the multiplicand (M) to the accumulator (A) and store the result in the accumulator.

3. Shift Right:



Perform an arithmetic shift right on the accumulator and the multiplier. This is equivalent to shifting the bits one position to the right, with the most significant bit (MSB) of Q becoming the new Q_-1, and the MSB of A becoming the new MSB of Q.

Step 3: Normalization

After completing the iteration, perform a final normalization step:

1. Shift A/Q: Perform one last arithmetic shift right on the accumulator and the multiplier to align their positions.

2. Result Extraction: The final result of the multiplication is stored in the accumulator A.

Example of Booth's Algorithm

Let's illustrate Booth's Algorithm with an example:

M (Multiplicand) = 1010
Q (Multiplier) = 1101

1. Initialization:

- A (Accumulator) = 0000
- Q (Multiplier) = 1101
- Q_-1 (Previous Qubit) = 0

2. Iteration:

- Iteration 1: Pattern = 001 (Subtract M from A)
- Iteration 2: Pattern = 110 (Add M to A)

3. Normalization:

- Shift A/Q Right: A = 1001, Q = 1110

Final Result:
A (Result) = 1001

Advantages and Applications of Booth's Algorithm

Booth's Algorithm offers several advantages:

1. Reduced Operations: Booth's Algorithm significantly reduces the number of additions and shifts compared to traditional multiplication methods. This leads to faster multiplication operations and conserves computing resources.

2. Optimized for Hardware: Booth's Algorithm is well-suited for hardware implementations, making it ideal for embedded systems, digital signal processors, and specialized hardware accelerators.

3. Efficient for Large Numbers: The algorithm's efficiency becomes more pronounced as the size of the numbers being multiplied increases, making it particularly valuable in applications requiring high-precision arithmetic.

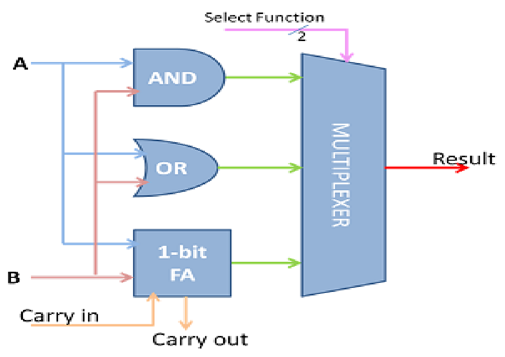
Arithmetic and Logic Unit (ALU)

The Arithmetic and Logic Unit (ALU) is a crucial component in the central processing unit (CPU) of a computer. The **Arithmetic Logic Unit (ALU)** is the heart of any CPU. An ALU performs three kinds of operations, i.e.

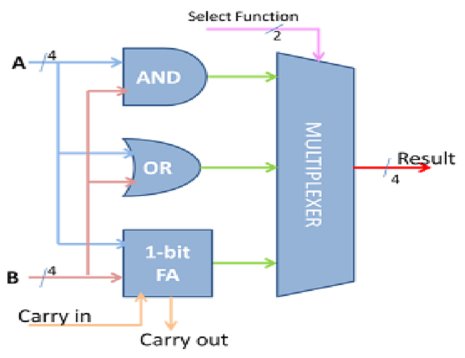
- Arithmetic operations such as Addition/Subtraction,
- Logical operations such as AND, OR, etc. and
- Data movement operations such as Load and Store

ALU derives its name because it performs arithmetic and logical operations. A simple ALU design is constructed with Combinational circuits. ALUs that perform multiplication and division are designed around the circuits developed for these operations while implementing the desired algorithm. More complex ALUs are designed for executing Floating point, Decimal operations and other complex numerical operations. These are called Coprocessors and work in tandem with the main processor.

The design specifications of ALU are derived from the Instruction Set Architecture. The ALU must have the capability to execute the instructions of ISA. An instruction execution in a CPU is achieved by the movement of data/datum associated with the instruction. This movement of data is facilitated by the Datapath. For example, a LOAD instruction brings data from memory location and writes onto a GPR. The navigation of data over Datapath enables the execution of LOAD instruction. We discuss Datapath more in details in the next chapter on Control Unit Design. The trade-off in ALU design is necessitated by the factors like Speed of execution, hardware cost, the width of the ALU.



1-bit 3-function Combinational ALU



4-bit 3-function Combinational ALU

Let's break down the design of an ALU into its key components and functions:

1. Inputs and Outputs:

- **Inputs (Operands):** The ALU typically takes two binary inputs, which are the operands for the operation it will perform. These operands could be data from registers, memory, or immediate values.
- **Outputs (Result):** The ALU produces a binary output, which is the result of the arithmetic or logic operation.

2. Control Lines:

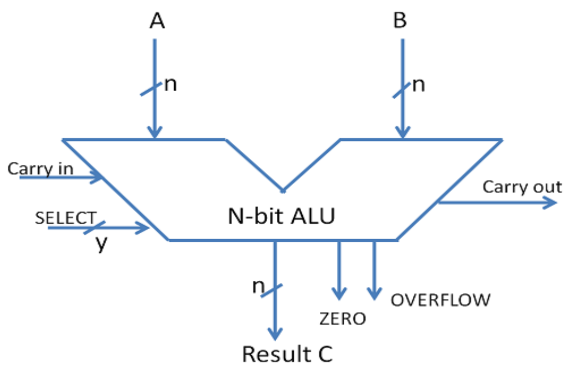
- The ALU has control lines that specify the operation to be performed. These control lines determine whether the ALU will perform addition, subtraction, AND, OR, XOR, or other operations.

