ARITHMETIC LOGIC UNIT USING VERILOG

Name	*Number*
Mohamed Ibrahim Elsayed Shosha	62
Fares Mostafa Ibrahim	59
Abdullah Abduljalil Zaky	52
Mohamed Ali Mohamed Elkhateb	80
Mostafa Abdulrahim Mostafa	85

1. Introduction

- The **Arithmetic Logic Unit (ALU)** is a critical component of the Central Processing Unit (CPU) that performs arithmetic and logical operations on binary data. These operations include basic arithmetic (addition, subtraction) and logical operations (AND, OR, XOR, NOT). In this project, an ALU was designed using **Verilog** HDL (Hardware Description Language) and tested through **simulation** to verify its functionality.
- The main goal of this project was to design a basic 8-bit ALU capable of performing various arithmetic, logical, and shift operations. This design was simulated using a Verilog code and tested using simulation tools like **ModelSim** or **EDA playground Simulator**.

2. Objective

- The primary objective of this project is to design and implement a simple ALU that can perform multiple arithmetic and logical operations, and then simulate and test it on an FPGA platform.
    - To design an ALU capable of performing arithmetic (addition, subtraction), logical (AND, OR, XOR), and shift (logical shift left/right) operations.
    - To implement the ALU design using **Verilog** HDL.
    - To test and simulate the ALU functionality using **simulation tools** like **ModelSim** or **EDA playground Simulator**.

3.ALU Design Overview

- The ALU design includes several operations, each controlled by a 4-bit **control signal**. The ALU operates on two 8-bit inputs, performing operations such as:
    - Arithmetic Operations: Addition, Subtraction
    - Logical Operations: AND, OR, XOR, NOT, NOR, XNOR
    - Shift Operations: Logical Shift Left (LSL), Logical Shift Right (LSR)

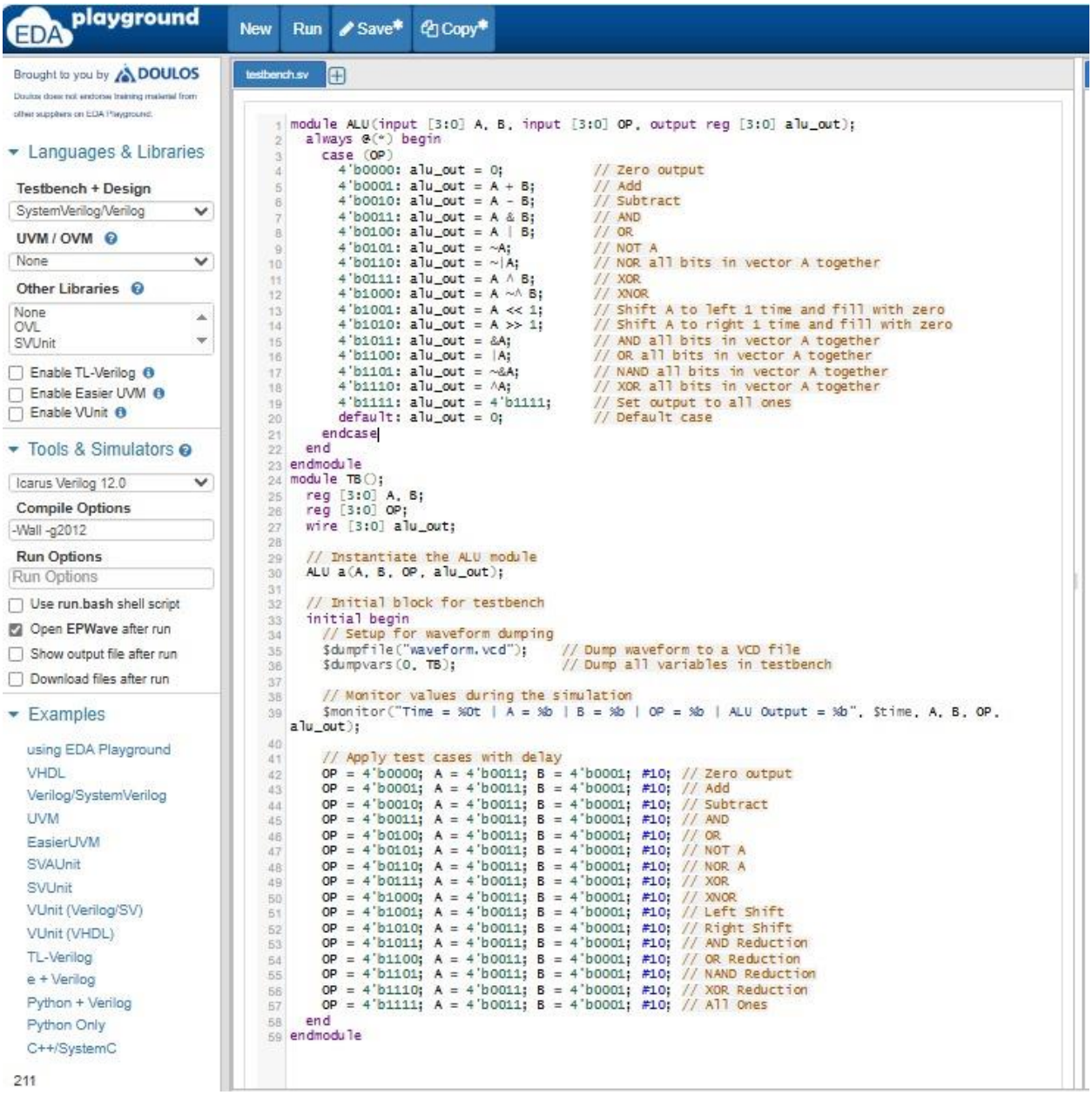
#	Operation	Out
0	0000	0000
1	0001	A+B
2	0010	A-B
3	0011	A & B
4	0100	A | B
5	0101	! A
6	0110	NORing A bits
7	0111	A XOR B
8	1000	A XNOR B
9	1001	Shift left(A)
A	1010	Shift Right (A)
B	1011	ANDing A bits
C	1100	ORing A bits
D	1101	NANDing A bits
E	1110	XORing A bits
F	1111	1111

