Project 1: 2-bit Arithmetic Logic Unit

Aidan Wong

ECE-150 - Digital Logic Design

October 9, 2025

1 Introduction

This project involves designing and implementing a 2-bit Arithmetic Logic Unit (ALU) as a combinatorial logic circuit using gates and multiplexers (muxes). It accepts two 2-bit words (A and B) and a 4-bit operation code (Op) to output a single 2-bit word (Y) with an optional carry-out (Cout), as shown in Figure 1.

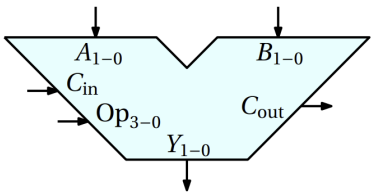

Figure 1: Block Diagram of 2-Bit ALU with inputs and outputs

1.1 Background

The Arithmetic Logic Unit (ALU) is a fundamental component of the Central Processing Unit (CPU) that handles all arithmetic and logical operations in a digital system. It receives binary input, executes the selected operation based on control signals, and outputs the result with optional status flags (e.g. carry, overflow, etc.). These outputs are then used by other parts of the processor to make decisions or perform further computations.

1.2 Technical Specifications

This ALU performs eight operations (as shown in Table 1), representing inputs using active-high DIP switches and outputs using active-low LEDs. Binary subtraction is unique among these operations in that it interprets its inputs and outputs in two's complement signed integers.

Operation Code	Operation	Expression
0000	No-op	Y = A
0001	Binary Addition	Y = A + B
0010	Binary Subtraction	Y = B - A
0011	Logical Shift of A	A = 01, Y = 1Cin
0100	Bitwise OR	Y = A + B
0101	Bitwise AND	Y = AB
0110	Bitwise NOT	Y = !A
0111	Bitwise XOR	Y = $A \oplus B$

Table 1: ALU operation results

1.3 Report Structure

This report will cover the ALU design process through three main sections: *Methods*, *Implementation*, and *Discussion and Conclusions*. The *Methods* section explains the technical requirements and logic behind each operation, including truth tables, Karnaugh maps (K-maps), and simplified expressions. The *Implementation* section explains how the circuit was constructed physically and the choices made to optimize the number of chips and the wire paths. It will also display the final breadboard with labeled IC chips and sub-circuits. The final *Discussion and Conclusions* section is a reflection of the final product, detailing future scalability, engineering and design choices, strengths and flaws, and an overview of the ALU's implemented functionality.

---

2 Methods

This section details the logical design process for each operation of the 2-bit ALU. Each operation was first expressed as a truth table and then simplified using a combination of K-maps, inspection, and Boolean algebra.

2.1 Pass-Through/No-OP (OP-0000)

Pass-Through/NO-OP is an operation that leaves the input unchanged, simply passing it to the output. In this implementation, the operation simply outputs the value of A and is unaffected by the state of B, as shown in Figure 2. This behavior is shown in the truth table and K-maps below (Figures 3 and 4 respectively)

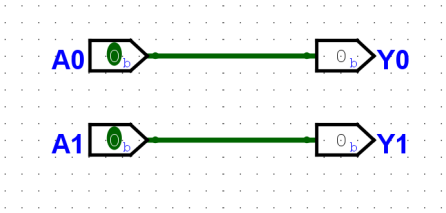

Figure 2: Circuit Diagram of Pass-Through operation in Logisim

2.1.1 Truth Table

A_1	A_0	Y_1	Y_0
0	0	0	0
0	1	0	1
1	0	1	0
1	1	1	1

Figure 3: Truth Table for Pass-Through Operation

2.1.2 K-Maps

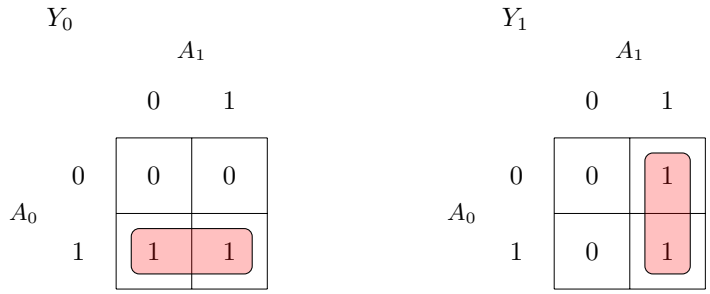

Figure 4: K-Maps for Pass-Through Operation

2.1.3 Simplified Expressions

Since the K-Maps for each output only have one group, the simplified Boolean expressions are directly given by those groups. This could have also been derived through inspection.

$Y_0 = A_0$

$Y_1 = A_1$

2.2 Binary Addition (OP-0001)

Binary addition adds two 2-bit unsigned integers ($A + B$) and outputs a two bit unsigned integer sum (Y). Any overflow (sum that exceeds the range representable by two bits) is handled by the carry-out (C_{out}), which serves as the most significant bit (MSB) of the full 3-bit sum.

It was created by first implementing a 1-bit full adder, which takes two single-bit inputs (A_0 and B_0) and a carry-in (C_{in}), producing a sum (Y_0) and a carry-out (C_{out}), as shown in Figure 5. Then, two of these one-bit adders were chained together by connecting the C_{out} of the least significant bit (LSB) adder to the most significant bit (MSB) adder, forming a 2-bit adder, as shown in Figure 6. This behavior is shown in the 1-bit truth table, 2-bit truth table and K-maps below (Figures 7, 8, and 9 respectively).

