Bangladesh University of Engineering and Technology

CSE 209

Computer Architecture Sessional

Offline 1: 4-bit ALU Simulation

Section - B2
Group - 2

2105091 - Shemanty Mahjabin
2105100 - Satak Kumar Dey
2105101 - Nahian Reza Nuhas
2105103 - Mahabuba Sharmin Mim

October 21, 2024

1 Introduction

The Arithmetic Logic Unit, or ALU, is an unsung hero in the world of computers. This vital component serves as the heart of computational operations, where the elegance of arithmetic meets the precision of logic. we can think of the ALU as a skilled craftsman, diligently working behind the scenes to perform essential arithmetic and logical tasks that keep our digital world running smoothly. The ALU is a digital circuit designed to carry out the heavy lifting of basic arithmetic tasks such as addition, subtraction, multiplication, and division, along with logical operations like AND, OR, NOT, and XOR. These operations are fundamental to all computational processes, making the ALU an indispensable element in the execution of computer programs. ALU functions can be divided into two main categories: arithmetic and logic. Arithmetic operations focus on numerical calculations, forming the backbone of mathematical computations in a computer. Logic operations handle decision-making and comparisons based on binary logic, which are crucial for controlling program flow and executing decisions in code.
The ALU works by taking binary data as input, processing it according to the instructions from the CPU's control unit, and then delivering the output. This seamless operation highlights the advanced engineering that powers modern computing systems. In this project, we're diving into a hands-on assignment to design and implement a 4-bit Arithmetic Logic Unit (ALU). Our journey will explore both software simulations and hardware construction, giving us a taste of the practical applications of our learning.
Efficiency is a major focus of this assignment, particularly in minimizing the number of integrated circuits (ICs) used. By adopting a minimalist approach, we aim to create a design that is functional yet optimized for component usage. This challenge encourages us to innovate and think critically about our design choices.
In the sections ahead, we'll break down our journey in creating a 4-bit ALU. We'll start with 'Problem Specification and Assigned Instructions', move on to 'Design Steps with K-Maps', and then explore 'Truth Tables', 'Block Diagrams', and 'Complete Circuit Diagrams'. We'll also provide a detailed 'Chart of ICs Used', mention the 'Simulator and Version' used for testing, and wrap up with comprehensive 'Discussions' along with an acknowledgment of 'Each Member's Contribution' to this assignment.

2 Problem Specification with assigned instructions

We are to design a 4-bit ALU with three selection bits cs0, cs1, and cs2 for performing the following operations: In addition, we need to implement the four status outputs, or flags, denoted as C (Carry

cs_2	cs_1	cs_0	Function	Output
$\times$	0	0	Add with carry	$F = A + B + 1$
0	0	1	Subtraction	$F = A - B$
0	1	$\times$	Transfer A	$F = A$
1	0	1	XOR	$F = A \oplus B$
1	1	0	Complement A	$F = \overline{A}$
1	1	1	Decrement A	$F = A - 1$

Table 1: Functional Design Specification

Flag), S (Sign Flag), V (Overflow Flag), and Z (Zero Flag). Their representations convey specific meanings: C contains the output carry C_{out} of the operation, S contains the sign of the result of the operation, V indicates any overflow has occurred due to the operation, and Z indicates whether the 4-bit of the result is 0 or not.

Flags will be affected as per the rules of Assembly Language. Our ALU is required to output flags under the simplified rules:
For NOT Operation: After the NOT operation, which makes the result 0000, if Z becomes 0 from 1, it will not be accepted. However, if Z becomes 1 or if the Z flag remains unchanged, both will be accepted. If S remains unchanged or reflects the highest order bit of the result, both will be accepted. But if the S flag is changed and it is changed to a wrong value, it will not be accepted.
For AND/OR/XOR Operation: C and V should be cleared after the operation. S and Z should be changed according to the output.
Here we are allowed to use only 2 input SSI (AND, OR, NOT, XOR, etc.) and MSI (MUX, Decoder, Adder, etc.). We are required to build an efficient design with the minimum possible number of ICs.
For simulation, we used Logisim (circuit analyzing software).