Fig. 1 (The Operations Table)

- Reduction Operations: AND Reduction, OR Reduction, NAND Reduction, XOR Reduction
- Zero output: Set all bits to 0
- All Ones: Set all bits to 1

4. Verilog Code Explanation

Below is the Verilog code for the ALU design. This code implements all the operations described above.

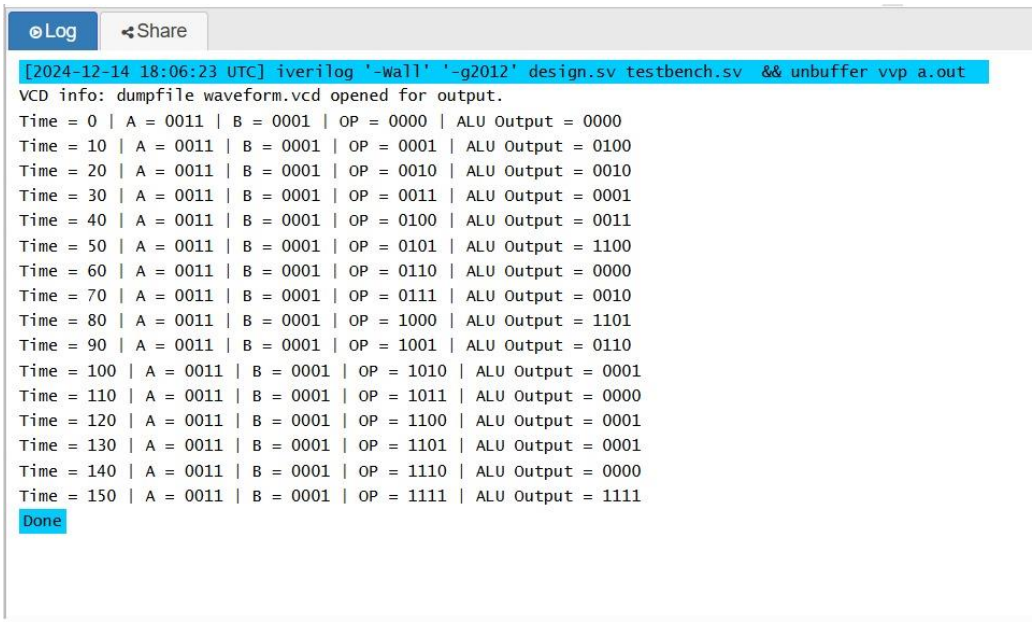



```
module ALU(input [3:0] A, B, input [3:0] OP, output reg [3:0] alu_out);
  always @(*) begin
    case (OP)
      4'b0000: alu_out = 0;           // Zero output
      4'b0001: alu_out = A + B;       // Add
      4'b0010: alu_out = A - B;       // Subtract
      4'b0011: alu_out = A & B;       // AND
      4'b0100: alu_out = A | B;       // OR
      4'b0101: alu_out = ~A;          // NOT A
      4'b0110: alu_out = ~|A;         // NOR all bits in vector A together
      4'b0111: alu_out = A ^ B;       // XOR
      4'b1000: alu_out = A ~^ B;      // XNOR
      4'b1001: alu_out = A << 1;      // Shift A to left 1 time and fill with zero
      4'b1010: alu_out = A >> 1;      // Shift A to right 1 time and fill with zero
      4'b1011: alu_out = &A;          // AND all bits in vector A together
      4'b1100: alu_out = |A;          // OR all bits in vector A together
      4'b1101: alu_out = ~&A;         // NAND all bits in vector A together
      4'b1110: alu_out = ^A;          // XOR all bits in vector A together
      4'b1111: alu_out = 4'b1111;     // Set output to all ones
      default: alu_out = 0;           // Default case
    endcase
  end
endmodule
module TB();
  reg [3:0] A, B;
  reg [3:0] OP;
  wire [3:0] alu_out;

  // Instantiate the ALU module
  ALU a(A, B, OP, alu_out);

  // Initial block for testbench
  initial begin
    // Setup for waveform dumping
    $dumpfile("waveform.vcd");     // Dump waveform to a VCD file