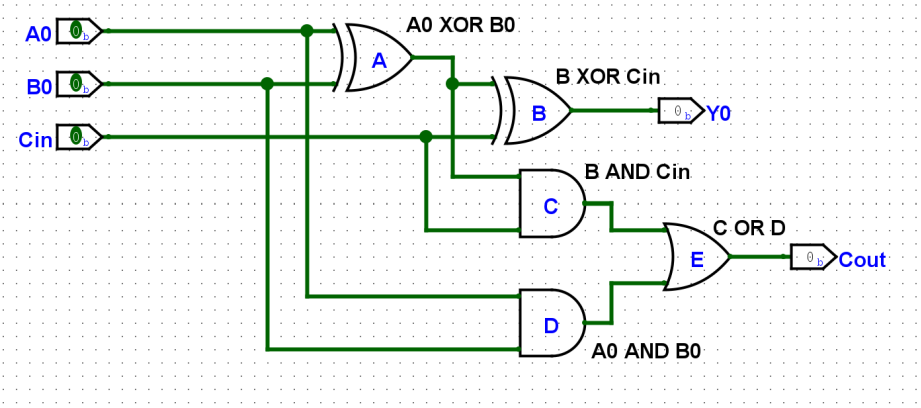

Figure 5: Circuit Diagram of 1-Bit Binary Addition in Logisim

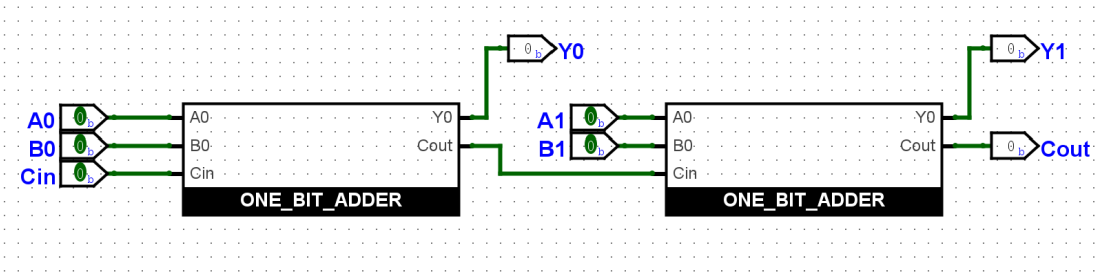

Figure 6: Circuit Diagram of 2-Bit Binary Addition in Logisim

Note: The One Bit Adder is black boxed for clarity.

2.2.1 Truth Tables

For single bit adder:

C_{in}	A_0	B_0	C_{out}	Y_0
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

Figure 7: Truth Table for 1-bit addition

Note: The inputs C_{in}, A_0, and B_0 are all binary and represent the values 0 or 1. The C_{out} is the MSB of the 2-bit output.

For 2-bit adder:

C_{in}	A_1	A_0	B_1	B_0	C_{out}	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	1
0	0	0	1	0	0	1	0
0	0	0	1	1	0	1	1
0	0	1	0	0	0	0	1
0	0	1	0	1	0	1	0
0	0	1	1	0	0	1	1
0	0	1	1	1	1	0	0
0	1	0	0	0	0	1	0
0	1	0	0	1	0	1	1
0	1	0	1	0	1	0	0
0	1	0	1	1	1	0	1
0	1	1	0	0	0	1	1
0	1	1	0	1	1	0	0
0	1	1	1	0	1	0	1
0	1	1	1	1	1	1	0
1	0	0	0	0	0	0	1
1	0	0	0	1	0	1	0
1	0	0	1	0	0	1	1
1	0	0	1	1	1	0	0
1	0	1	0	0	0	1	0
1	0	1	0	1	0	1	1
1	0	1	1	0	1	0	0
1	0	1	1	1	1	0	1
1	1	0	0	0	0	1	1
1	1	0	0	1	1	0	0
1	1	0	1	0	1	0	1
1	1	0	1	1	1	1	0
1	1	1	0	0	1	0	0
1	1	1	0	1	1	0	1
1	1	1	1	0	1	1	0
1	1	1	1	1	1	1	1

Figure 8: Truth Table for 2-bit addition

Note: Similarly to the 1-bit adder, each input is binary. C_{out} is the MSB of the full 3-bit output, while Y_0 is the LSB.

2.2.2 K-Maps

Only the K-maps for single-bit addition are shown because 2-bit addition was implemented by combining 2 single-bit adders (thus the expressions and logic are equivalent for each bit).

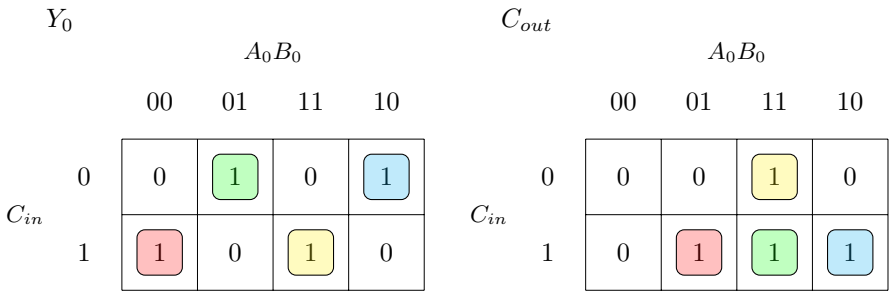

Figure 9: K-Maps for Addition

Note: Although groups of two could have been formed on the K-map for C_{out} (as shown in Figure 10) to create a simpler expression, grouping by one allowed the reuse of more gates from the sum (S_0) expression.

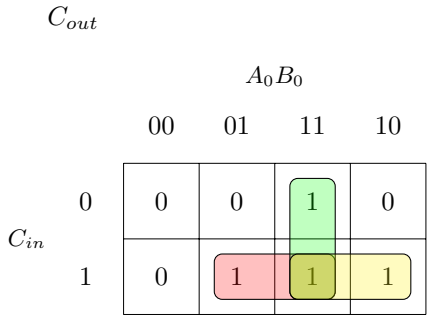

Figure 10: K-Maps for Alternative C_{out}

2.2.3 Simplified Expressions

$$
\begin{aligned}
Y_0 &= \overline{C_{in}}(A_0 \oplus B_0) + C_{in}(\overline{A_0 \oplus B_0}) \\
&= C_{in} \oplus A_0 \oplus B_0
\end{aligned}
$$

$$
\begin{aligned}
C_{in} &= C_{in}\overline{A_0}B_0 + C_{in}A_0B_0 + \overline{C_{in}}A_0B_0 + C_{in}A_0\overline{B_0} \\
&= A_0B_0(C_{in} + \overline{C_{in}}) + C_{in}(\overline{A}B + A\overline{B}) \\
&= A_0B_0 + C_{in}(A \oplus B)
\end{aligned}
$$

Alternative C_{in} (with groups of 2)

$$
\begin{aligned}
C_{in} &= A_0B_0 + C_{in}A_0 + C_{in}B_0 \\
&= B_0(A_0 + C_{in}) + C_{in}A_0
\end{aligned}
$$

2.3 Binary Subtraction (OP-0010)

Binary subtraction subtracts two 2-bit 2's complement signed integers ($B - A$) and outputs a 2-bit 2's complement signed integer difference (Y). Overflow is not handled in this implementation, meaning any case with a difference that exceeds the range representable by two-bit 2's complement is thrown out. However, these out-of-range integers can be represented when extending the circuit by connecting the carry-out (C_{out}) to the carry-in (C_{in}) of additional stages, which is further discussed in the Scalability section of this report.

