6-bit ALU Design

Introduction

I developed a 6-bit Arithmetic Logic Unit (ALU) using hardware and VHDL. This ALU was implemented entirely with the use of gate-level (structural) VHDL. Making the task significantly more challenging compared to behavioural VHDL, where constructs such as `if`, `case`, or loops can simplify operation selection and logic. Instead, the design had to rely entirely on (bool logic ie gates only) constructing operations and result selection using combinations of `AND`, `OR`, `NOT`, and `XOR` gates.

A key design choice was to simulate the behaviour of a multiplexer using pure Boolean logic. This allowed the ALU to select the correct operation output based on a 3-bit control signal (`operation`). Each operation result was precomputed and conditionally passed to the output using logical expressions replicating the selection process of a multiplexer in hardware.

The ALU supports six operations: addition, subtraction, AND, OR, XOR, and NOR. It accepts two 6-bit binary inputs (`A` and `B`) and produces a (6bit) output vector (`output`), along with three additional flags: `zero` active when the result is all 0s, `Cout` carry out from the final addition/subtraction bit, and `ofl` overflow flag for signed addition/subtraction.

Also most of the time was spent on subtraction operation, which required the use of two's complement. Since subtraction is not natively available (ie - sign) ,I had to utilize the standard method is to invert all bits of `B` using `NOT` and add 1 via a carry-in signal. This converts `B` into its two's complement form, enabling the circuit to perform `A - B` as `A + (~B + 1)` using only logic gates.

The ALU was tested using both simulation and hardware testing. A testbench (`test_alu.vhd`) was developed in Vivado with predefined values for `A`, `B`, and `operation`. These were used to generate waveforms, which were verified by converting them back into expected binary or hex outputs or checking specific bits for flag accuracy. Simulation results have shown that each operation worked correctly and matched expected logic behaviour.

In addition to simulation, I conducted hardware testing using the Basys3 FPGA board. I manually configured the constraints file to map specific switches (SW0–SW14) and LEDs (LD0–LD8) to the ALU's inputs and outputs. Each test case was carried out by physically moving the switches and observing LED responses, including the 3-bit operation selector. The output logic for all operations particularly `AND`, `OR`, `XOR`, and `NOR` followed their truth tables precisely, verifying it works on hardware.

The goal of producing a working 6-bit ALU using only gate-level VHDL was fully achieved. The design passed both simulation and physical board testing. Logic was verified through waveform analysis and manual input scenarios.

