CSE 306
Computer Architecture Sessional

Assignment-1
4-bit ALU Simulation

Section - A1
Group - 01

Group Members:

1. 2005001 - Anik Saha
2. 2005012 - Abrar Jahin Sarker
3. 2005013 - Al Muhit Muhtadi
4. 2005017 - Abdullah Muhammed Amimul Ehsan
5. 2005023 - Jaber Ahmed Deeder

1 Introduction

First proposed by John Von Neumann, ALU stands as a cornerstone in the realm of computer architecture ever since its coinage in 1945. With diminishing transistor dimensions, implementing and incorporating wider-word ALUs in tiny microprocessors has kept becoming easier and cheaper. In the light of our own experimentation, this report aims at exploring the strategies of ALU design in considerable depth. We first provide a modest introduction to the theoretical foundation. Afterwards we present our methodology, outcomes and scopes of further exploration.

1.1 Definition

An arithmetic logic unit abbreviated as ALU is a combinational digital circuit that performs arithmetic and bitwise operations on integer binary numbers.

1.2 Overview

Figure 1 presents a block diagram of a 4-bit ALU which takes two 4 bit numbers A($A_3A_2A_1A_0$) and B($B_3B_2B_1B_0$) as inputs and 3 control selection bits cs_2, cs_1, cs_0. After performing the operation, it outputs 5 bits namely, $C_{out}, S_3, S_2, S_1, S_0$ Additionally, it updates the 4 flags named S, C, V, Z in the Flags register accordingly.

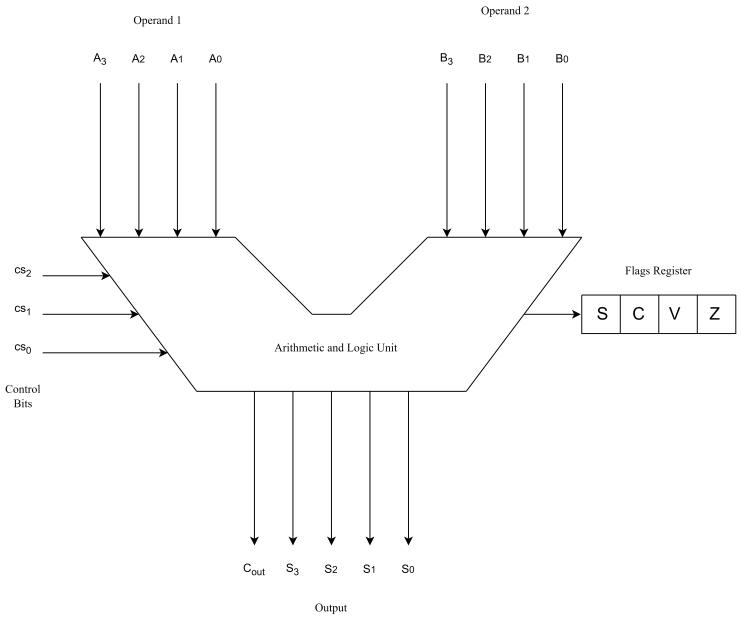


Figure 1: Block Diagram of 4 bit ALU

To generalize, the inputs for an n-bit ALU are two n-bit binary numbers and k control selection bits, where the value of k is design specific. And it outputs an n-bit binary number S and one output carry. Alongside these, it calculates the 4-flags and store them in the Flags register.

1.3 Components

1.3.1 Control Selection

A number of mutually independent distinct input bits are used as control selection bits which determine the nature of the operation to be performed on the operands. Sometimes, one of these bits, usually the most significant one, is used solely to distinguish between arithmetic and logical operations. However, cases depending upon the requirements may well appear where one single bit is not adequate to meet this purpose.

Each of the 2^k combinations of the possible inputs in the control bits is mapped to a single operation defined on the operands. It is, however, permissible to have multiple combinations of selection bits to map to the same operation.