To perform $B - A$, this circuit flips the bits of A and adds one, which is an expansion of the intermediary steps of the previously built addition circuit. Similarly to addition, a one-bit full subtractor was created first, taking two single-bit inputs (A_0 and B_0) and a constant carry-in (C_{in}) of 1 to implement the 'add one' step, as shown in Figure 11. This creates a difference (Y_0) and carry-out (C_{out}). Finally, two of these 1-bit subtractor were chained together by connecting the C_{out} of the least significant bit (LSB) subtractor to the most significant bit (LSB) subtractor to form a 2-bit subtractor, as shown in Figure 12. This is shown in the 1-bit truth table, 2-bit truth table, and K-maps below (Figures 13, 14, and 15 respectively).

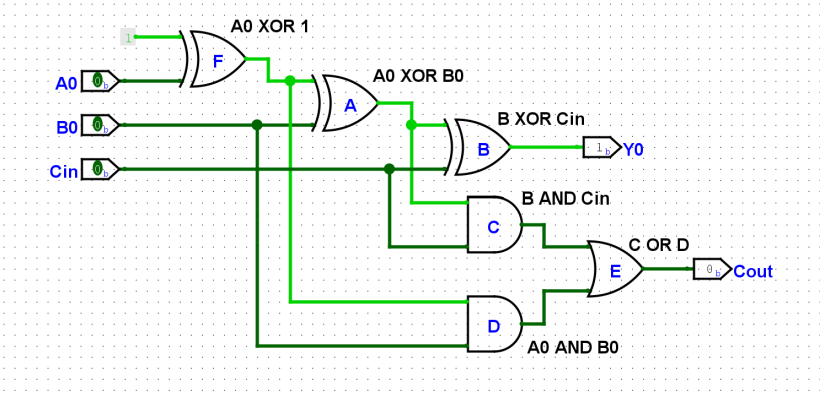

Figure 11: Circuit Diagram of 1-Bit Subtraction in Logisim

Note: An XOR with one input as a one functions as an inverter. This was used to reduce the total number of chips.

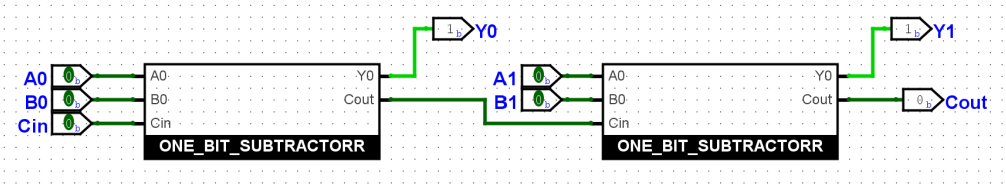

Figure 12: Circuit Diagram of 2-Bit Subtraction in Logisim

Note: The One Bit Subtractor is black boxed for clarity.

2.3.1 Truth Table

For single bit subtractor

C_{in}	A_0	B_0	C_{out}	Y_0
0	0	0	0	1
0	0	1	1	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	0
1	0	1	1	1
1	1	0	0	1
1	1	1	1	0

Figure 13: Truth Table for 1-Bit Subtraction

Note: When the input C_{in} is 0, the truth table is simply representing the borrow propagation logic and the inputs A_0 and B_0 have no standalone meaning. When C_{in} is 1, A_0 and B_0 are both -1 (as they represent the MSB in the 2's complement number A and B respectively). The range of 1-bit is extremely limited, and $C_{in} = 1$, $A_0 = 1$, and $B_0 = 0$ is the only one properly displayed in 1-bit 2's complement ($-1 - 0 = -1$).

For 2-bit subtractor

C_{in}	A_1	A_0	B_1	B_0	C_{out}	Y_1	Y_0
0	0	0	0	0	0	1	1
0	0	0	0	1	1	0	0
0	0	0	1	0	1	0	1
0	0	0	1	1	1	1	0
0	0	1	0	0	0	1	0
0	0	1	0	1	0	1	1
0	0	1	1	0	1	0	0
0	0	1	1	1	1	0	1
0	1	0	0	0	0	0	1
0	1	0	0	1	0	1	0
0	1	0	1	0	0	1	1
0	1	0	1	1	1	0	0
0	1	1	0	0	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	0	0	1	0
0	1	1	1	1	0	1	1
1	0	0	0	0	1	0	0
1	0	0	0	1	1	0	1
1	0	0	1	0	1	1	0
1	0	0	1	1	1	1	1
1	0	1	0	0	0	1	1
1	0	1	0	1	1	0	0
1	0	1	1	0	1	0	1
1	0	1	1	1	1	1	0
1	1	0	0	0	0	1	0
1	1	0	0	1	0	1	1
1	1	0	1	0	1	0	0
1	1	0	1	1	1	0	1
1	1	1	0	0	0	0	1
1	1	1	0	1	0	1	0
1	1	1	1	0	0	1	1
1	1	1	1	1	1	0	0

Figure 14: Truth Table for 2-Bit Subtraction

Note: This truth table follows the same logic as the 1-bit subtractor besides the fact that it has a wider range of displayable outputs. When calculating, only the MSB can represent -1, while the other on bits represent 1.

2.3.2 K-Maps

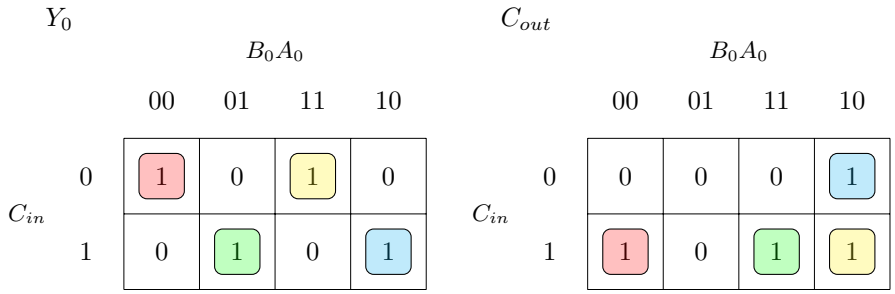

Figure 15: K-Maps for Subtraction

Note: Although groups of two could have been formed on the K-map for C_{out} (as shown in Figure 16) to create a simpler expression, grouping by one allowed the reuse of more gates from the sum (S_0) expression.