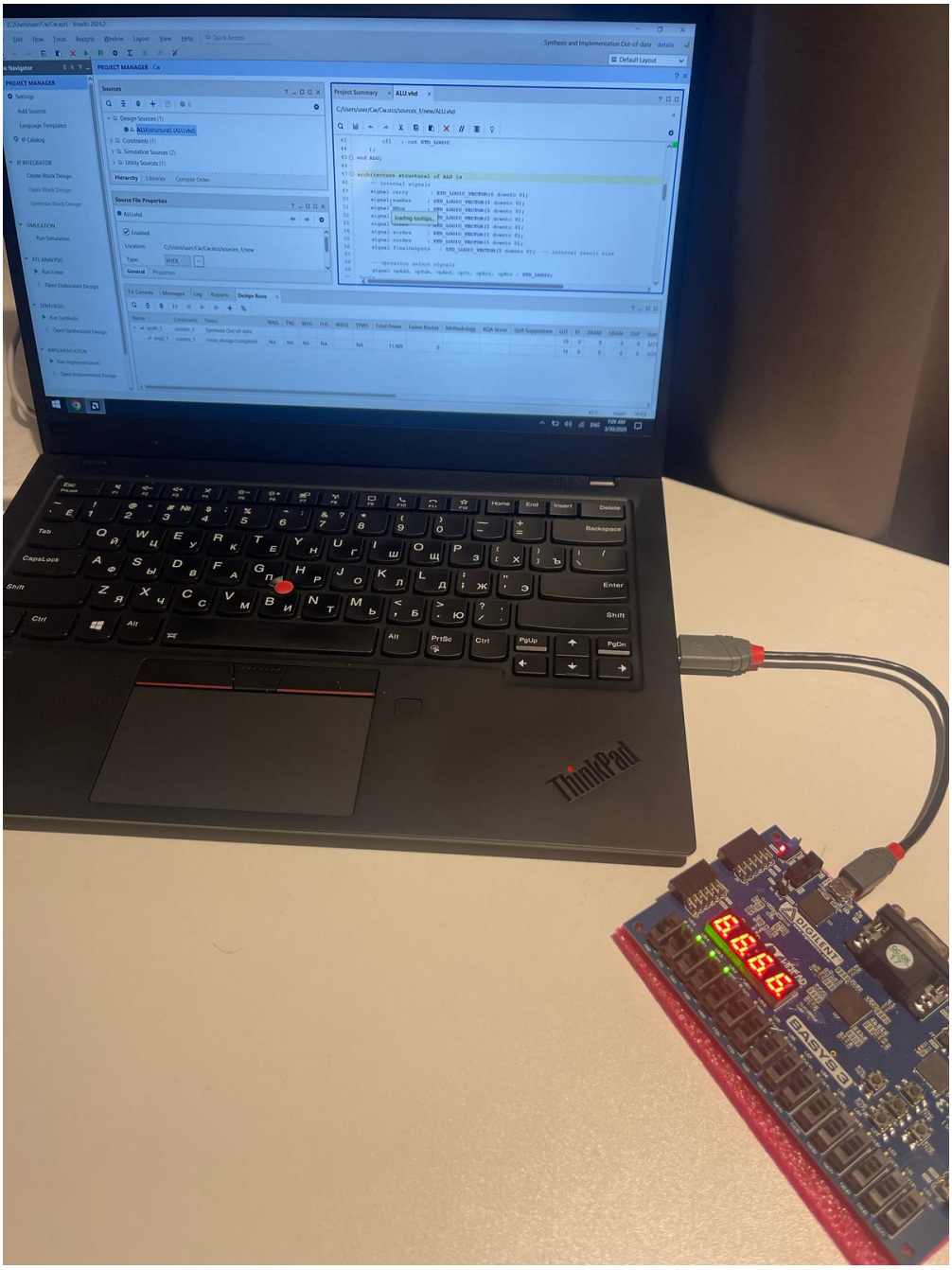

Figure 1: ALU Hardware Testing Setup on Basys3 Board

General Design Methodology

For this project I have decided to use a bottom-up design approach, starting from the most fundamental building block the adder.

The adder proved to be the most difficult module to implement. Initially, I planned to write a simple adder, but after further reading and reviewing lecture content, I decided to implement a ripple-carry adder, calculating the result and carry bit-by-bit using only `XOR`, `AND`, and `OR` gates. This was the best approach especially for the hardware part.

The subtractor was implemented next, and it leveraged the two's complement method, where `B` is inverted using `NOT` gates and a carry-in of 1 is introduced to the adder. This allowed subtraction to be performed using the same logic gates as addition.

After the arithmetic components, I moved on to implementing the logical operations `AND`, `OR`, `XOR`, and `NOR`. These were much easier, as they only required single gate operations applied bit-by-bit across inputs `A` and `B`.

For operation selection, I designed a logic-based solution to mimic the behaviour of a multiplexer. All operations were computed in parallel, and the correct one was selected by evaluating the operation input bits using Boolean expressions. This allowed me to avoid any behavioral constructs and the result could be routed purely through logic gates.

Throughout the development, testing each block individually proved to be an effective and interesting process. I followed the same bottom-up method for testing: after finishing one logic block, I would test it immediately. Once I verified the system worked, I would move on to the next step. This approach made debugging more manageable, as any errors discovered could only come from the most recent module earlier blocks were already known to be stable. This has made it easier to identify and fix bugs effecitvely.

I chose this methodology because it gave me full control over the code, allowed me to meet all project restrictions, and made debugging more intuitive, as the structure of each block was very predictable. It also ensured hardware accuracy, matching the way real digital circuits operate.

However, this approach did come with trade-offs. The design resulted in a much longer and more repetitive code, as each bit had to be processed manually with individual lines. Maintaining and navigating the code was more challenging, especially when debugging the overflow logic. Additionally, considerable attention was needed to ensure that the constraints file correctly mapped inputs and outputs to the physical switches and LEDs on the Basys3 board.

Adder Design and Implementation

```
-- adder caloualtion
sumRes(0) <= A(0) XOR BMux(0) XOR carry(0);
carry(1)  <= (A(0) AND BMux(0)) OR (carry(0) AND (A(0) XOR BMux(0)));
sumRes(1) <= A(1) XOR BMux(1) XOR carry(1);
carry(2)  <= (A(1) AND BMux(1)) OR (carry(1) AND (A(1) XOR BMux(1)));
sumRes(2) <= A(2) XOR BMux(2) XOR carry(2);
```

Figure 2: A reference diagram of a 1-bit full adder code.