1.3.2 Operands

Two independent operands usually consisting of the same number of bits are fed into the ALU through circuitry. All operations determined by the control selection bits are applied on these two operands.

For operations which do not impose symmetry upon the operands, for instance, subtraction, order of the operation matters. In this case, usually subtraction means subtraction of B from A.

For bitwise logical operations, the corresponding action is performed bit by bit and the output is also generated bit by bit and no two distinct bits affect others' outputs.

Operands are not usually used directly into the circuitry. Most often, each bit of an operand goes through identical circuitry and thus produces the corresponding input bits which need to be passed into the full adder.

1.3.3 Arithmetic Unit

Full adder, although usually interpreted as a circuitry for adding two n-bit numbers and generating the summation with output carry, can be utilized in multifaceted ways, thereby allowing a designer to efficiently perform seemingly disjoint arithmetic operations by intelligently manipulating the different input bits of a full adder. Regarding the full adder, one significant consideration is that, ideally the full-adder must include an internal carry-lookahead mechanism, Otherwise race conditions may appear and may pose to be an impediment towards accurate calculations.

Common arithmetic operations include transfer, addition, addition with carry, increment, decrement, subtraction, subtraction with borrow, negation (2's complement). The mathematical interpretation of the aforementioned operations is presented in the table below:

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

Bitwise logical operations on n-bit operands is just as important in terms of computation as are the arithmetic ones. These operations include bitwise AND, OR, Exclusive-OR (XOR) and One's complement. But depending on requirements, more complex logical operations may as well be needed. The calculations of the logic unit may either be carried out in a separate circuit or may be incorporated into the arithmetic unit after careful manipulations. In the former case, additional circuit elements need to be introduced to merge the two units and to run the appropriate unit for respective selection of control bits.

Alongside bitwise logical operations, another type of bitwise operation named Logical Shift is often required. This may involve right shift, left shift or bit rotation. Whether this operation will be implemented inside a separate unit or inside the ones mentioned above is often a matter of design choice.

1.3.5 Output

Just as it happens in full adder, the output is generated as an n-bit number in the n distinct bits of the outputs of full adder. Additionally an extra bit named output carry (C_{out}) is generated. If overflow does not happen for the particular scenario, then the output corresponds to the result of the intended operation that is defined by the combination of the selection bits.

It is to note that, for some particular operations, the output is to be perceived after disregarding the output carry. For instance, since subtraction is carried out as 2's complement, the output, when C_{out} is considered, is actually equal to the desired result plus 2^n. Therefore to interpret the correct difference between the operands, one needs to ignore the final output carry.

1.3.6 Flags Register

Last but not the least, an ALU leaves valuable information regarding the overall nature of the computation in a register called Flags Register. Usually there are 4 flags that are updated accordingly after performing the computations on the operands. However, in case of some operations, some particular flags may as well need to be kept constant.

The details regarding the equation and significance of each flag is shown in the table below:

Name of Flag	**Equation**	**Significance**
Sign (S)	S_3	It denotes the most significant bit of the output from full adder. Thus this flag helps determine the sign of the output.
Carry (C)	C_{out}	It denotes the output carry.
Overflow (V)	$C_{out} \oplus C_3$	This flag indicates whether there occurred an overflow while performing the calculation. If this flag is set, it means that the apparent output is not equal to the intended result and corrective measures must be taken while interpreting the output.
Zero (Z)	$\overline{S_3 + S_2 + S_1 + S_0}$	It indicates whether all bits of the output are zero.

Table 2: Equation and Significance of Flags

1.4 Design Strategies

With varying requirements, design strategies may differ from each other by a significant amount. Additionally, not all design processes focus on optimizing the same parameter. However, to broadly classify, the design process may incorporate one of the following two prevalent patterns:

1.4.1 Disjoint Implementation

One simple approach involves separate implementation of arithmetic and logic unit circuits. In this case, the Arithmetic unit is carried out with a full adder and the logic unit calculates the bitwise operations with direct involvement of corresponding gates and multiplexers. Then, to merge these two circuits, an additional multiplexer is needed to differentiate and determine which circuit will be active for which combinations.