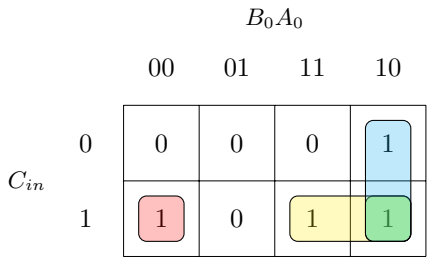

Figure 16: K-Maps for Alternative C_{out}

Note: The Red and Green boxes make up one group

2.3.3 Simplified Expressions

$$
\begin{aligned}
Y_0 &= \overline{C_{in}A_0B_0} + \overline{C_{in}}A_0B_0 + C_{in}A\overline{B_0} + C_{in}\overline{A_0}B \\
&= C_{in}(A_0\overline{B_0} + \overline{A_0}B_0) + \overline{C_{in}}(\overline{A_0B_0} + A_0B_0) \\
&= \overline{C_{in}(A_0 \oplus B_0)} + \overline{C_{in}}(\overline{A_0 \oplus B_0}) \\
&= \overline{C_{in} \oplus (A \oplus B)} \\
C_{out} &= C_{in}\overline{B_0A_0} + C_{in}A_0B_0 + C_{in}B_0\overline{A_0} + \overline{C_{in}}B)\overline{A_0} \\
&= B\overline{A_0} + C_{in}(\overline{A_0 \oplus B_0})
\end{aligned}
$$

Alternative C_{in} (with groups of 2)

$C_{out} = C_{in}\overline{A_0} + C_{in}B_0 + B_0\overline{A_0}$

Note: The Y_0, $B_0\overline{A_0}$, and $\overline{A_0 \oplus B_0}$ are inverses from the corresponding terms in addition (Helpful in creating combined circuit discussed in Implementation)

2.4 Bitwise Logical Shift (OP-0011)

Bitwise Logical Shift is an operation that shifts individual bits to the left or right (increasing or decreasing its significance), inserting a specified value into the vacated positions. In this implementation, the operation performs a left shift on the input word A, moving each bit one to the left, as shown in Figure 17. The LSB is filled with the carry-in value (C_{in}), while the MSB is discarded. This behavior is shown in the Truth Table and K-Map below (Figures 18 and 19 respectively).

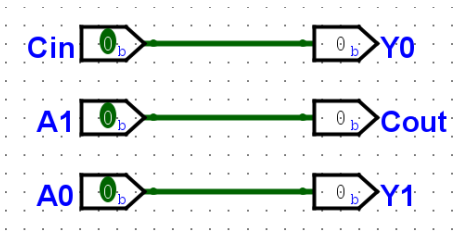

Figure 17: Circuit Diagram of Bitwise Logical Shift in Logisim

2.4.1 Truth Table

C_{in}	A_1	A_0	C_{out}	Y_1	Y_0
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	1	1

Figure 18: Truth Table for Logical Shift

Note: For the inputs, C_{in} is the MSB and A_0 is the LSB. For the outputs, C_{out} is the MSB and Y_0 is the LSB.

2.4.2 K-Map

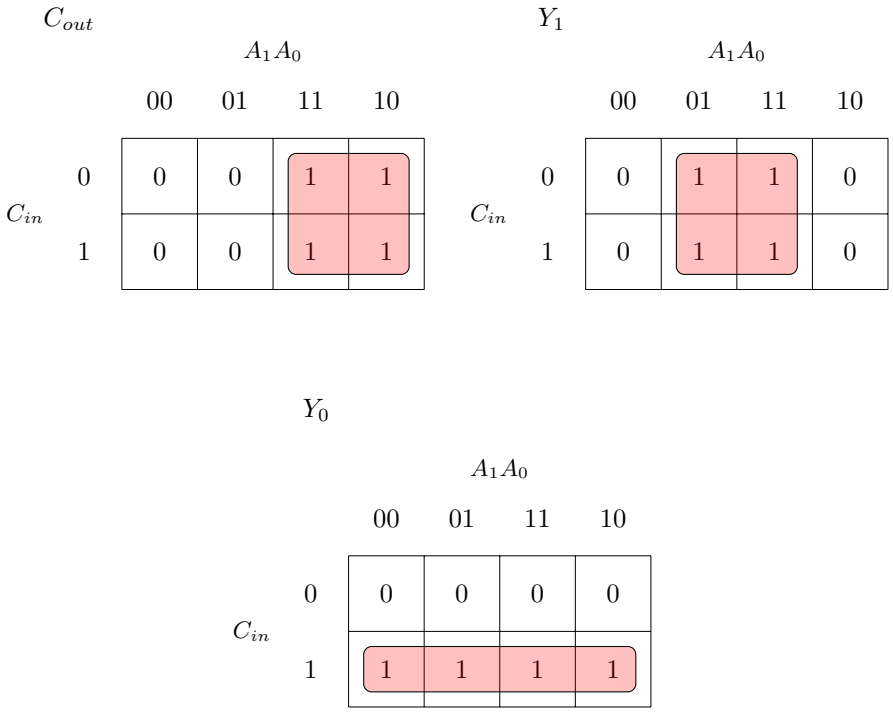

Figure 19: K-Maps for Logical Shift

2.4.3 Simplified Expressions

Since the K-Maps for each output only have one group, the simplified Boolean expressions are directly given by those groups. This could have also been derived through inspection.

$C_{out} = A_1$
$Y_1 = A_0$
$Y_0 = C_{in}$

2.5 Bitwise OR (OP-0100)

Bitwise OR is an operation that compares two binary inputs bit by bit and outputs a 1 (digital high) in the positions where at least one of the corresponding inputs is 1. A digital low (0) only occurs when both inputs are 0. In this implementation , each bit of A is compared to the corresponding input of B (A_n compared to B_n), and displays the logical OR for each bit position, as shown in Figure 20. This logic is also displayed by the Truth Table and K-Map below (Figures 21 and 22 respectively).

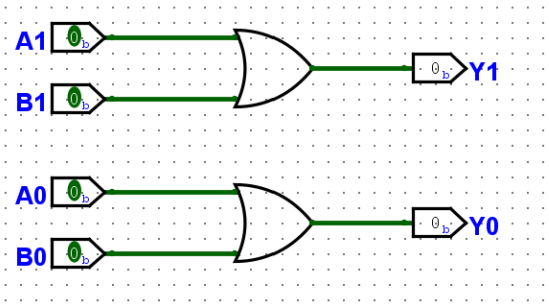

Figure 20: Circuit Diagram of Bitwise OR in Logisim

2.5.1 Truth Table

A_1	A_0	B_1	B_0	Y_1	Y_0
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	1	1
0	1	1	1	1	1
1	0	0	0	1	0
1	0	0	1	1	1
1	0	1	0	1	0
1	0	1	1	1	1
1	1	0	0	1	1
1	1	0	1	1	1
1	1	1	0	1	1
1	1	1	1	1	1

