CSE 210

Computer Architecture Sessional

Assignment-1

4-bit ALU Simulation

Section - C1
Group - 02

Group Members:

1. 2105133 - Md. Abir Hossain
2. 2105134 - Md. Shadman Shafie
3. 2105143 - Fatin Ishrak Arian
4. 2105144 - Asikur Rahman
5. 2105150 - Prachurja Dhar

1 Introduction

The Arithmetic Logic Unit (ALU) is a fundamental component in both classical and modern computing architecture. It performs arithmetic and logical operations, serving as the core of all computational tasks. Since its introduction in the early days of digital computing, the ALU has seen significant improvements in terms of complexity and efficiency. With the increasing demand for faster and more reliable computations, the design of ALUs has evolved, accommodating more sophisticated control systems and improved bit-width operations.

In this assignment, we explore the design and simulation of a 4-bit ALU, a simplified but highly illustrative model. The purpose of this report is to simulate how a 4-bit ALU processes binary data, handles control signals, and manages flags that signal specific operation outcomes such as zero results, carry operations, overflow, and signed operations.

1.1 Definition

An Arithmetic Logic Unit (ALU) is a combinational logic circuit designed to perform a set of arithmetic and logical operations on binary inputs. These operations can include addition, subtraction, logical AND, OR, and XOR, among others. The ALU operates by receiving inputs, applying control signals to determine the operation, and outputting both a result and specific flags that provide further information about the result.

1.2 Overview

Figure 1 illustrates a block diagram of a 4-bit ALU. The ALU accepts two 4-bit binary numbers as inputs, denoted as Operand A (A_3, A_2, A_1, A_0) and Operand B (B_3, B_2, B_1, B_0), along with three control bits (C_2, C_1, C_0). The ALU processes these inputs to generate a 4-bit output alongside flags for carry, zero, overflow, and sign outcomes. The control bits dictate which operation (e.g., addition, subtraction, logical operations) is performed on the inputs. Additionally, the ALU updates flags that indicate specific properties of the result, such as sign (S), zero (Z), carry (C), and overflow (V).

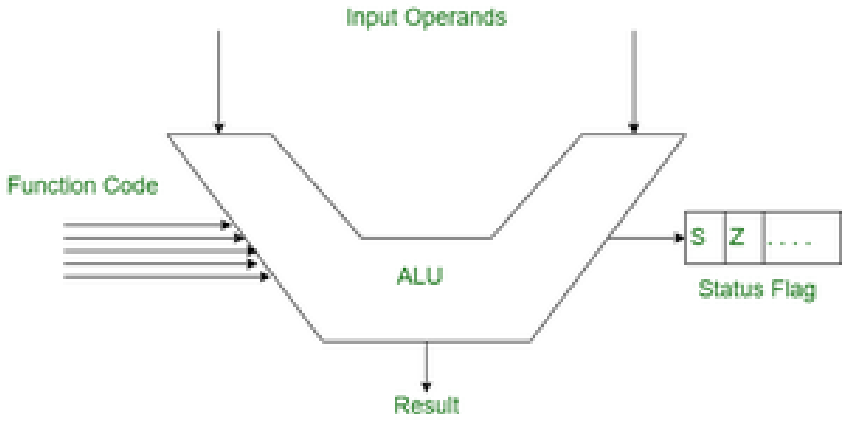


Figure 1: Block Diagram of a 4-bit ALU

The figure demonstrates the flow of data within the ALU, where binary inputs are processed through combinational logic units controlled by the selection bits. The output

flags provide crucial status information used for higher-level decision-making in processor architectures.

1.3 Components

1.3.1 Control Selection

A number of distinct input bits, called control selection bits, are used to determine which operation the ALU should perform on the operands. The most significant bit is often used to distinguish between arithmetic and logical operations. Depending on the specific needs of the system, it may be necessary to include more control bits to differentiate between additional operations.

In a typical ALU, each possible combination of the control bits corresponds to a different operation. It is possible, however, to assign the same operation to multiple control bit combinations if needed.