While this approach seems easy to understand and implement, the major drawback is that, while combining the two circuits together, it needs a number of unnecessary ICs which may exceed size and cost restrictions.

1.4.2 Unified Implementation

An alternative approach considers converting the logical operations to equivalent arithmetic operations and thereby performing them in the same circuit as of the arithmetic one. Although the design process tends to get a bit complicated in this approach, the tradeoff is well worth it when the reduction in the number of ICs is considered.

2 Problem Specification with Assigned Instructions

Design a 4-bit ALU with three control selection bits cs_0, cs_1, cs_2. The assigned instructions and corresponding combinations of control selection bits are shown in the table below:

Control Signals			Functions	Description
cs2	cs1	cs0		
0	X	0	Add	$A + B$
0	0	1	Neg A	$-A(\overline{A} + 1)$
0	1	1	Add with Carry	$A + B + 1$
1	0	0	Increment A	$A + 1$
1	0	1	AND	$A.B$
1	1	X	XOR	$A \oplus B$

Table 3: Problem Specification

3 Detailed design steps with K-Maps

3.1 Design Steps

1. Here we have to perform four types of arithmetic operations (Add, Negation, Add with carry, Increment) and two types of logical operations(logical AND , logical XOR).

2. For the adder,the first input X_i is either A_i or $\overline{A_i}$ or A_iB_i or $A_i \oplus B_i$.We used a 4×1 multiplexer for each bit. The MUX has S_{x_1} and S_{x_0} as selection bits.The selection bits are controlled by control bits(c_{s_2},c_{s_1},c_{s_0}).The details is shown in the truth table and k-Map.

3. For the adder,the second input Y_i is either B_i or 0. We used a 2×1 multiplexer for each bit. The mux has S_{y_0} as its selection bit.The selection bit is controlled by control bits(c_{s_2},c_{s_1},c_{s_0}).The details in shown in the truth table and k-Map.

4. The input carry(C_{in}) of the adder IC is either 0 or 1.It should be zero for negation, add with carry, and increment operations.This is also controlled by control bits(c_{s_2},c_{s_1},c_{s_0}).The details is shown in the truth table and k-Map.

5. In the arithmetic unit, the adder adds A, B with $C_{in} = 0$ and $C_{in} = 1$ for Add, Add with Carry operations respectively. The multiplexers provide Y_i as a 0 for increment operation.During negation,X_i=$\overline{A_i}$ and Y_i=0 and C_{in}=1.

6. During logical operations,X_i is transferred as Y_i=0 and C_{in}=0.

7. Zero flag, ZF is computed by adding the 4 output bits using 3 OR gates and then inverting $O_0 + O_1 + O_2 + O_3$ by 1 XOR gate($X \oplus 1 = \overline{X}$).

8. The carry flag(C) is directly obtained from C_{out} of the parallel adder

9. The sign flag(SF) is obtained from the MSB of the sum(S_3).

10. For the overflow flag,we needed C_{out} and C_3. C_{out} is directly accessible from the Adder IC. C_3 is calculated as below:

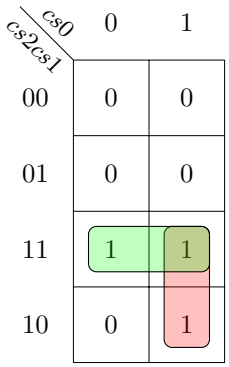
$$
\begin{gathered}
S_3 = X_3 \oplus Y_3 \oplus C_3 \\
S_3 \oplus C_3 = X_3 \oplus Y_3 \\
C_3 = X_3 \oplus Y_3 \oplus S_3 \\
\text{OF} = C_{out} \oplus C_3 \\
\text{OF} = C_{out} \oplus X_3 \oplus Y_3 \oplus S_3
\end{gathered}
$$

3.2 K-maps

We will be following Table 5 and 6 to construct the K-maps for selection bits of multiplexers and C_{in}.