Figure 21: Truth Table for Bitwise OR

2.5.2 K-Map

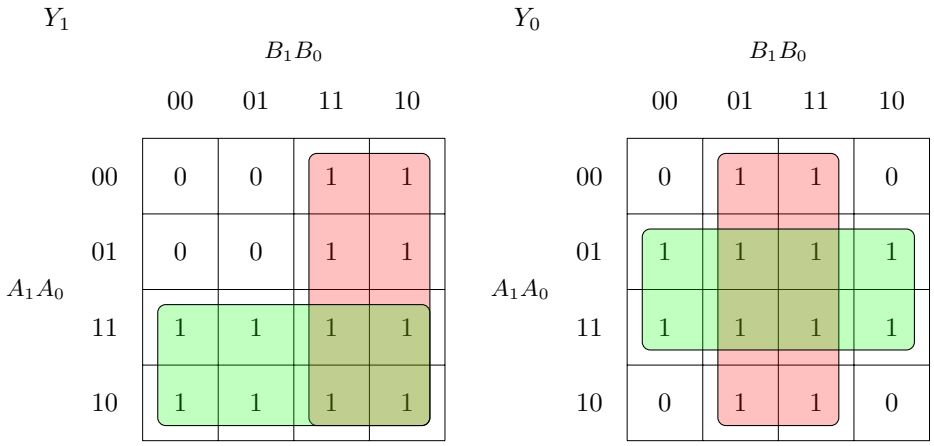

Figure 22: K-Maps for Bitwise OR

2.5.3 Simplified Expressions

Since the K-Maps for each output only have one group, the simplified Boolean expressions are directly given by those groups. This could have also been derived through inspection.
$Y_1 = A_1 + B_1$
$Y_0 = A_0 + B_0$

2.6 Bitwise AND (OP-0101)

Bitwise AND is an operation that compares two binary inputs bit by bit and outputs a 1 (digital high) in the positions where both corresponding inputs are 1. A digital low (0) occurs whenever at least one of the inputs are 0. In this implementation, each bit of A is compared to the corresponding input of B (A_n compared to B_n), and displays the logical AND for each position, as shown in Figure 23. The Truth Table and K-Map below also display this behavior (Figures 24 and 25 respectively).

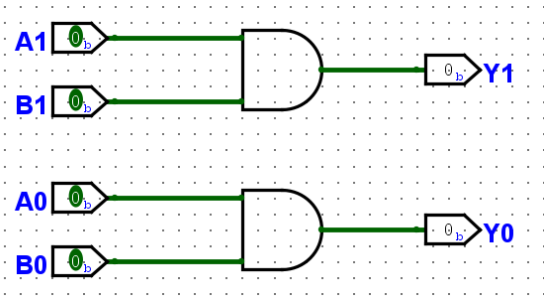

Figure 23: Circuit Diagram of Bitwise AND in Logisim

2.6.1 Truth Table

A_1	A_0	B_1	B_0	Y_1	Y_0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	0	0
0	1	0	1	0	1
0	1	1	0	0	0
0	1	1	1	0	1
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	1	0
1	0	1	1	1	0
1	1	0	0	0	0
1	1	0	1	0	1
1	1	1	0	1	0
1	1	1	1	1	1

Figure 24: Truth Table for Bitwise AND

2.6.2 K-Map

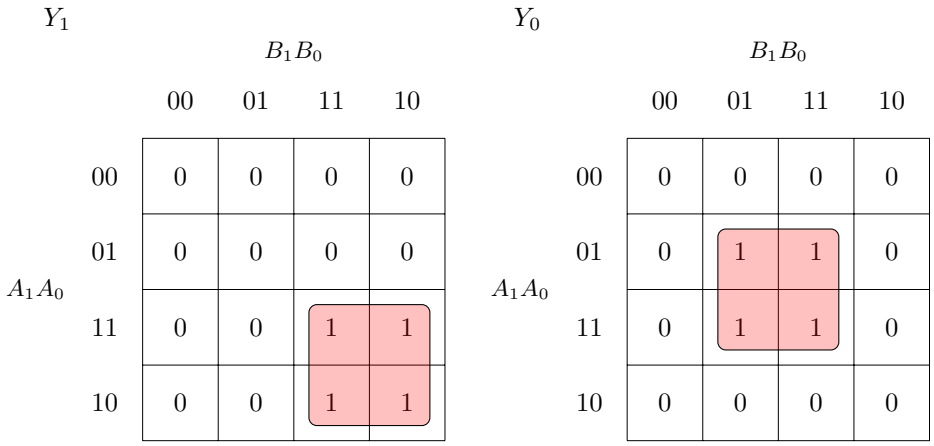

Figure 25: K-Maps for Bitwise AND

2.6.3 Simplified Expressions

Since the K-Maps for each output only have one group, the simplified Boolean expressions are directly given by those groups. This could have also been derived through inspection.

$Y_1 = A_1B_1$
$Y_0 = A_0B_0$

2.7 Bitwise NOT (OP-0110)

Bitwise NOT is an operation that inverts each binary input, converting every digital high (1) to a digital low (0) and vice versa. In this implementation, the operation is performed on A and the input B has no impact on the output, as shown in Figure 26. This behavior is also displayed in the Truth Table and K-Map below (Figures 27 and 28 respectively).

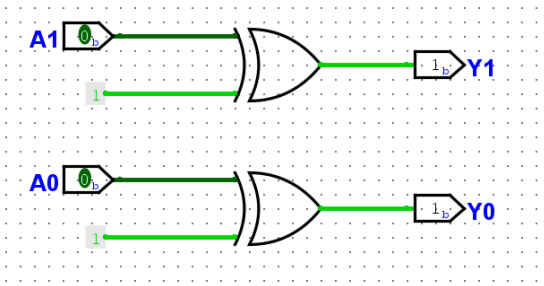

Figure 26: Circuit Diagram of Bitwise NOT in Logisim

Note: An XOR with one input as a one functions as an inverter. This was used to reduce the total number of chips.

2.7.1 Truth Table

A_1	A_0	Y_1	Y_0
0	0	1	1
0	1	1	0
1	0	0	1
1	1	0	0

Figure 27: Truth Table for Bitwise NOT

2.7.2 K-Map

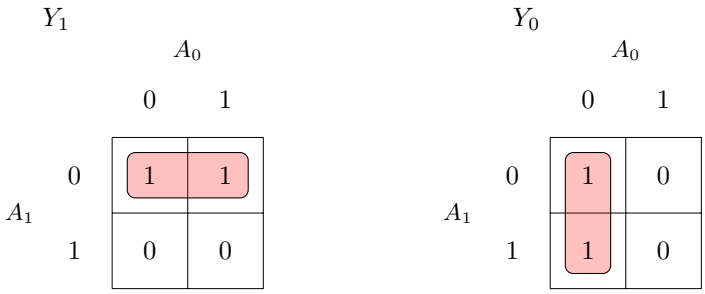

Figure 28: K-Maps for Bitwise NOT

2.7.3 Simplified Expressions

Since the K-Maps for each output only have one group, the simplified Boolean expressions are directly given by those groups. This could have also been derived through inspection.