1.3.2 Operands

The ALU typically receives two inputs, called operands, which are binary numbers of the same size. These operands are processed based on the control selection bits. For arithmetic operations that depend on the order of operands, such as subtraction, the ALU generally subtracts the second operand (B) from the first operand (A).

For logical operations, the ALU processes each bit individually and outputs the result bit by bit. No single bit in one operand influences the output of other bits.

In most cases, operands go through specific circuitry, where each bit of the operand is treated independently and passed to the full adder.

1.3.3 Arithmetic Unit

The full adder is a fundamental component of the ALU. Although its primary function is to add two binary numbers, it can also be used in other operations like subtraction, increment, decrement, and more by manipulating the inputs intelligently. The full adder handles the summation of two operands, along with any carry input, and generates the result along with a carry output.

The ALU also takes special care to avoid race conditions, which can result from simultaneous changes in inputs or outputs and lead to incorrect results. By using mechanisms like a carry-lookahead adder, these race conditions can be minimized, ensuring more accurate arithmetic calculations.

The ALU performs operations such as addition, subtraction with borrow, negation, and transfer. The table below summarizes these common arithmetic operations:

Name of Operation	**Equation**
Transfer A	A
Addition	A + B
Addition with Carry	A + B + 1
Subtraction	A + B' + 1 (A - B)
Subtraction with Borrow	A + B' (A - B - 1)
Negation	A' + 1 (-A)

Table 1: Arithmetic Operations

1.3.4 Logic Unit

Logical operations on n-bit operands are of paramount importance in the realm of computation, just as much as their arithmetic counterparts. These operations encompass basic bitwise functions such as AND, OR, XOR (Exclusive-OR), and One's Complement. However, depending on specific requirements, more intricate logical operations may also be necessitated. The computation of the logical unit can either be integrated within the arithmetic unit or carried out as a standalone circuit. If it is integrated into the arithmetic unit, additional circuit elements may need to be incorporated to merge both units and enable the correct routing of control signals for the selected operation.

In addition to the standard bitwise logical operations, another frequently utilized operation is the Logical Shift. This operation includes right shift, left shift, or even bit rotations. Whether this shifting mechanism is handled within the same unit or in a separate one is generally a matter of design preference.

1.3.5 Output

Similar to the behavior of a full adder, the output in an ALU is produced as an n-bit result, with each bit emerging from distinct outputs. Alongside the primary n-bit output, an additional bit, known as the output carry (C_{out}), is generated. This carry bit represents any overflow from the arithmetic operation. In cases where no overflow occurs, the result produced directly corresponds to the operation chosen by the selection bits.

It is important to recognize that in certain specific operations, the final output must be interpreted without considering the output carry. For example, in subtraction executed via 2's complement, if the output carry is disregarded, the result aligns with the anticipated outcome minus 2^n. Therefore, in order to correctly interpret such operations, it is necessary to ignore the final output carry.

1.3.6 Flags Register

Finally, the ALU provides essential feedback regarding the nature of the computation through a set of status flags, housed in a register referred to as the Flags Register. Typically, this register contains four flags, which are updated automatically based on the results of the ALU operations. These flags can offer valuable information, such as whether the output is negative or if an arithmetic overflow has occurred.

In some cases, these flags must remain constant to ensure correct interpretation of subsequent operations. The table below outlines the equations and significance of each flag:

Name of Flag	Equation	Significance
Sign (S)	S_3	Indicates the most significant bit of the result from the full adder, which helps determine the sign (positive or negative) of the output.
Carry (C)	C_{out}	Represents the output carry from the addition, signifying whether an arithmetic carry has been generated.
Overflow (V)	$C_{out} \oplus C_3$	This flag signals if an overflow has occurred during the calculation, meaning that the output is incorrect in relation to the anticipated result. Corrective measures are often required when this flag is set.
Zero (Z)	$S_3 + S_2 + S_1 + S_0$	Indicates whether the result is zero, meaning all bits of the output are set to 0.