    $dumpvars(0, TB);              // Dump all variables in testbench

    // Monitor values during the simulation
    $monitor("Time = %0t | A = %b | B = %b | OP = %b | ALU Output = %b", $time, A, B, OP,
alu_out);

    // Apply test cases with delay
    OP = 4'b0000; A = 4'b0011; B = 4'b0001; #10; // Zero output
    OP = 4'b0001; A = 4'b0011; B = 4'b0001; #10; // Add
    OP = 4'b0010; A = 4'b0011; B = 4'b0001; #10; // Subtract
    OP = 4'b0011; A = 4'b0011; B = 4'b0001; #10; // AND
    OP = 4'b0100; A = 4'b0011; B = 4'b0001; #10; // OR
    OP = 4'b0101; A = 4'b0011; B = 4'b0001; #10; // NOT A
    OP = 4'b0110; A = 4'b0011; B = 4'b0001; #10; // NOR A
    OP = 4'b0111; A = 4'b0011; B = 4'b0001; #10; // XOR
    OP = 4'b1000; A = 4'b0011; B = 4'b0001; #10; // XNOR
    OP = 4'b1001; A = 4'b0011; B = 4'b0001; #10; // Left Shift
    OP = 4'b1010; A = 4'b0011; B = 4'b0001; #10; // Right Shift
    OP = 4'b1011; A = 4'b0011; B = 4'b0001; #10; // AND Reduction
    OP = 4'b1100; A = 4'b0011; B = 4'b0001; #10; // OR Reduction
    OP = 4'b1101; A = 4'b0011; B = 4'b0001; #10; // NAND Reduction
    OP = 4'b1110; A = 4'b0011; B = 4'b0001; #10; // XOR Reduction
    OP = 4'b1111; A = 4'b0011; B = 4'b0001; #10; // All Ones
  end
endmodule
```

5. Simulation and Testing

The ALU design was tested using **ModelSim or EDA playground Simulator** Simulator by creating different testbenches. The simulation verified that the ALU performs correctly for all operations.

- **Output :**



- **Simulation :**

Design using Proteus

1. Overview

This section describes the implementation of basic digital logic operations, including AND, XOR, Addition, and Multiplication using digital components within the Proteus design software. These components demonstrate fundamental operations crucial in arithmetic and logic circuits.