$Y_0 = \overline{A_0}$
$Y_1 = \overline{A_1}$

2.8 Bitwise XOR (OP-0111)

Bitwise XOR is an operation that compares two binary inputs bit by bit and outputs a 1 (digital high) in the positions where the corresponding inputs are different. In this implementation, each bit of A is compared to the corresponding input of B (A_n compared to B_n), displaying a digital high when one input is high and the other is low, as shown in Figure 29. The Truth Table and K-Maps below also display this behavior (as shown in Figures 30 and 31 respectively).

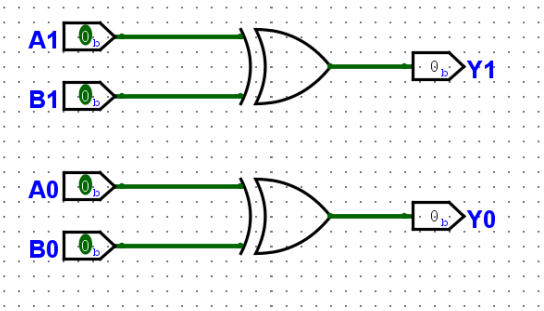

Figure 29: Circuit Diagram of Bitwise XOR in Logisim

2.8.1 Truth Table

A_1	A_0	B_1	B_0	Y_1	Y_0
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	1	0	0
0	1	1	0	1	1
0	1	1	1	1	0
1	0	0	0	1	0
1	0	0	1	1	1
1	0	1	0	0	0
1	0	1	1	0	1
1	1	0	0	1	1
1	1	0	1	1	0
1	1	1	0	0	1
1	1	1	1	0	0

Figure 30: Truth Table for Bitwise XOR

2.8.2 K-Map

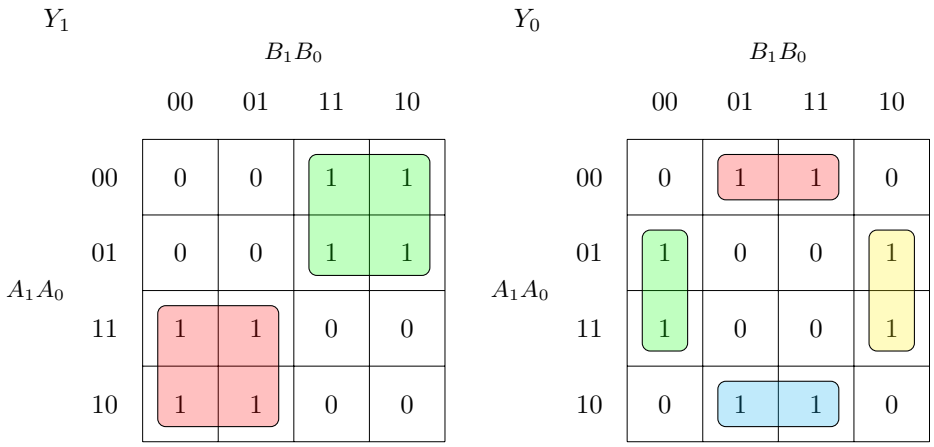

Figure 31: K-Maps for Bitwise XOR

2.8.3 Simplified Expressions

Since the K-Maps for each output only have one group, the simplified Boolean expressions are directly given by those groups. This could have also been derived through inspection.

$Y_0 = A_0 \oplus B_0$
$Y_1 = A_1 \oplus B_0$

---

3 Implementation

This implementation is built around a 2-bit adder-subtractor, where the circuit logic at various intermediary steps is reused to compute the other operations in parallel. A multiplexer then selects the appropriate output based on the op code, ensuring constant-time execution regardless of the operation.

3.0.1 Planning

To optimize the number of chips used, the design process began by creating a simulation of the circuit in Logisim (shown in Figure 32), followed by grouping the simplified expressions found in *Methods* and identifying common logic that could be reused.

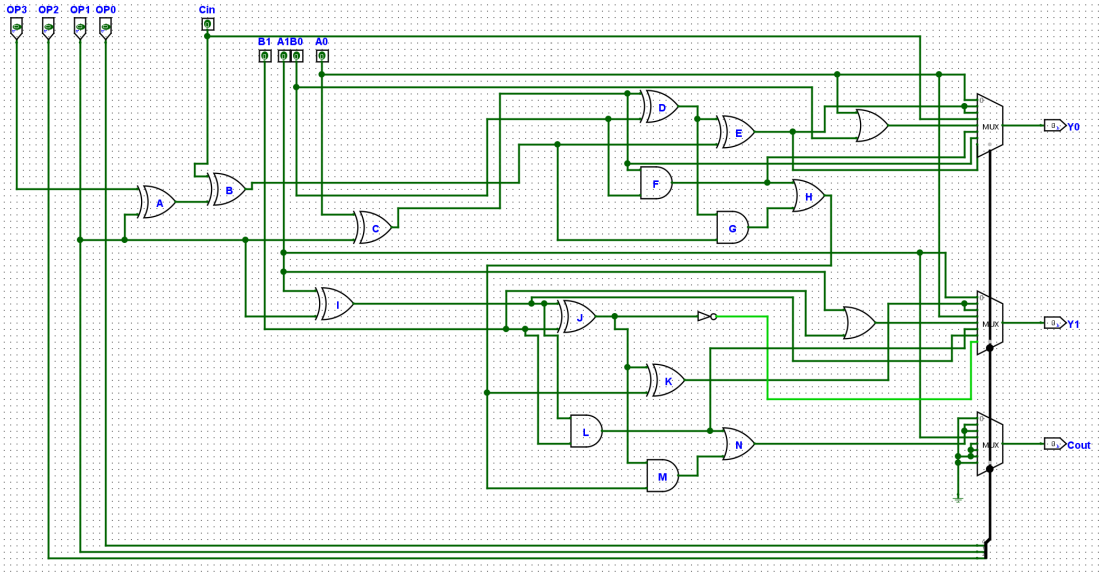

Figure 32: Circuit Planning Done in Logisim

The resulting expressions for each operation are summarized below (Table 2), and also serve as a key to the mapping diagram (Figure 33).