3 Detailed Design Steps with K-Maps

3.1 Design Steps

We are working with two 4-bit binary operands, A and B, where each bit is labeled as follows:

- Operand A: A_3, A_2, A_1 , A_0
- Operand B: B_3, B_2, B_1 , B_0

Here, A_3 and B_3 are the most significant bits (MSBs) of the respective operands.
To control which operation is being performed, we have three selector bits (cs_2, cs_1, cs_0). These bits determine whether we are performing an arithmetic or a logical operation. There are two categories of operations:

- **Arithmetic Operations:**
  - **Addition with Carry (A + B + carry-in):** Performs the addition of operands A and B, with an additional carry-in bit. The inputs are A and B, and the carry-in is set to 1.
  - **Subtraction (A - B):** Subtracts the 4-bit binary number B from A by adding A and the 2's complement of B. The inputs are A and the complement of B (B'), with the carry-in set to 1 to facilitate subtraction.
  - **Transfer (A):** Outputs the value of operand A directly, disregarding operand B. In this operation, the inputs are A and 0000, with the carry-in set to 0.
  - **Decrement (A - 1):** Decrements the value of A by 1. The inputs are A and 1111 (the 2's complement of 1), and the carry-in is set to 0, ensuring that the operation results in $A - 1$.
- **Logical Operations:**
  - **Exclusive-OR (A $\oplus$ B):** Performs a bitwise XOR operation between A and B, where each bit is set to 1 if the corresponding bits of A and B differ. The inputs for this operation are A and B, with the carry-in bit set to 0. Flags are utilized to ensure that the carry bit is reset to 0 during each bitwise operation. For further details, refer to the 3.1.

– **Complement ($\overline{A}$):** Generates the bitwise complement of operand A, where each 1 is flipped to 0 and each 0 is flipped to 1. The input bits for this operation are A and 1111. In the full adder, the XOR operation between a_i and 1 will produce a_i'. Flags are used to set the carry bit to 0 for each bit operation. For further details, refer to the 3.1.

We also have to generate some flags . Such as :
Carry Flag (C): The carry flag, often represented as C, tells us if there was an overflow from the arithmetic unit during calculations. When we perform logical operations, we intentionally set the carry flag to 0. This approach ensures that we won't run into overflow issues while working with logic.

Sign Flag (S): The sign flag, known as S, provides information about the result's sign. It's derived directly from the most significant bit (F4). So, if F4 is 1, we know the result is negative; if it's 0, the result is positive.

Overflow Flag (V): The overflow flag, or V, helps us understand if an overflow has occurred during calculations. This flag can be figured out from the intermediate results. In logical operations, since the carry flag is held at 0, we don't have to worry about overflow here.

Zero Flag (Z): The zero flag, represented as Z, is quite straightforward. It gets set when all four bits of the result (F4, F3, F2, F1) are zero. If even one of those bits is non-zero, the flag is turned off. You can think of it like this:

$$Z = \overline{F_4} \cdot \overline{F_3} \cdot \overline{F_2} \cdot \overline{F_1} = \overline{(F_4 + F_3 + F_2 + F_1)}$$

This equation captures the essence of the zero flag, letting us know if the result is indeed zero.

Using Flags to Handle A/L operations Correctly

In our design, we utilize two 4-bit full adders to execute the arithmetic and logical operations outlined previously. A crucial control bit, denoted as l', determines the nature of the operation. When $l' = 0$, we perform a logical operation; when $l' = 1$, we proceed with an arithmetic operation.
To facilitate this control, the carry-in signal (Cin) is combined with l' using an AND operation. This configuration provides an efficient mechanism for managing the transition between arithmetic and logical operations.
For the first full adder, the inputs are organized as follows: Y_1, S_1, 0, Y_0 for one set of inputs, and A_1, l', 0, A_0 for the other. The outputs from this adder include F_1, S_2, S_1, and F_0, where S_1 serves as the carry-out signal C_1.
In this context, it's essential to ensure that C_2 remains zero for the second bit operation. This can be mathematically represented as follows:

$$C_2 = C_1 \cdot 0 + C_1 \cdot 0 + 0 \cdot 0$$

The carry C_3 is significant because it indicates whether we are conducting an arithmetic or logical operation. The relationship can be expressed as:

$$C_3 = C_1 \cdot l' + C_1 \cdot C_2 + l' \cdot C_2$$

Given that $C_2 = 0$, this simplifies to:

$$C_3 = C_1 \cdot l'$$

This means that if $l' = 1$, then $C_3 = C_1$, allowing the operation to proceed with inputs Y_1 and A_1, resulting in an arithmetic operation. On the other hand, if $l' = 0$, C_1 is set to 0, leading to a standard logical operation.
The interaction between the carry output from the first adder, the control bit l', and the carry-in signal plays a pivotal role in determining whether the carry should propagate to subsequent operations. This design elegantly manages the execution of both arithmetic and logical operations.
For the second full adder, we apply the same methodology to derive the outputs F_2 and F_3. This consistent approach across both adders ensures that our system can effectively perform the desired arithmetic and logical functions.

3.2 Optimization techniques

As our goal is to minimizing gate numbers , so we took some steps .These are listed below :

- We used 4*1 mux So that we can choose Y bit effectively
- Instead of using $Z = \overline{(F_4 + F_3 + F_2 + F_1)}$ equation we used $Z = \overline{(F_4 + F_3)} \cdot \overline{(F_2 + F_1)}$ to lessen the gate number

3.3 Equations For MUX

Y_i represents modified representation of B_i and thus Y represents B. So, the following 4 combinations can be obtained: Keeping B as it is ($Y_i = B_i$), Inverting all bits of B ($Y_i = \bar{B_i}$), Changing each bit of B to 0 ($B_i = 0$), and Changing each bit of B to 1 ($B_i = 1$).

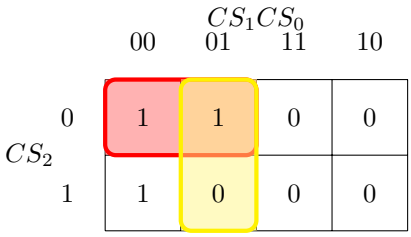
cs_2	cs_1	cs_0	Y_i	Function
0	0	0	B	$Y = cs_0 \oplus B$
0	0	1	$\overline{B}$	
0	1	0	0	Y=0
0	1	1	0	
1	0	0	B	Y=B
1	0	1	B	
1	1	0	1	Y=1
1	1	1	1	

3.4 K-Maps

We will be following table 2 to construct the K-Maps. We are keeping A fixed (hence,$X_i = A_i$) and changing B to generate different operations.

3.4.1 K-Map for Z_{in}

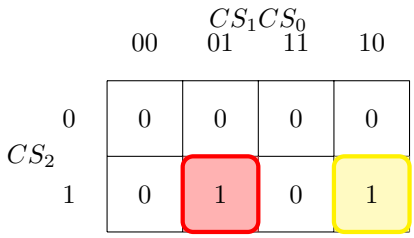
Z_{in} represents the modified carry input, denoted as C_{in}, provided to the adder circuit based on the specific operation being performed. The modification of Z_{in} depends on the nature of the operation being carried out in the circuit.



$$Z = \overline{CS_1} \cdot \overline{CS_0} + \overline{CS_2} \cdot \overline{CS_1}$$
$$= \overline{(CS_1 + CS_2 \cdot CS_0)}$$

3.4.2 K-map for l

We also maintain a flag l to determine the type of operation, whether it is an arithmetic or logical operation. We set l = 1 for arithmetic operations and l = 0 for logical operations.



$$l = CS_2(CS_1 \cdot \overline{CS_0} + \overline{CS_1} \cdot CS_0)$$
$$= CS_2(CS_1 \oplus CS_0)$$