2. Implemented Modules

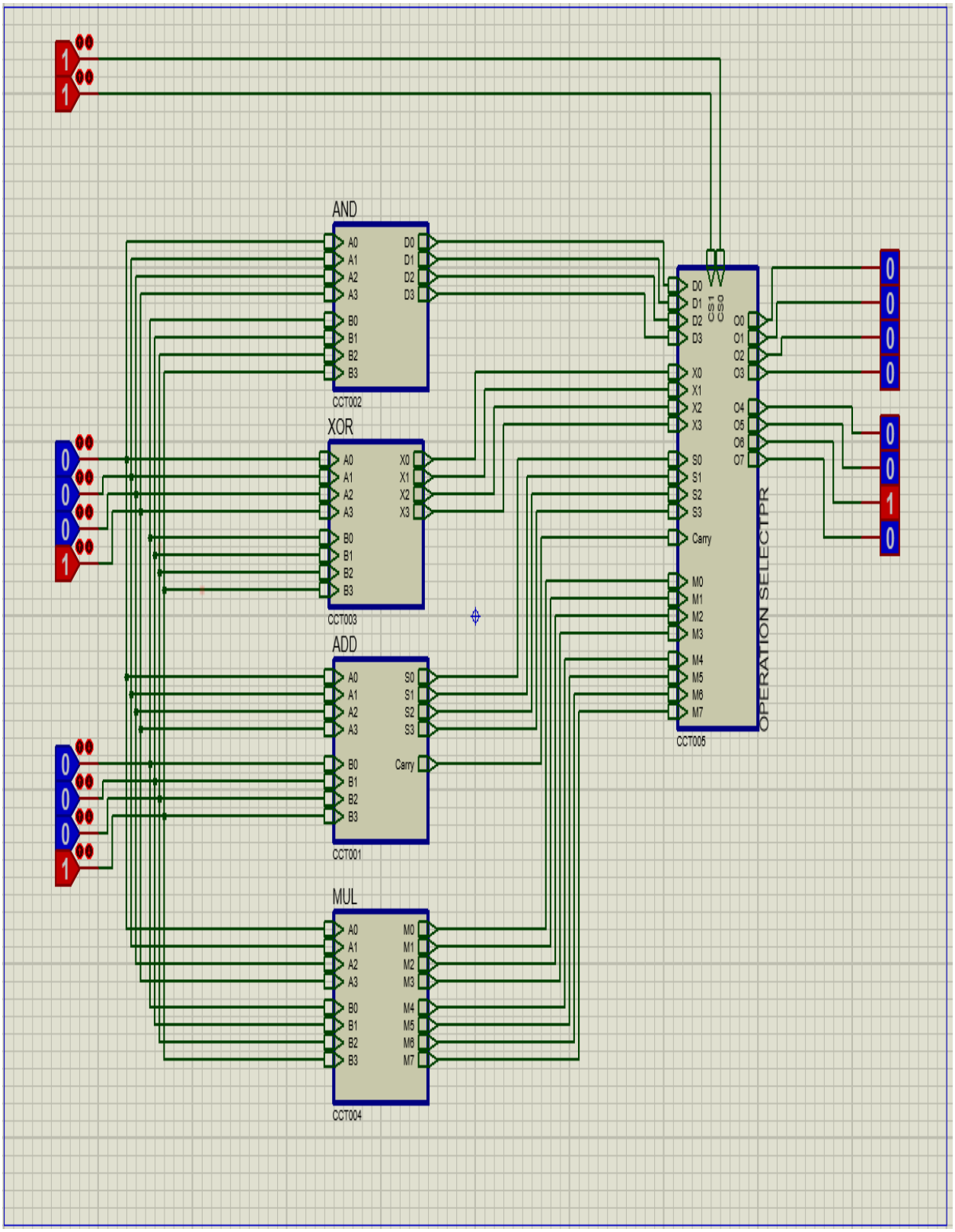
The following modules and gates were utilized to achieve the required operations:

- AND Gate: Performs the logical AND operation between two input signals.
- XOR Gate: Executes the exclusive-OR (XOR) operation, producing a HIGH output when the inputs differ.
- Adder (Full Adder): Realizes binary addition with carry output, achieved using the 74LS283 IC.
- Multiplier: Implements binary multiplication, often using combinational logic (AND gates and adders).

3.Design Description

1. **AND Gate**:
    - The AND gate outputs `1` (HIGH) only when both inputs are `1`.
    - This operation was implemented using **AND components** from the Proteus library.
2. **XOR Gate**:
    - The XOR gate outputs `1` when inputs are different (`A` $\oplus$ `B`).
    - XOR gates were selected and connected using the Proteus design environment.
3. **Adder (74LS283)**:
    - A 4-bit binary full adder IC **74LS283** was utilized to perform addition.
    - This IC adds two 4-bit binary numbers along with a carry-in and produces a 4-bit sum and carry-out.
4. **Multiplier**:
    - Multiplication was designed using a combination of **AND gates** (to generate partial products) and **Adders** (to sum up the partial products).
    - The process mimics binary multiplication logic.