Code	Operation	Process
A	$Op_1 \oplus Op_1$	Intermediary
B	$A \oplus C_{in}$	Intermediary
C	$A_0 \oplus Op_1$	Output + Intermediary
D	$B_0 \oplus C$	Intermediary
E	$B \oplus D$	Output
F	B_0C	Output + Intermediary
G	BD	Intermediary
H	$F + G$	Intermediary
I	$A_1 \oplus Op_1$	Output + Intermediary
J	$B_1 \oplus I$	Output + Intermediary
K	$H \oplus J$	Output
L	IB_1	Output + Intermediary
M	JH	Intermediary
N	$M + L$	Output
O	$A_0 + B_0$	Output
P	$A_1 + B_1$	Output

Table 2: Optimized Expressions for building ALU

Note: OP_n represents a bit of the operation code
Note 2: Outputs derived by directly feeding an input to the MUX are not shown

To efficiently map out the circuit to eliminate/reduce the amount of "ugly" wires (crossed, squeezed, etc), the breadboards were drawn out in Excel and wired using an arbitrary color scheme, as shown in Figure 33.

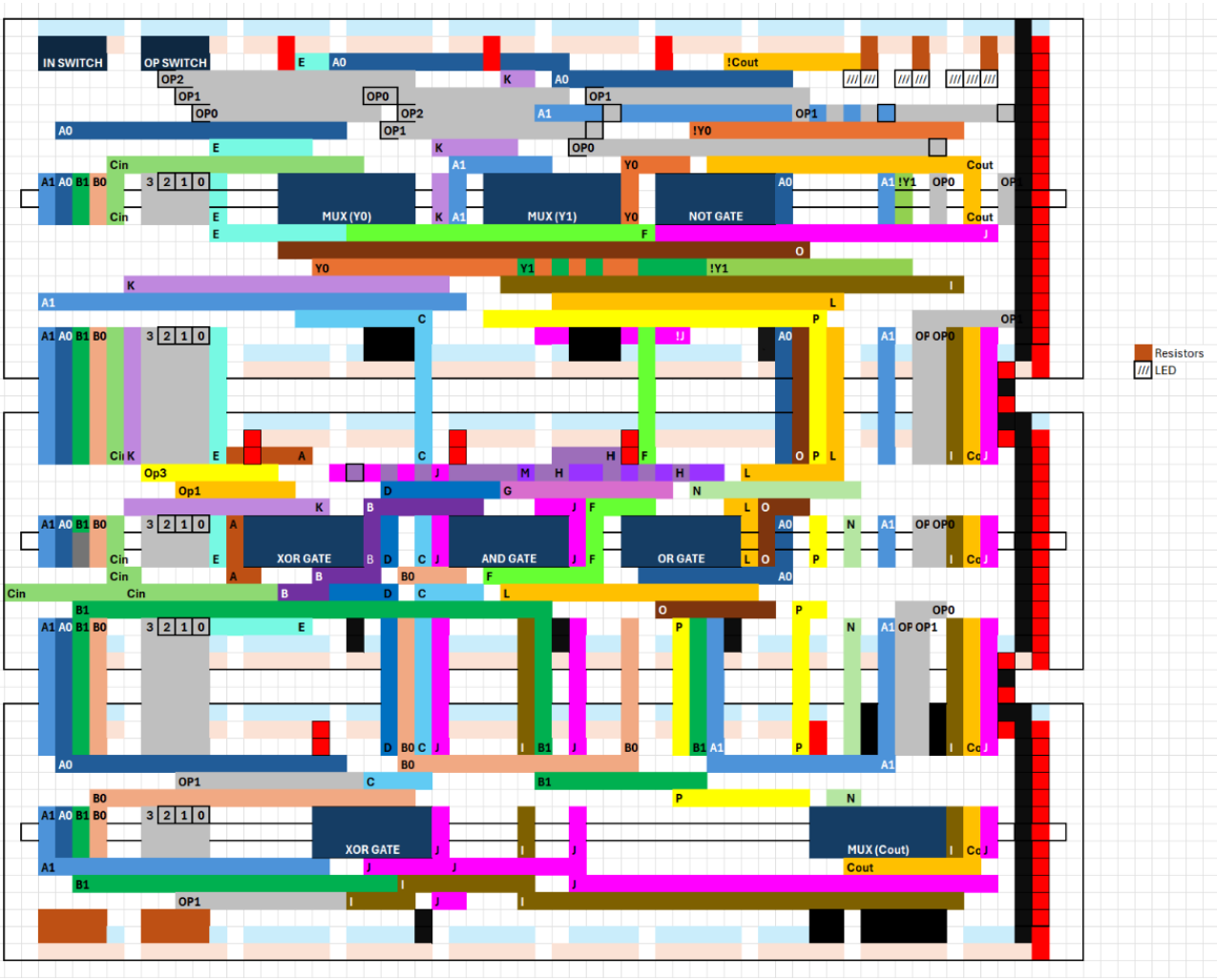

Figure 33: Breadboard Planning Done in Excel

Note: A key is included for key components and the start of each wire is denoted by the appropriate letter of the operation it represents.

Note 2: The first breadboard has double the amount of rows to indicate where and how wires will be squeezed. The wires with a spotted pattern are also squeezed, but not represented with an additional row (due to their isolated existence on their respective breadboard).

3.0.2 Physical Build

The model was then used to build the physical implementation of the ALU on a breadboard, as shown in Figure 34. For visual clarity, a color-coding scheme based on the expressions shown in Table 2 was created (represented in Table 3).