The IC of parallel adder takes X_i and Y_i and C_{in} as input.We need X_i as A_i or its complement or its logical changes with B.This values are received as X_i(output of the multiplexer) and the kmap for selection bits(S_{x_1} and S_{x_0}) of the multiplexer are as follows:

3.2.1 K-map for S_{x_1}

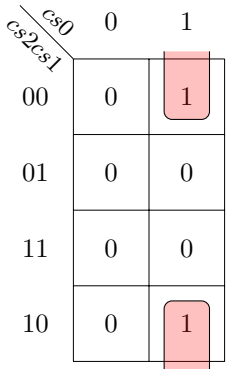


We can easily express S_{x_1} as sum of minterms.

$$S_{x_1} = c_{s_2}c_{s_1} + c_{s_2}c_{s_0}$$

$$S_{x_1} = c_{s_2}(c_{s_1} + c_{s_0})$$

3.2.2 K-map for S_{x_0}

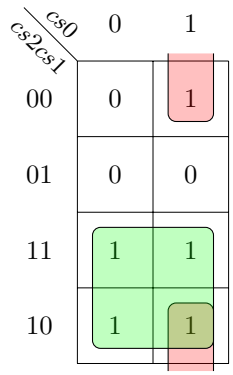


So, there are two minterms.

$$S_{x_0} = \overline{c_{s_1}}c_{s_0}$$

3.2.3 K-map for S_{y_0}

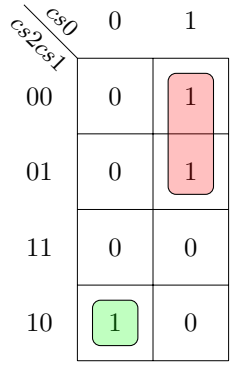
S_{y_0} is the selection bit for the multiplexer that selects B and 0 as input of a 2*1 MUX .



$$S_{y_0} == c_{s_2} + \overline{c_{s_1}} c_{s_0}$$

3.2.4 K-map for C_{in}

It is the input carry bit of the adder used inside arithmetic unit.



$$C_{in} = \overline{c_{s_2}} c_{s_0} + c_{s_2} \overline{c_{s_1}} . \overline{c_{s_0}}$$

$$C_{in} = \overline{c_{s_2}} c_{s_0} + c_{s_2} . \overline{c_{s_1} + c_{s_0}}$$

4 Truth Table

cs2	cs1	cs0	X_i	Y_i	Z_i	C_{in}	Function
0	0	0	A_i	B_i	C_i	0	Add
0	0	1	$\overline{A_i}$	0	C_i	1	NEG A
0	1	0	A_i	B_i	C_i	0	Add
0	1	1	A_i	B_i	C_i	1	Add with carry
1	0	0	A_i	0	C_i	1	Increment A
1	0	1	A_iB_i	0	C_i	0	AND
1	1	0	$A_i \oplus B_i$	0	C_i	0	XOR
1	1	1	$A_i \oplus B_i$	0	C_i	0	XOR

Table 4: Truth Table of X_i, Y_i, Z_i, C_{in}

cs2	cs1	cs0	X_i	S_{x_1}	S_{x_0}
0	0	0	A_i	0	0
0	0	1	$\bar{A}_i$	0	1
0	1	0	A_i	0	0
0	1	1	A_i	0	0
1	0	0	A_i	0	0
1	0	1	A_iB_i	1	1
1	1	0	$A_i \oplus B_i$	1	0
1	1	1	$A_i \oplus B_i$	1	0