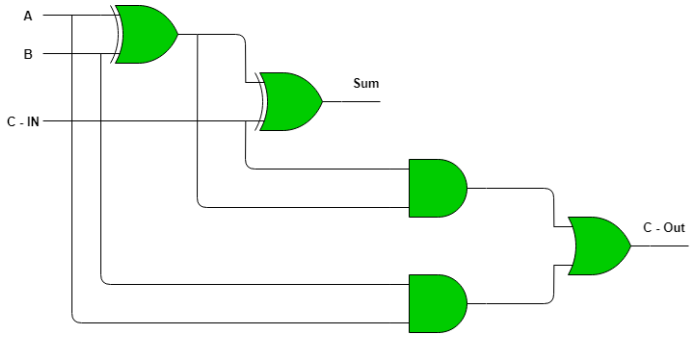

Figure 3: full adder diagram that was utilized for implementaiton of the system.

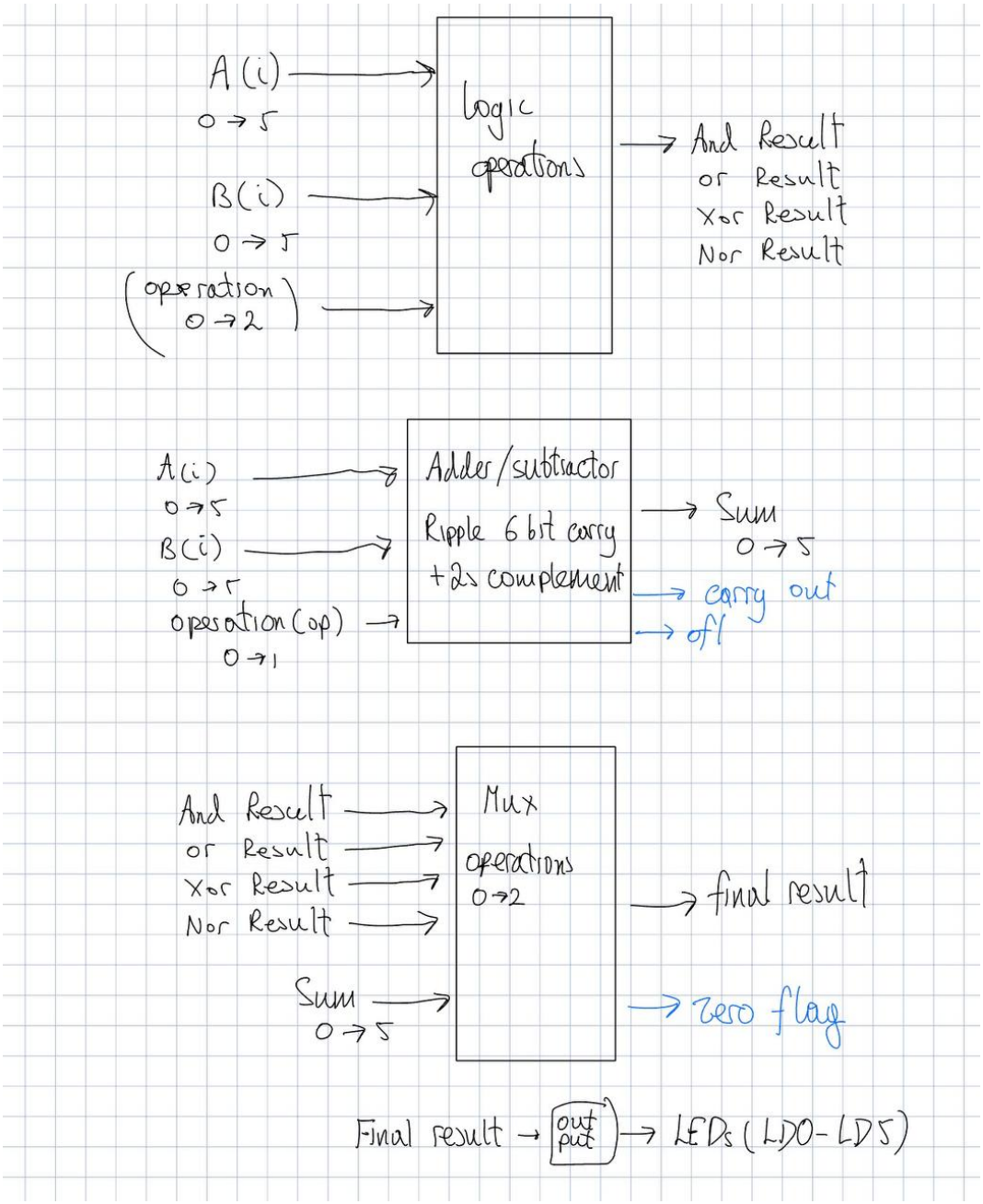

Figure 4: Overall ALU structure showing separate blocks for arithmetic, logic, and output selection.