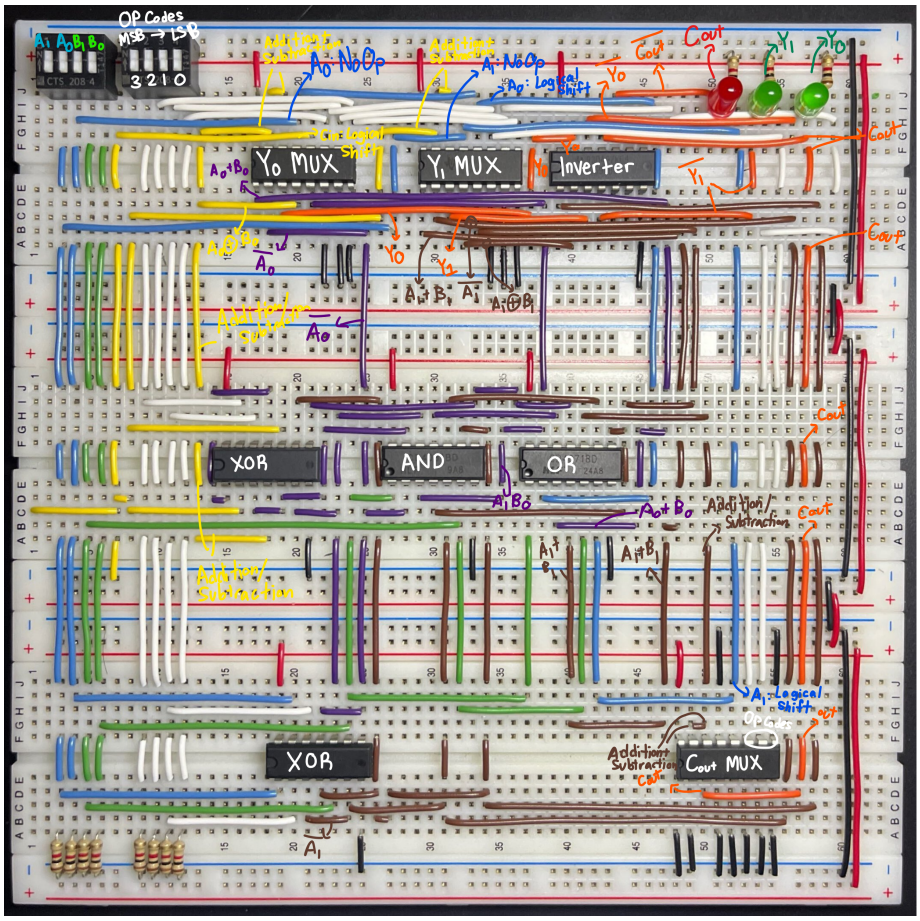

Figure 34: Breadboard Implementation with Labeled Sub-circuits

Color	Representation
Red	Power Wires
Black	Ground Wires
White	Operation Codes
Purple	A-H + O
Brown	I-N + P
Yellow	K and E
Orange	Output
Blue	A (A_1A_0)
Green	B (B_1B_0)

Table 3: Color Coding Scheme for Breadboard

3.0.3 Optimizations

This section will briefly discuss significant optimizations made to reduce the total number of logical gates (omits the simple reuse of existing gates).

1. **Binary Addition and Subtraction**: The operation code bit Op_1 was used as a signal whether or not to invert the A inputs. Conveniently, OP_1 is set to 1 only for Subtraction and Bitwise NOT (excluding XOR and Logical Shift since they do not use this part of the circuit), which allowed the A inputs to pass through normally in addition mode (since $OP_1 = 0$ in the operation code for addition). This created a circuit that could change between addition and subtraction based on the OP code, reducing the amount of gates by a substantial amount.

2. **Bitwise NOT**: This operation was implemented by using the same OP_1 signal that was used for Binary Addition and Subtraction to invert the A bits using an XOR.

3. **Bitwise XOR**: The XOR operation was derived by identifying gate configurations in the addition and subtraction circuit that exhibited XOR-like behavior. For Y_0, this was the output of the XOR gate labeled E in Figure 32, while for Y_1, it was derived by taking the inverse of the XOR labeled J in Figure 32.

---

4 Discussion and Conclusion

A 2-bit ALU was designed and successfully implemented on a breadboard, including active-high dip switches and active-low LED outputs. It takes in 4 arguments as inputs ($A : A_1A_0$, $B : B_1B_0$, $OP : OP_3OP_2OP_1OP_0$, and C_{in}) and outputs 2 arguments ($Y : Y_1Y_0$ and C_{out}) according to the operation table, which defines the function for each operation code. The design allows multiple operations to be computed in parallel, with a multiplexer selecting outputs based on the op code.

Operation Code	Operation	Expression
0000	No-op	Y = A
0001	Binary Addition	Y = A + B
0010	Binary Subtraction	Y = B - A
0011	Logical Shift of A	A = 01, Y = 1Cin
0100	Bitwise OR	Y = A + B
0101	Bitwise AND	Y = AB
0110	Bitwise NOT	Y = !A
0111	Bitwise XOR	Y = $A \oplus B$

Table 4: ALU operation results (Copy of Table 1 in Technical Specifications)

4.1 Scalability

This circuit can be scaled to a 2N-bit ALU by chaining the C_{out} output of each lower-order operation to the carry-in (C_{in}) of the next higher-order operation. It is important to note that the C_{out} of the original circuit (before expansion) becomes an internal component when chained and does not represent a part of the final output anymore for all relevant operations. Furthermore, only operations that rely on carry propagation, such as binary addition, binary subtraction, and logical shift, can be scaled in this way. Bitwise operations like AND, NOT, and XOR operate independently on each bit and do not require chaining/the use of a C_{in}.

4.2 Engineering Choices

- **Usage of 2 Dip Switches:** This implementation only uses 2 Dip Switches (1 for the Op code and 1 for the inputs A and B) instead of the 3 specified. This was because an additional dip switch would increase the complexity of wiring by taking up extra room. It was also deemed extraneous component-wise because it was not used for anything besides representing inputs (which could be done with a single dip switch).

- **OP Code Signals:** A 3-bit OP code was shown to the MUXes instead of a 4-bit one because the additional bit does not change the functionality of the ALU or introduce new operations. This additional bit of the OP code was instead used to signal whether the ALU should take in a C_{in} from a previous stage or use a constant logic value. This was especially important for enabling scalability in 2's complement subtraction. This was taken further with the C_{out} MUX, feeding it only a 2-bit OP code because none of the operations it was involved the third bit of the OP code.
- **Active-Low LEDs:** Supplies the LED with a constant voltage source, which makes the voltage provided more consistent. This minimizes potential errors caused by voltage fluctuations or noise.

4.3 Strengths and Weaknesses

Strengths

- **Organized Wiring:** This circuit was wired with minimal squeezing, no crossed wires, and color-coded connections, making it easy to trace signals and debug.
- **Active-Low LEDs:** Supplies the LED with a constant voltage source, which makes the voltage provided more consistent. This minimizes potential errors caused by voltage fluctuations or noise.
- **Efficient Logic Design:** The amount of chips was minimal, if not the lowest possible for an active-low configuration.

Weaknesses

- **Incomplete OP Code Handling:** The exclusion of the third bit of the OP Code caused the C_{out} MUX to interpret OP codes for Bitwise gates as pass through, addition, subtraction, and Bitwise shift. This caused the C_{out} LED to calculate for those values, but it is not a major issue because the carry-out for Bitwise operations is ignored in normal use of these operations.
- **No Overflow Detection for Subtraction:** If the result of the subtraction of two numbers exceeds the range representable by 2 bit 2's complement, the circuit cannot indicate that overflow has occurred. This leads to confusing outputs for those cases, but can be resolved by scaling the ALU to handle more bits.

4.4 Future Consideration

- **Additional Operations:** Implementation of more complex operations, such as multiplication and division.
- **Integration with other CPU processes:** Use the ALU as part of a larger system, such as integrating it with other CPU processes like registers and memory.
- **Overflow and Error Detection:** Implement logic to detect and display overflow and underflow in arithmetic operations to improve reliability and robustness.