Table 2: Equations and Significance of the ALU Flags

1.4 Design Strategies

The design strategies for implementing computational units can vary significantly based on specific requirements. Different strategies emphasize optimizing distinct parameters, depending on the needs of the system. However, to classify these approaches broadly, two primary methodologies can be distinguished:

1.4.1 Disjoint Implementation

One straightforward strategy involves the separate design of the arithmetic and logical unit circuits. In this approach, the arithmetic unit is implemented using a full adder, while the logical unit executes bitwise operations through the direct application of corresponding logic gates and multiplexers. Subsequently, to integrate both circuits, an additional multiplexer is required to control and decide which unit becomes active based on the operational command.

Although this method is relatively easy to conceptualize and implement, its major drawback lies in the increased number of unnecessary integrated circuits (ICs). This redundancy can lead to an excessive rise in both physical space requirements and overall cost.

1.4.2 Unified Implementation

An alternative methodology aims to merge the logical operations into equivalent arithmetic processes. By executing both operations within the same circuit as the arithmetic unit, this approach reduces redundancy. While this design can become more complex, the payoff in reducing the total number of ICs often justifies the added intricacy. The streamlined circuit design resulting from this integration can offer considerable savings in terms of both hardware size and cost.

2 Problem Specification

The problem assigned is based on the control signals and their corresponding functions in Group 2 of the ALU (Arithmetic Logic Unit) operations table. The following table lists the specific control signals and the operations that are carried out for each combination.

cs2	**cs1**	**cs0**	**Operation**	**Description**
0	0	0	Sub with borrow	$A + \overline{B}$
0	0	1	Add	$A + B$
0	1	0	Transfer A	A
0	1	1	OR	$A \vee B$
1	X	0	Increment A	$A + 1$
1	X	1	NEG A	$\overline{A} + 1$

Table 3: Assigned Operations and Mathematical Descriptions for Group 2

In this context, the operations performed by the ALU are determined by the combination of control signals (cs2, cs1, cs0). Each combination triggers a different operation, with details provided in the table above. The operations in Group 2 range from basic arithmetic (addition, subtraction) to logical operations (OR) and data transfers (Transfer A). There are also unary operations such as incrementing or negating the value of register A.

3 Detailed Design Steps with K-Maps

3.1 Design Steps

1. Here we have performed five types of arithmatic operations(Sub with Borrow, Add, Transfer A, Increment A and Neg A) and one logical operation (OR).

2. For the adder, the first input X_i is either $A \oplus C_{s2}C_{s0}$ or $A \vee B$. We have used a 2×1 multiplexer for each bit. The MUX has S_{x0} as selection bit. The selection bit is controlled by control bits(C_{s2},C_{s1},C_{s0}).The details is shown in the truth table and K-Map.

3. Y_i is either $\overline{B}$ or B. We have used a 2×1 multiplexer for each bit. The MUX has S_{y0} as selection bit and $S_{y_{en}}$ as enable bit. The selection bit is controlled by control bits(C_{s2},C_{s1},C_{s0}).The details is shown in the truth table and K-Map.

4. The input carry(C_{in}) of the adder IC is either 0 or 1. It surprisingly is identical to C_{s2}.The details is shown in the truth table and k-Map.

5. Zero flag, ZF is computed by adding the 4 output bits using 3 OR gates and then inverting $O_0 + O_1 + O_2 + O_3$ using 1 NOT gate .

6. Two 4-bit parallel adder ICs are used in-order to extract C_3 and calculate the overflow flag.

7. The carry flag (C) is obtained from S_2 of the second parallel adder by setting setting the $A_2 = 0$ and $B_2 = 0$.

8. The sign flag (SF) is obtained from the MSB of the sum. (S_1) of the second parallel adder is the MSB.

9. For the overflow flag (OF), we need C_{out} and C_3. C_{out} is accessible from S_2 of the second Adder IC. C_3 is accessible from S_4 of first adder IC. The calculations are as follows:

$$\text{For the first adder}$$
$$S_4 = A_4 \oplus B_4 \oplus C_3$$
$$A_4 = B_4 = 0\text{,then}$$
$$S_4 = C_3$$
$$\text{For the second adder}$$
$$A_1 = X_4$$
$$B_1 = Y_4$$
$$C_{in} = S_4\text{(of the first adder)}$$
$$\text{Therefore, } C_1 = C_{out}$$
$$\text{Now, } S_2 = A_2 \oplus B_2 \oplus C_1$$
$$A_2 = B_2 = 0\text{,then}$$
$$S_2 = C_1 = C_{out}$$
$$\text{Finally, } OF = C_{out} \oplus C_3$$