1. Adder/Subtractor Unit, This was the first part I built. It takes inputs `A` and `B`, and based on the selected operation, it outputs either the sum or the difference. The logic for this is based on the ripple-carry adder structure, as shown in the code snippet above.

2. Logic Operation Unit, This section handles bitwise logic operations such as `AND`, `OR`, `XOR`, and `NOR`. Each operation processes the inputs in parallel and produces results.

3. Operation Selector (MUX-Style Logic), This block acts as a multiplexer, using a 3-bit control input (`operation`) to determine which result should be passed to the final output. It also calculates the zero flag if all output bits are 0.

Finally, the selected result is displayed via the LEDs on the Basys3 board, providing a clear hardware indication of the output.

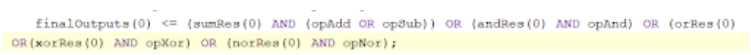

Figure 5: Snippet for the multiplexer logic showing how it selects which operation result picked

Detailed Design

The input to the ALU consists of two `STD_LOGIC_VECTOR(5 downto 0)` signals `A` and `B`. These represent 6-bit binary numbers, where `A(0)` and `B(0)` are the least significant bits (LSB) and `A(5)` and `B(5)` are the most significant bits (MSB). This vector format makes it easy to handle multibit values, enabling bit-level operations without declaring each bit individually. An additional 3-bit input, named `operation`, acts as the mode selector and determines which arithmetic or logic operation the ALU performs.

The design includes multiple parallel blocks. The adder/subtractor unit uses a ripple-carrying structure, built using gate-level logic for each bit. Subtraction is achieved by inverting the `B` input and setting the initial carry-in to 1, therefore creating a two's complement operation. This arithmetic block outputs both the result and the carry and overflow information. The logic operations - `AND`, `OR`, `XOR` and `NOR` are implemented separately, using simple gate logic and operate bit by bit across the six bits.

Boolean logic selector was implemented for the multiplexer. Each operation is computed in parallel, and a group of control signals (`opAdd`, `opSub`, `opAnd`, etc.) derived from the `operation` input determine which result is passed through to the output. This logic selector enables only the active operation's result to reach the output, whcih is the way its supposed to act in real world scenarios.

The final result is sent through a 6-bit output vector to the Basys3 board's LEDs. In addition to the result, the ALU also generates three output flags:

- **Zero flag**, set when all bits of the result are 0.
- **Carry-out flag**, taken from the final carry bit in the adder.
- **Overflow flag**, computed using the XOR of the last two carry bits.

One of the most challenging parts of the implementation was getting the overflow logic to work correctly. Initially, I misunderstood how overflow behaves in signed binary arithmetic and had difficulty identifying edge cases, especially when dealing with maximum and minimum values. After reviewing theory and analysing test cases, I managed to resolve the logic and implement an accurate overflow.

Ripple-Carry Adder Logic

The adder in my ALU is implemented using a ripple-carry structure, where each bit of the sum is calculated individually using basic logic gates (`XOR`, `AND`, `OR`). The carry output from each bit is passed to the next higher bit.

The logic for a 1-bit full adder is:

- `sum = A XOR B XOR Cin`
- `Cout = (A AND B) OR (Cin AND (A XOR B))`

This structure is repeated for all 6 bits, creating a 6-bit ripple-carry adder. The carry-in (`Cin`) of the first bit is set depending on whether the operation is an addition or subtraction (0 for add, 1 for subtract).

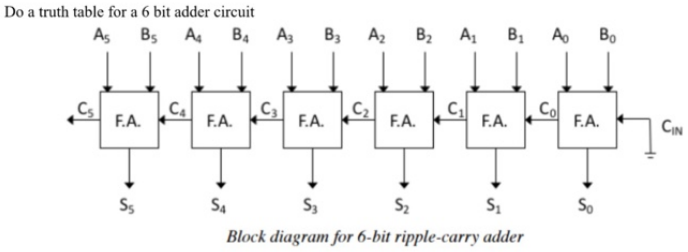

Figure 6: Structure of the 6-bit RCA.