Table 5: Truth Table of MUX input for X_i

cs2	cs1	cs0	X_i	S_{y_0}
0	0	0	B_i	0
0	0	1	0	1
0	1	0	B_i	0
0	1	1	B_i	0
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	0	1

Table 6: Truth Table of MUX input for Y_i

5 Block Diagram

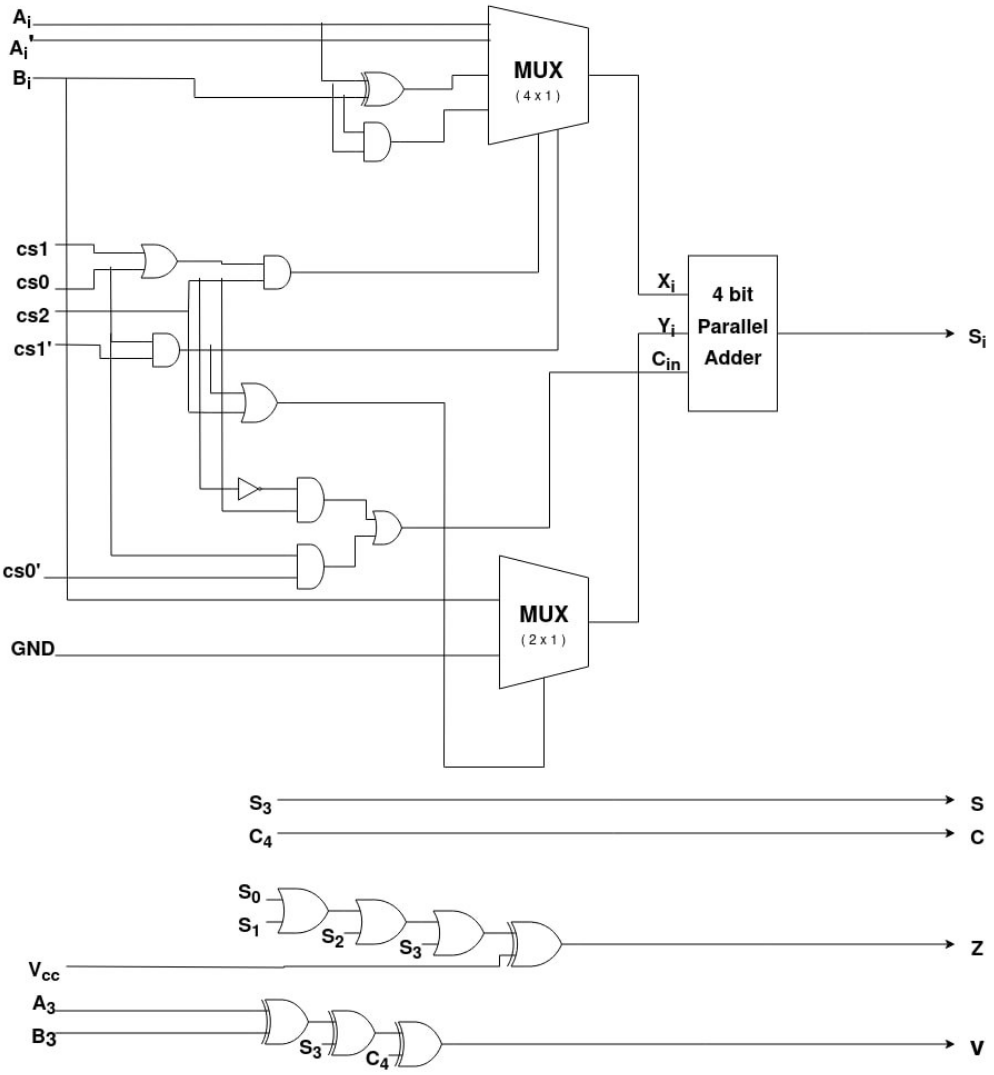


Figure 2: Block Digram of ALU

6 Complete Circuit Diagram

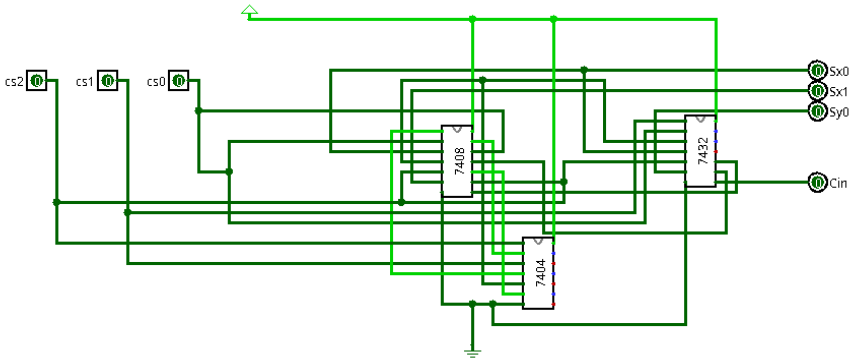


Figure 3: Control Processing Unit

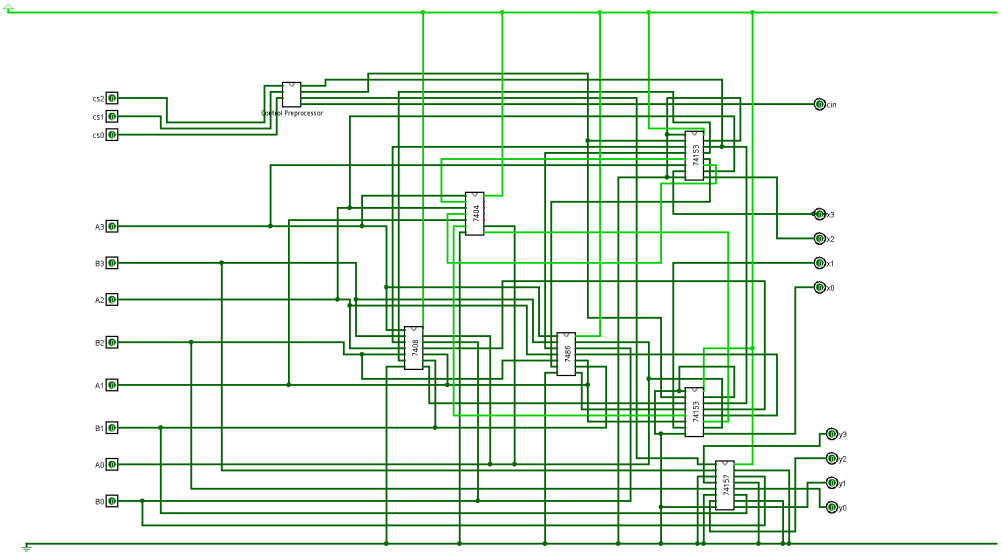


Figure 4: Input Processing Unit

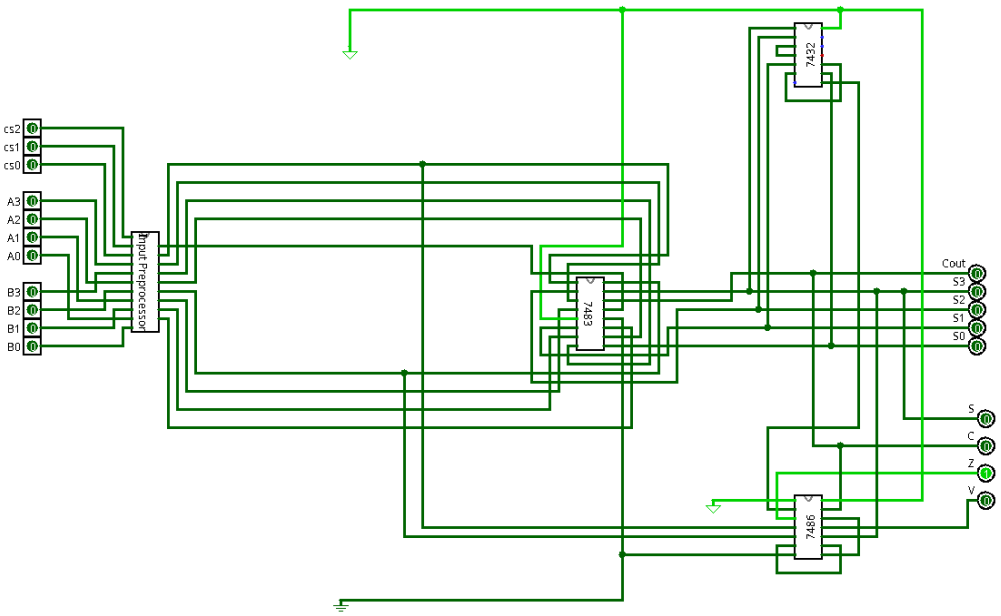


Figure 5: Arithmetic Logic Unit (ALU)

7 ICs Used with Count as a Chart

IC	Quantity
IC 74153	2
IC 74157	1
IC 7408	2
IC 7432	2
IC 7404	2
IC 7486	2
IC 7483	1
Total	12

Table 7: ICs Used with Quantity

8 The Simulator Used along with the Version Number

Logisim - 2.7.1

9 Discussion

The assignment on the implementation of ALU, our very first step onto the intricate realm of hardware design, assisted in strengthening and solidifying our understanding of the internal building block of computing in multifaceted dimensions. Alongside serving as a practical visualizer for our theoretical knowledge, the task helped refurbish our problem solving strategies and optimization techniques to a considerable extent.