3.2 K-Maps

We will be following Table 3 to construct the K-maps for selection bits of multiplexers and C_{in}. The IC of the parallel adder takes X_i, Y_i, and C_{in} as inputs. We need X_i as A_i, or its complement, or its logical changes with B_i. These values are received as X_i (the output of the multiplexer), and the K-map for the selection bits (S_x) of the multiplexer is as follows:

3.2.1 K-map for S_x

C_{s2} \ $C_{s1}C_{s0}$	00	01	11	10
0	0	0	1	0
1	0	0	0	0

We can easily express S_x as sum of minterms: $S_x = \overline{C_{s2}}C_{s1}C_{s0}$

3.2.2 K-map for S_y

C_{s2} \ $C_{s1}C_{s0}$	00	01	11	10
0	0	1	d	d
1	d	d	d	d

We can easily express S_y as sum of minterms: $S_y = C_{s0}$

3.2.3 K-map for $S_{y_{en}}$

C_{s2} \ $C_{s1}C_{s0}$	00	01	11	10
0	0	0	1	1
1	1	1	1	1

We can easily express $S_{y_{en}}$ as sum of minterms: $S_{y_{en}} = C_{s1} + C_{s2}$

3.2.4 K-map for C_{in}

C_{s2} \ $C_{s1}C_{s0}$	00	01	11	10
0	0	0	0	0
1	1	1	1	1

We can easily express C_{in} as sum of minterms: $C_{in} = C_{s2}$

Functions	$\mathbf{X}_i$	$\mathbf{Y}_i$	$\mathbf{C}_{in}$
$A + \overline{B}$	A	$\overline{B}$	0
$A + B$	A	B	0
A	A	0	0
$A \vee B$	$A \vee B$	0	0
$A + 1$	A	0	1
$\overline{A} + 1$	$\overline{A}$	0	1

Table 4: Truth Table with X_i, Y_i, C_{in}

cs0	**cs1**	**cs2**	**Functions**	$\mathbf{X}_i$	$\mathbf{S}_x$
0	0	0	$A + \overline{B}$	A	0
1	0	0	$A + B$	A	0
0	1	0	A	A	0
1	1	0	$A \vee B$	$A \vee B$	1
0	X	1	$A + 1$	A	0
1	X	1	$\overline{A} + 1$	$\overline{A}$	0

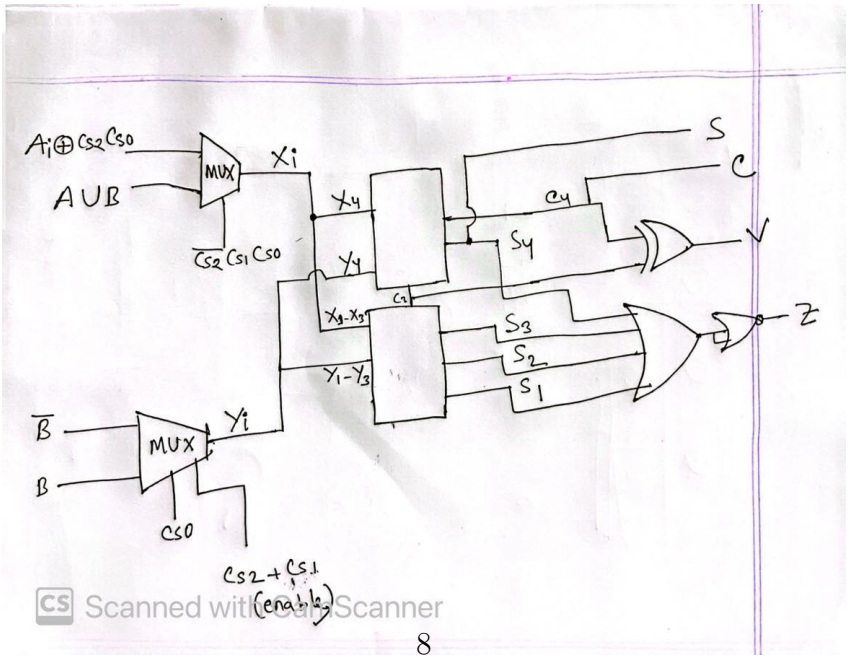
Table 5: Truth Table with X_i and the MUX selection bits S_x