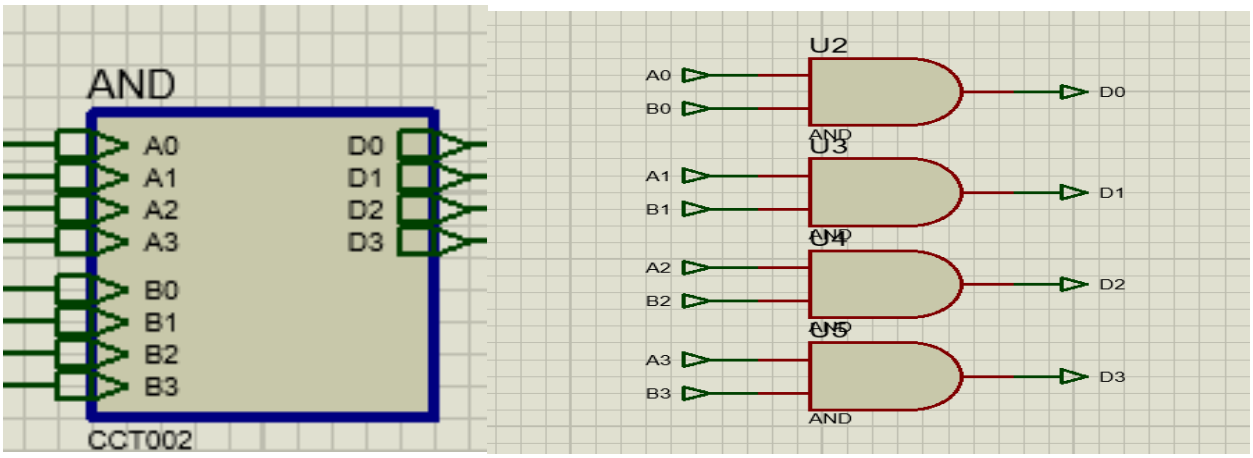
Design Pictures :



Parent and it's Child :