3. Arithmetic Operations:

- **Addition and Subtraction:**
    - The ALU needs the ability to perform addition and subtraction. For subtraction, a 2's complement operation is often used.
- **Multiplication and Division (optional):**
    - Some ALUs may have dedicated circuits or additional components to perform multiplication and division, though these operations can also be implemented through a series of additions and subtractions.

4. Logic Operations:

- The ALU includes logic gates for operations such as AND, OR, XOR, and NOT. These operations are essential for manipulating and comparing binary data.



5. Flags:

- **Zero Flag:** Indicates whether the result of an operation is zero.
- **Negative Flag:** Indicates whether the result is negative.
- **Overflow Flag:** Indicates if an arithmetic operation produces a result too large for the designated number of bits.
- **Carry Flag:** Used in multi-precision arithmetic and addition to indicate if there is a carry-out.

6. Multiplexers and Demultiplexers:

- Multiplexers are used to select the input operands or constant values based on the control lines.
- Demultiplexers are used to route the output to the appropriate destination, such as a register or the next stage in the pipeline.

7. Registers:

- The ALU interacts with registers to store intermediate results and input operands. Registers are fast, temporary storage locations within the CPU.

8. Bit-wise Operations:

- ALUs often include bit-wise operations like shifting (left shift, right shift) and rotation. These operations are useful in various scenarios, such as bit manipulation and addressing.

9. Timing and Pipelining:

- The design must consider timing requirements to ensure that the outputs are stable and valid at the right time.
- Pipelining may be employed to increase throughput by allowing the ALU to begin processing the next instruction before completing the current one.

10. ALU Control Unit:

- The ALU control unit interprets the control lines and generates the necessary signals to perform the specified operation. It manages the operation of the ALU based on the opcode of the instruction.

11. Bus Interface:

- The ALU interfaces with the system bus to send and receive data to and from memory, registers, and other components.

12. Power Optimization:

- Design considerations may include power optimization techniques to minimize energy consumption, especially in mobile devices or battery-powered systems.

13. Error Handling:

- Some ALUs include error detection and correction mechanisms to ensure the integrity of data during processing.

14. Parallelism and Vector Operations (Optional):

- In some advanced processors, ALUs may be designed to handle parallel or vector operations for improved performance in certain types of computations.

The design of an ALU is a critical aspect of CPU architecture, and variations exist depending on the specific requirements and goals of the processor design. ALU design involves a balance between speed, complexity, and power consumption to meet the overall performance goals of the computer system.

IEEE 754 standard for floating-point arithmetic

IEEE 754 standard for floating-point arithmetic is one of the most widely adopted standards for representing and performing operations on floating-point numbers in computing. Keep in mind that standards can be updated, and it's a good idea to check the latest documentation for any revisions or additions.

The IEEE 754 standard defines formats and rules for floating-point arithmetic in binary and decimal formats. Here are some key aspects of the IEEE 754 standard:

Formats:

1. **Single Precision (32 bits):**
   - Sign bit: 1 bit
   - Exponent: 8 bits
   - Significand (Fraction): 23 bits
2. **Double Precision (64 bits):**
   - Sign bit: 1 bit
   - Exponent: 11 bits

    - Significand (Fraction): 52 bits

3. **Extended Precision (80 bits):**
    - Sign bit: 1 bit
    - Exponent: 15 bits
    - Significand (Fraction): 64 bits

Components:

Sign Bit:

- 0 for positive numbers, 1 for negative numbers.

Exponent:

- Represents the exponent value in a biased format. A bias is added to allow both positive and negative exponents. The bias is $2^{(k-1)} - 1$, where k is the number of bits in the exponent field.

Significand (Fraction):

- Also known as the mantissa or significand, it represents the fractional part of the floating-point number.

Special Values:

1. **Zero and Subnormal Numbers:**
    - Special representations for zero and very small numbers.
2. **Infinity and NaN (Not a Number):**
    - Representations for positive and negative infinity, as well as a NaN value to signify undefined or indeterminate results.

Rounding Modes:

- IEEE 754 defines several rounding modes for operations involving floating-point numbers. Common modes include rounding to the nearest representable value, rounding toward positive infinity, rounding toward negative infinity, and rounding toward zero.

Precision and Accuracy:

- The standard ensures that basic arithmetic operations (addition, subtraction, multiplication, and division) are performed with a specified level of precision, and the results are correctly rounded.

Interchange Format:

- The standard specifies a binary interchange format, allowing consistent representation of floating-point numbers across different computer systems.

Compliance:

- IEEE 754 compliance is essential for ensuring that floating-point operations yield consistent results across different hardware architectures.

https://docs.oracle.com/cd/E19957-01/806-3568/ncg_math.html **Follow this link for more detail**

BY: Kapil Sharma