The design was meticulously crafted with a primary focus on optimization, prompting subsequent refinement to minimize the count of integrated circuits (ICs). Our requirement for input variables X and Y, encompassing A, A' (complement of A), AB, A XOR B for X, and B, B' for Y, necessitated a thoughtful approach. While the multiplexer provided the expected X and Y, our commitment to utilizing basic gates and XOR gates compelled us to generate inputs for the MUX systematically.

In the input preprocessor, intermediate adder inputs were calculated and produced efficiently. MUX's were used to reduce IC counts. The attached truth tables depict relationships with various inputs with control bits. Expressions for flags were calculated with maximum accuracy.

4 position DIP switches were used to give inputs to the circuit. Their main advantages are that they are quicker to change and provide a visual indication of the position of the switch, making it easier for the user to understand the functioning of the circuit.

Rigorous testing and step-by-step debugging were integral to our circuit implementation process. During debugging, meticulous attention was devoted to ensuring all essential connections. Any potential incorrect connections were rectified. Additionally, vigilant checks were conducted to prevent inadvertent shorts in wires or components. Verifying the actual polarity of LEDs and other instruments was paramount. To enhance the debugging process, an extra LED was strategically connected, enabling us to validate the correctness and expected output at every stage of development.

Backtracking was necessary to debug several mis-produced outputs. Care was taken so that no two wires become short circuited. Ground and Vcc pins of IC's were connected correctly.

To obtain simplified expressions and to compare the corresponding IC counts over a bulk set of choices, a python script was used. The script scraped the output from a website and delivered that to another pipeline, thereby automating the process. Wires and resistors were reduced to small sizes using wire strippers to make the circuit cleaner. Black tape and double sided tapes were used to provide structural integrity. For extensive verification and testing, a python script was used to randomly generate test cases and their corresponding solutions to check if the circuit is working correctly.

To recapitulate, we exerted our fullest efforts onto a correct and optimized implementation. We are grateful to our supervisor, teachers and peers for their priceless assistance. We hope to explore this core concept of Computer Architecture in greater depth in future.