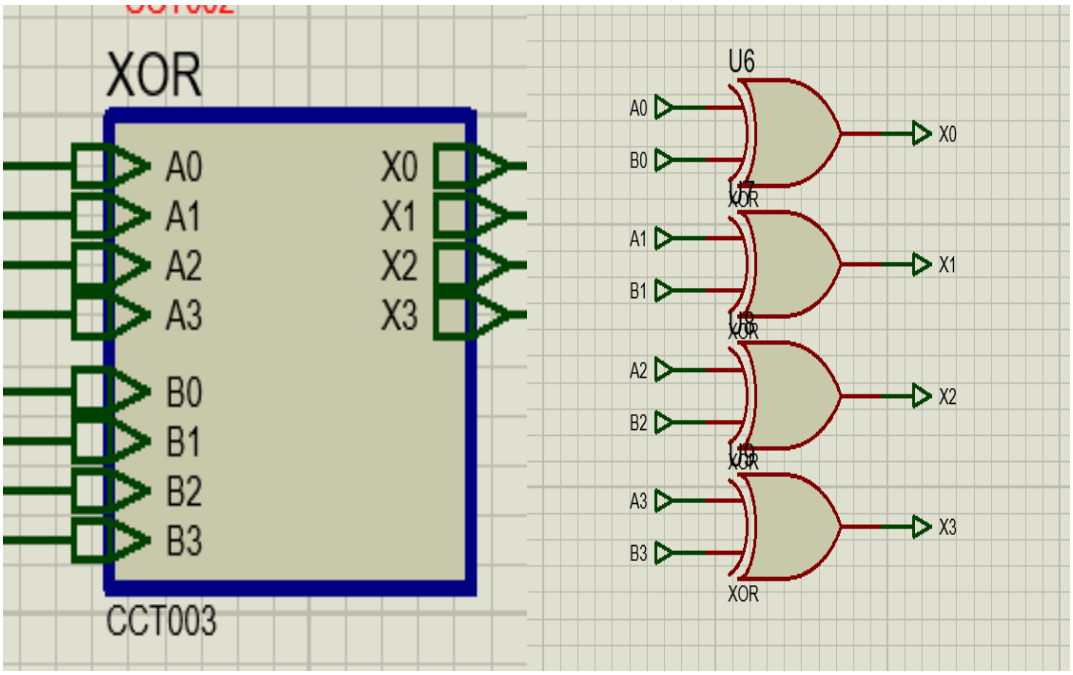
1.AND Operation

A logic gate that outputs 1 only when both inputs are 1, otherwise outputs 0



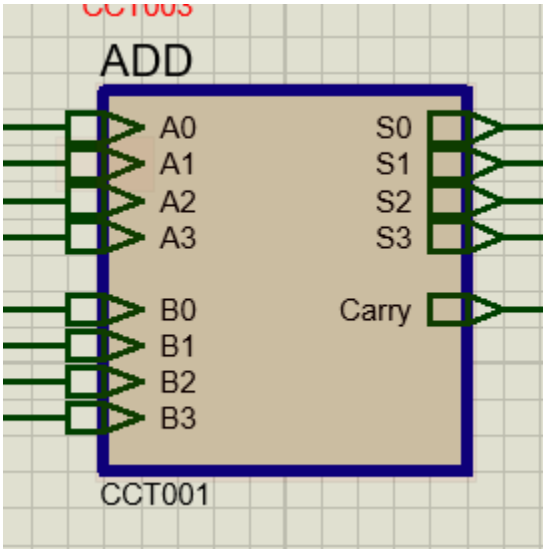
2.XOR Operation

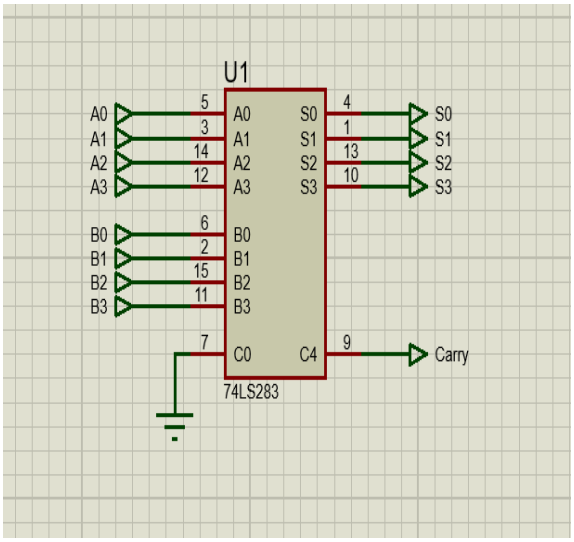
A logic gate that outputs `1` when the inputs are different (`0,1` or `1,0`), and `0` when they are the same



3.Addition Operation

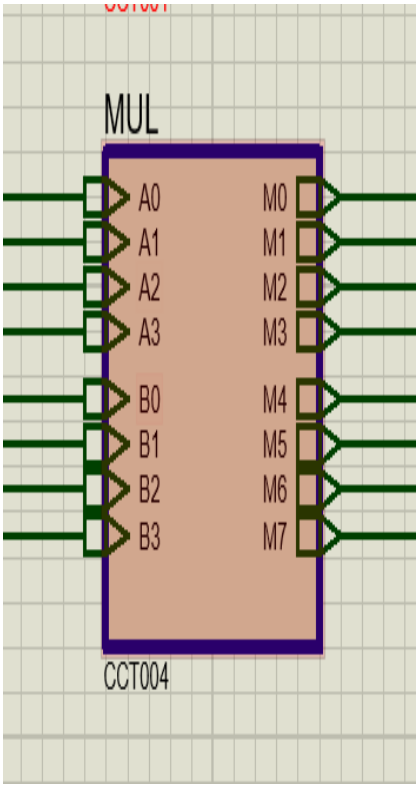
A binary addition operation, implemented using a full adder to sum two binary numbers and handle carry propagation.