cs0	**cs1**	**cs2**	**Functions**	$\mathbf{Y}_i$	$\mathbf{S}_y$	$\mathbf{S}_{y_{en}}$
0	0	0	$A + \overline{B}$	$\overline{B}$	0	0
1	0	0	$A + B$	B	1	0
0	1	0	A	0	X	1
1	1	0	$A \vee B$	0	X	1
0	X	1	$A + 1$	0	X	1
1	X	1	$\overline{A} + 1$	0	X	1

Table 6: Truth Table with Y_i and the MUX selection bits S_y, $S_{y_{en}}$

4 Truth Table

5 Block Diagram



CS Scanned with CamScanner

Figure 2: Block Diagram

6 Complete Circuit Diagram

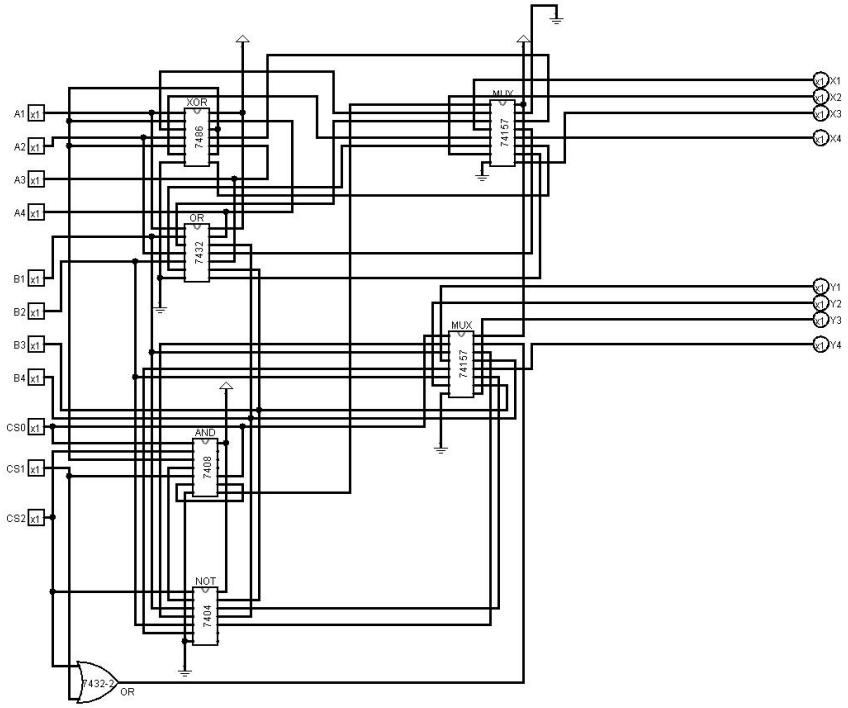


Figure 3: Input Processing Unit

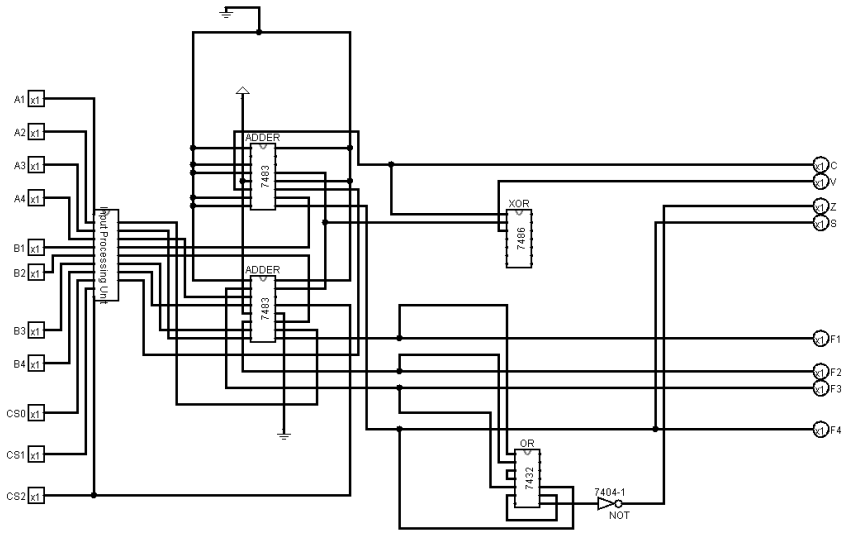


Figure 4: Arithmetic Logic Unit

7 ICs Used with Count as a Chart

IC	Quantity
IC 74157	2
IC 7408	1
IC 7432	2
IC 7404	1
IC 7486	2
IC 7483	2
Total	10

8 Simulator Used Along The Version Number

Logisim - 2.7.1

9 Discussion

The simulation and design of the 4-bit ALU reveal several key insights into its operational efficiency and the complexities involved in its architecture. As observed, the ALU effectively performs a range of arithmetic and logical operations, demonstrating its versatility as a fundamental building block in digital systems.

One of the primary observations is the significance of control signals in determining the operations performed by the ALU. The different combinations of control signals (cs2, cs1, cs0) allow the ALU to execute various tasks, from basic arithmetic to more complex logical operations. This flexibility highlights the importance of a well-defined control unit that efficiently manages the flow of operations based on the input signals.

Moreover, the design strategies implemented—both disjoint and unified—illustrate the trade-offs between complexity and efficiency. While the disjoint approach allows for clear separation of functions, it may lead to increased resource usage and complexity in terms of circuit design. In contrast, the unified approach, though more intricate, offers significant savings in terms of hardware resources and integration, making it a more favorable option in modern ALU design.