10 Contribution of Each Member

10.1 Logic Design and Simplification of Expressions

Student ID	Surname	Contribution
2005001	Anik	Prepared block diagram with IC pin numbers mentioned, obtained simplified expressions for Sx_1, Sx_0, Sy_0
2005012	Jahin	Prepared truth tables for Sx_1, Sx_0, Sy_0
2005013	Muhit	Prepared truth tables for X_i, Y_i, C_{in}
2005017	Amim	Prepared block diagram with IC pin numbers mentioned, obtained simplified expressions for X_i, Y_i, C_{in}
2005023	Jaber	Obtained primary expressions for the Flags Z(Zero Flag) and V(Overflow Flag)

10.2 IC Count Optimization

Student ID	Surname	Contribution
2005001	Anik	Performed optimal manipulations in preprocessing to reduce the number of IC 7408, replaced a NOT gate with XOR gate to lessen one extra IC 7404
2005012	Jahin	Developed synchronization among distinct modules to efficiently use maximum ports of ICs and to keep the modules disjoint
2005013	Muhit	Introduced the possibility of using MUX to avoid using too many gates in preprocessing, resolved errors related to optimization
2005017	Amim	Found a way of determining Overflow without explicitly obtaining the penultimate carry, thereby reducing one extra IC 7483, finalized the solution containing 12 ICs
2005023	Jaber	Wrote a Python bot to scrape SOP forms from the website 32x8.com for automated comparisons

10.3 Logisim Simulation

Student ID	Surname	Contribution
2005001	Anik	Developed Control Preprocessor which generates Sx_1, Sx_0, Sy_0, C_{in} from cs_2, cs_1, cs_0
2005012	Jahin	Developed the circuitry of the full adder (IC 7483) which generates S_3, S_2, S_1, S_0
2005013	Muhit	Developed Input preprocessor (jointly) which generates X_i, Y_i from A_i, B_i, Sx_1, Sx_0, Sy_0
2005017	Amim	Developed Input preprocessor (jointly) which generates X_i, Y_i from A_i, B_i, Sx_1, Sx_0, Sy_0
2005023	Jaber	Developed the circuitry of flags calculation which generates S, C, Z, V

10.4 Hardware Implementation

Student ID	Surname	Contribution
2005001	Anik	Developed Control Preprocessor which generates Sx_1, Sx_0, Sy_0, C_{in} from cs_2, cs_1, cs_0, assisted in developing the circuitry of Full Adder and flags calculation
2005012	Jahin	Developed a switching circuit for controlling the 11 input bits through three 4-bit DIP switches, also assisted in developing an output viewer circuit consisting of LEDs
2005013	Muhit	Developed Input preprocessor (jointly) which generates X_i, Y_i from A_i, B_i, Sx_1, Sx_0, Sy_0 and assisted in the circuitry of Full Adder and flags calculation
2005017	Amim	Developed Input preprocessor (jointly) which generates X_i, Y_i from A_i, B_i, Sx_1, Sx_0, Sy_0 and assisted in the circuitry of Full Adder and flags calculation
2005023	Jaber	Assisted in developing an output viewer circuit consisting of LEDs, finalized the hardware with improvements in structural integrity

10.5 Verification and Testing

Student ID	Surname	Contribution
2005001	Anik	Developed a python script for verifying the generated truth table in Logisim through comparison with given functions
2005012	Jahin	Refurnihsed the switching circuit for easy inputting mechanism, Performed random testing
2005013	Muhit	Performed extensive random testing and resolved bugs through backtracking
2005017	Amim	Performed crucial debugging while performing the verification process, performed random testing
2005023	Jaber	Developed a python script for randomly generating testcases and their corresponding solutions based on given control bits

10.6 Report Preparation

Student ID	Surname	Contribution
2005001	Anik	Section A, B, J
2005012	Jahin	Section E, F
2005013	Muhit	Section D
2005017	Amim	Section C
2005023	Jaber	Section G, H