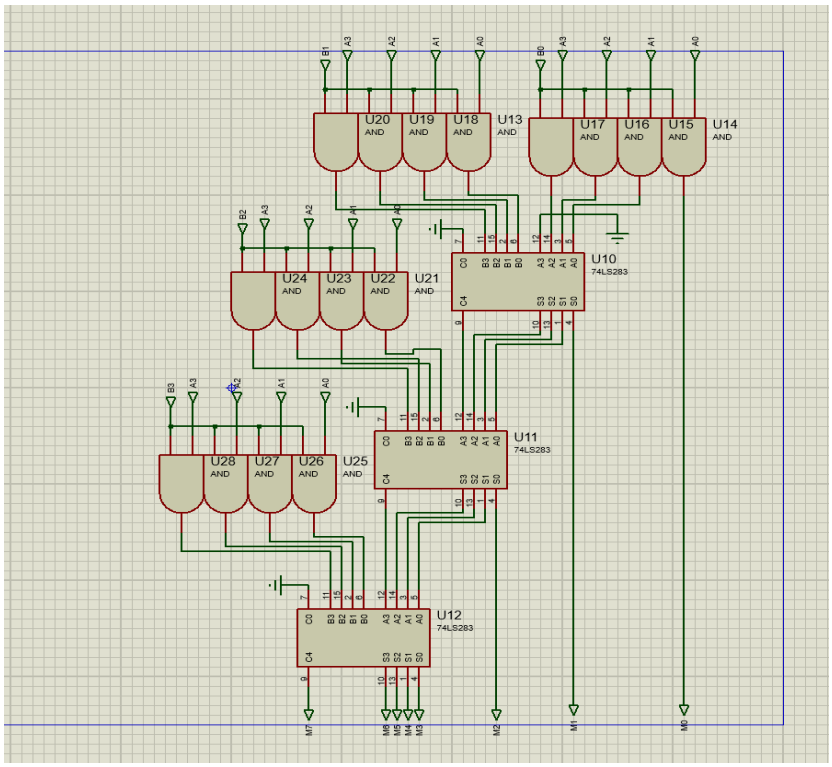




4.Multiblication Operation

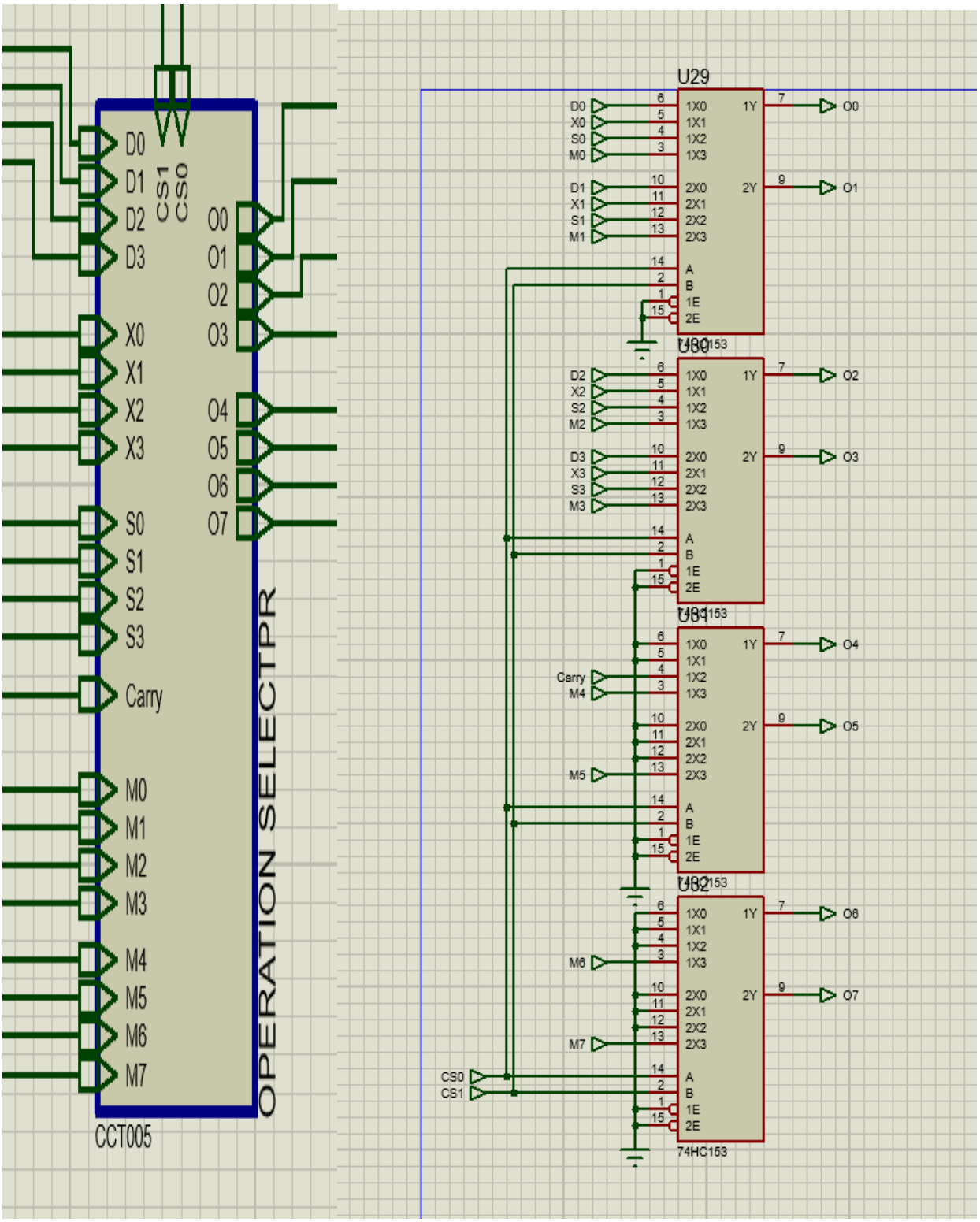
A multiplication operation for binary numbers, using a combination of AND gates and adders to generate partial products, then summing them