Another critical aspect is the handling of flags and status indicators, which are crucial for subsequent computational decisions. The flags (Sign, Carry, Overflow, and Zero) provide essential feedback that can guide further operations in the processor. Understanding how these flags are updated during operations is vital for maintaining the integrity of computations, particularly in cases of conditional branching or arithmetic overflow.

In conclusion, the 4-bit ALU simulation not only serves as an educational exercise in understanding the core functionalities of ALUs but also underscores the importance of efficient design principles and effective control mechanisms in achieving reliable computational outcomes. Future enhancements could explore larger bit-width ALUs and more complex operations, reflecting the ongoing evolution of computational architectures.

10 Contribution

10.1 Developing Simplified Expressions and designing logic

Generating equation for x_i , s_x and prepare truth table, K Map for s_x and block diagram for x_i : 2105150,2105144
Generating equation for y_i ,s_y, S_{yen} and prepare truth table, K Map for s_y ,s_{yen} and block diagram for y_i : 2105133, 2105134
Generating equation for c_{in} and prepare K map for c_{in} : 2105143
Merging BLock diagrams and generate equations for zero flag and overflow flag : 2105143

10.2 Minimize IC Count

Developing the idea of using mux in equation of x during logical operation to minimize IC Count : 2105150,2105144
Using mux in equation of y instead of basic gates : 2105133,2105134
Using two adders to obtain the c_3 bit instead of xor gate : 2105143

10.3 Software Implementation

Developing Input processing unit : 2105133, 2105134
Developing Arithmetic part of circuit using input processing unit as a module : 2105150,2105144
Developing the flag part and testing the whole circuit : 2105143

10.4 Hardware Implementation

Developing switching circuit for controlling 11 input bits through 11 6 pin DPDT switches and assisting in making the output viewer circuit through LEDS : all members
Developing the input processor unit circuit which generates x_i , y_i, c_{in} : 2105133,2105134
Developing the final part of circuit using full adders, or , xor and not gate : 2105144,2105150, 2105143
Testing : All members

10.5 Report Writing

All sections except contribution : 2105143
Contribution : 2105150