4 Truth Table

For a clearer interpretation of the variables employed, we constructed this truth table:

cs_2	cs_1	cs_0	X_i	Y_i	Z_{in}	Function	l
0	0	0	A_i	B	1	$A+B+1$	0
0	0	1	A_i	$\overline{B}$	1	$A+\overline{B}+1$	0
0	1	0	A_i	0	0	A	0
0	1	1	A_i	0	0	A	0
1	0	0	A_i	B	1	$A+B+1$	0
1	0	1	A_i	B	0	$A \oplus B$	1
1	1	0	A_i	1	0	$\overline{A}$	1
1	1	1	A_i	1	0	$A-1$	0

5 Block Diagram

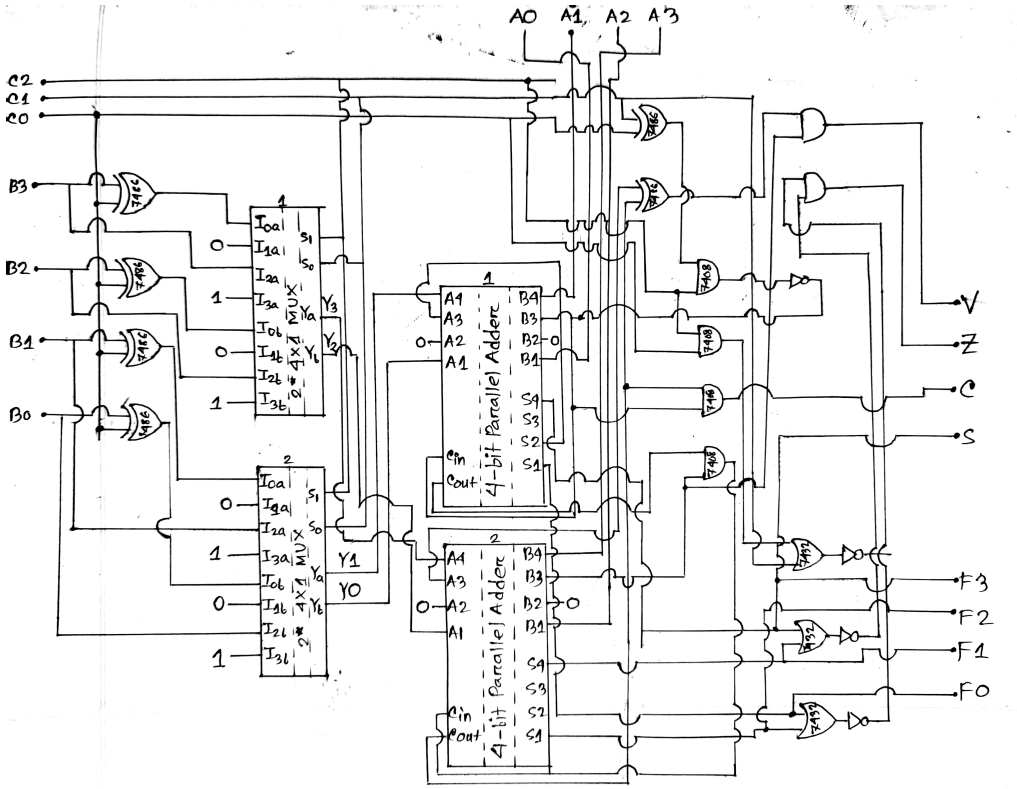


Figure 1: Block Diagram of ALU

6 Circuit Diagram

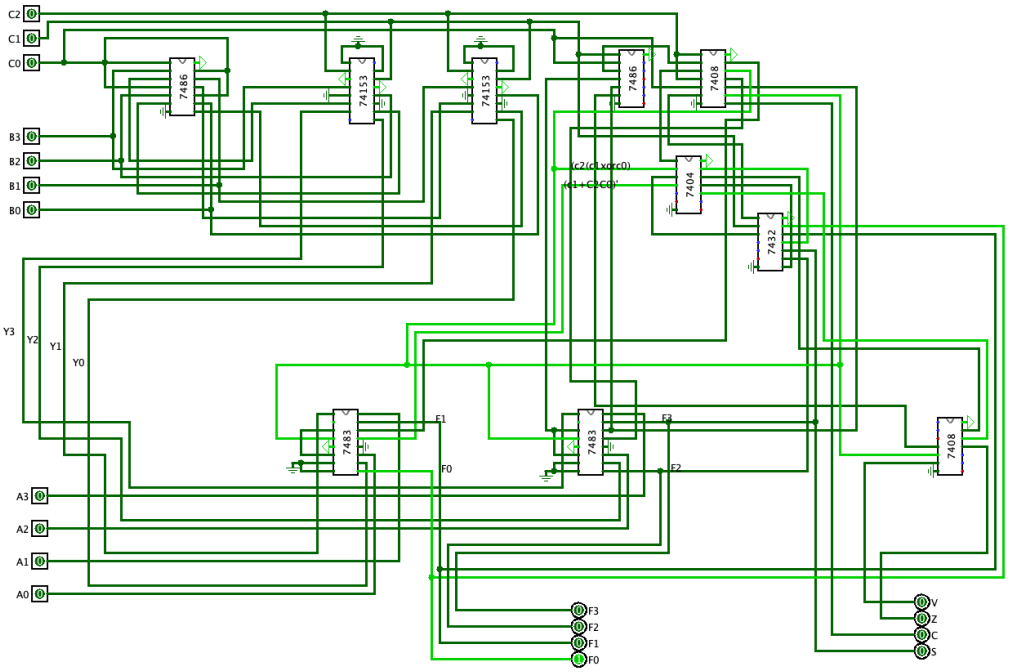


Figure 2: Circuit Diagram (logisim)

7 ICs used with count as a chart

In this section, the integrated circuits (ICs) utilized in the implementation, along with their respective quantities, are presented in a concise chart.

IC Number	IC Name	Quantity
SN74HC08N	Quad 2-Input AND	2
SN74HC32N	Quad 2-Input OR	1
SN74HC86N	Quad 2-Input XOR	2
SN74HC04N	Hex Inverter	1
SN74HC83N	4-bit Binary Full Adder	2
SN74HC153N	Multiplexer	2

Table 2: ICs Used with Quantity

8 The simulator used along with the version number

Software: Logisim
Version: Logisim-win-2.7.1

9 Discussion

Throughout the process of designing and implementing a 4-bit ALU capable of handling basic arithmetic and logical operations, we encountered several practical challenges. The following points summarize our experience:

- Initially, we used 6-pin switches, but they weren't fitting tightly into the breadboard. This led to unstable connections, making it hard to reliably test the circuit. We later switched to 3-pin switches, which fit better and provided more consistent results.
- During our first test, when we connected the inputs to check the circuit, we noticed that the Arduino powering the setup would shut off after about a minute. We tried using a USB connection, but the problem persisted. After careful inspection, we discovered a short circuit in one of the connections. Once we addressed this, the Arduino remained powered, and the issue was resolved.