5. Operation Selector

A control unit that selects the desired arithmetic or logical operation (AND, XOR, ADD, or MUL) based on input control signals.



Advantages of an ALU:

1. **Versatility**:
    - The ALU can perform a wide variety of arithmetic (addition, subtraction), logical (AND, OR, XOR), and shift operations. This makes it a fundamental building block for any digital processor, including CPUs and microcontrollers.
2. **Speed**:
    - ALUs are highly optimized to perform mathematical and logical operations in parallel, offering fast execution times. This is essential for high-performance systems.
3. **Compact Design**:
    - ALUs are often designed as compact hardware components. They are integrated into processors and can execute complex operations in just a few clock cycles, making them space-efficient.
4. **Parallelism**:
    - The ALU can perform multiple operations simultaneously, increasing throughput. For example, an ALU can process multiple data streams in parallel if designed to support such operations (like vector processors).

Disadvantages of an ALU:

1. **Limited Scope**:
    - The ALU's primary role is to handle basic arithmetic and logical operations. More complex tasks, such as control flow or high-level algorithmic operations, require additional units like the Control Unit (CU) and memory management, which increases complexity.
2. **Hardwired Design**:
    - Many ALUs are hardwired to perform a specific set of operations. If more operations are needed, the ALU must be redesigned, which is not as flexible as software implementations.
3. **Latency**:
    - Although ALUs can perform operations very quickly, in larger systems with a high number of operations or pipelines, the time it takes to pass data between stages in a processor (due to factors like clock cycles and memory access times) can add latency.
4. **No Memory Operations**:
    - The ALU itself does not interact with memory directly, so data must be fetched from memory separately, which might introduce delays or additional complexity in systems with limited data bandwidth.

Applications of an ALU:

1. **Central Processing Units (CPUs)**:
    - **Application**: ALUs are integral components of CPUs in personal computers, mobile devices, servers, and embedded systems. The ALU performs all the arithmetic and logical operations that are required for program execution. For example, calculations in scientific applications, gaming graphics, and running algorithms.
    - **Example**: In a general-purpose processor like an Intel or AMD CPU, the ALU performs tasks like adding values in programs or comparing integers.
2. **Graphics Processing Units (GPUs)**:
    - **Application**: GPUs, especially in parallel computing applications, leverage many ALUs to perform vector and matrix operations. These operations are essential for rendering graphics and performing computations in scientific simulations.
    - **Example**: ALUs are used in a GPU to process complex graphics, like image rendering or physics simulations in 3D games or machine learning workloads.
3. **Digital Signal Processing (DSP)**:
    - **Application**: ALUs are crucial in digital signal processing systems, where they perform mathematical operations on signals (such as filtering or Fourier transforms) to analyze or modify digital signals.
    - **Example**: In a DSP chip, the ALU can be used to perform operations on audio signals, like filtering, amplification, or sound modulation.
4. **Embedded Systems**:
    - **Application**: ALUs are widely used in embedded systems, especially when performing basic mathematical and logical operations. For example, in automotive ECUs (electronic control units) and microcontrollers, ALUs process sensor data and control actuators.
    - **Example**: In a microcontroller for controlling a robot, the ALU might compute the robot's position or the motor speed using mathematical operations.
5. **Cryptography**:
    - **Application**: ALUs are used in encryption and decryption algorithms, which require fast bitwise operations (like XOR, AND, or shifts). The ALU performs these operations to secure data in systems requiring high security.
    - **Example**: In RSA encryption, ALUs are used to perform modular exponentiation, which is a core operation in the encryption and decryption process.