- We used small wires throughout the project to keep the setup neat and minimize the clutter caused by the many connections. This helped us avoid confusion while debugging the circuit and ensured that we could clearly trace each connection.
- In one of the more time-consuming setbacks, two of our ICs were found to be defective, leading to incorrect readings. It took some time to isolate the issue, but once identified, we replaced the faulty ICs and the circuit began to function as expected.
- Another problem arose with the LED bulbs. A few of the LEDs we were using were defective, causing incorrect outputs when connected to the circuit. Initially, we thought the circuit design was flawed, but after testing and replacing the faulty LEDs, the output was as expected.
- We made sure to double-check all connections to verify that every component was placed correctly and functioning properly. This final step was crucial in ensuring that our ALU operated as intended.

In the end, we successfully implemented the 4-bit ALU, overcoming various hardware issues along the way. The experience taught us the importance of thorough testing and careful assembly, as even minor hardware problems can lead to significant setbacks. Through perseverance, we gained a deeper understanding of digital circuit design and hardware troubleshooting, both of which are invaluable skills in electronics projects.

10 Contribution of Each Member

Circuit Design : 2105103
Logism : 2105103 , 2105091
Hardware Design: 2105091 , 2105100 , 2105102 , 2105103
Report Writing: 2105091 , 2105100 , 2105102 , 2105103