Subtractor

In this project, subtraction is not implemented as a separate unit but cleverly integrated into the existing adder structure using the two's complement method.

The fundamental equation for two's complement subtraction is:

$$A - B = A + (\sim B + 1)$$

The diagram shows a 6-bit ripple-carry adder/subtractor, consisting of six Full Adder (FA) blocks. Each bit of input `A` and `B` enters a corresponding full adder stage, where the outputs produce the 6-bit result `S(res)(5 downto 0)`. The key subtraction logic is built into the `XOR` gates placed before the `B` inputs:

- These `XOR` gates take the input `B(i)` and a mode control signal `M`
- If `M = 0`: the `XOR` outputs `B(i)` → Normal addition
- If `M = 1`: the `XOR` outputs $\tilde{B}$`(i)` → Inversion for subtraction

The initial carry-in (`C0`, seen at the bottom left) is also:

- Set to 0 in addition mode
- Set to 1 in subtraction mode

Together, this produces the required operation: `A + `$\tilde{B}$` + 1`, i.e., `A - B`.

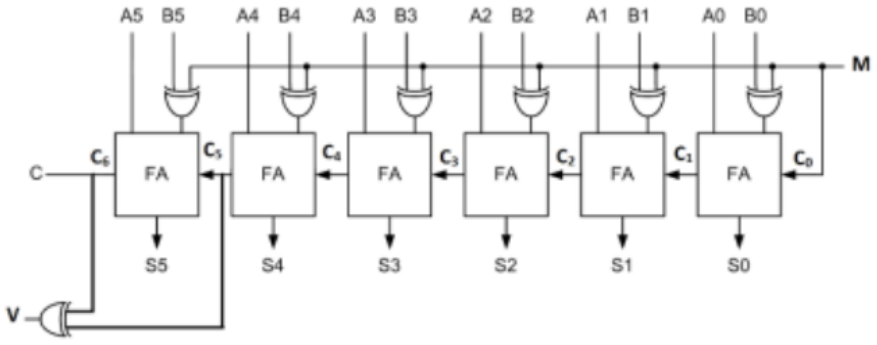

Figure 7: A 6-bit adder/subtractor using XOR gates and mode control to switch between addition and subtraction.

Multiplexer Logic

The MUX's purpose is to select which operation result (e.g., ADD, SUB, AND, OR) gets passed to the final output based on the 3-bit `operation` input.

Each operation (like ADD or SUB) has its own control signal (e.g., `opAdd`, `opSub`, etc.) that becomes active when the `operation` input matches its assigned binary code. For example, when the selector is `001`, it enables the subtraction mode, meaning `opSub = 1`.

What happens next is:

- The output from the subtractor (e.g., `sumRes`) is ANDed with `opSub`, so only the subtraction result passes through.

- Every other operation (AND, OR, etc.) is blocked because their control lines are 0, so their outputs are not used.
- All of these controlled outputs are then passed through an OR gate, and only the active one contributes to the final result.
- This is repeated bit-by-bit across all 6 output bits.

So the final MUX-like selection is just built from AND and OR gates, and whichever operation is enabled goes to output.

Operation Decision

The next key part of the design is the operation selection logic, which determines which block of the ALU (e.g. adder, subtractor, logic gates) becomes active based on a 3-bit input. This input A,B,C acts as the operation selector.

To handle this, I implemented a 3-to-8 decoder using only logic gates. Since I have 6 distinct operations (ADD, SUB, AND, OR, XOR, NOR), a 3-bit input is required to cover all possibilities. Three bits allow for 8 combinations (from 000 to 111), but only 6 are actually used. This means there are 2 don't care cases, which are marked in the schematic as dotted lines with the corresponding invalid operation code written beside them.

Each valid operation code activates a unique control signal like `opAdd`, `opSub`, etc. These control signals are generated through AND logic applied to combinations of the A, B, and C bits (and their negations) directly implementing the truth table of a decoder.

For example:

- `opAdd` is enabled when `operation = 000`
- `opSub` when `operation = 001`
- ... and so on, up to `opNor` for `111`

Each of these control signals is then used to gate the output of its respective logic block using AND gates, so that only one operation result is passed to the final MUX output.

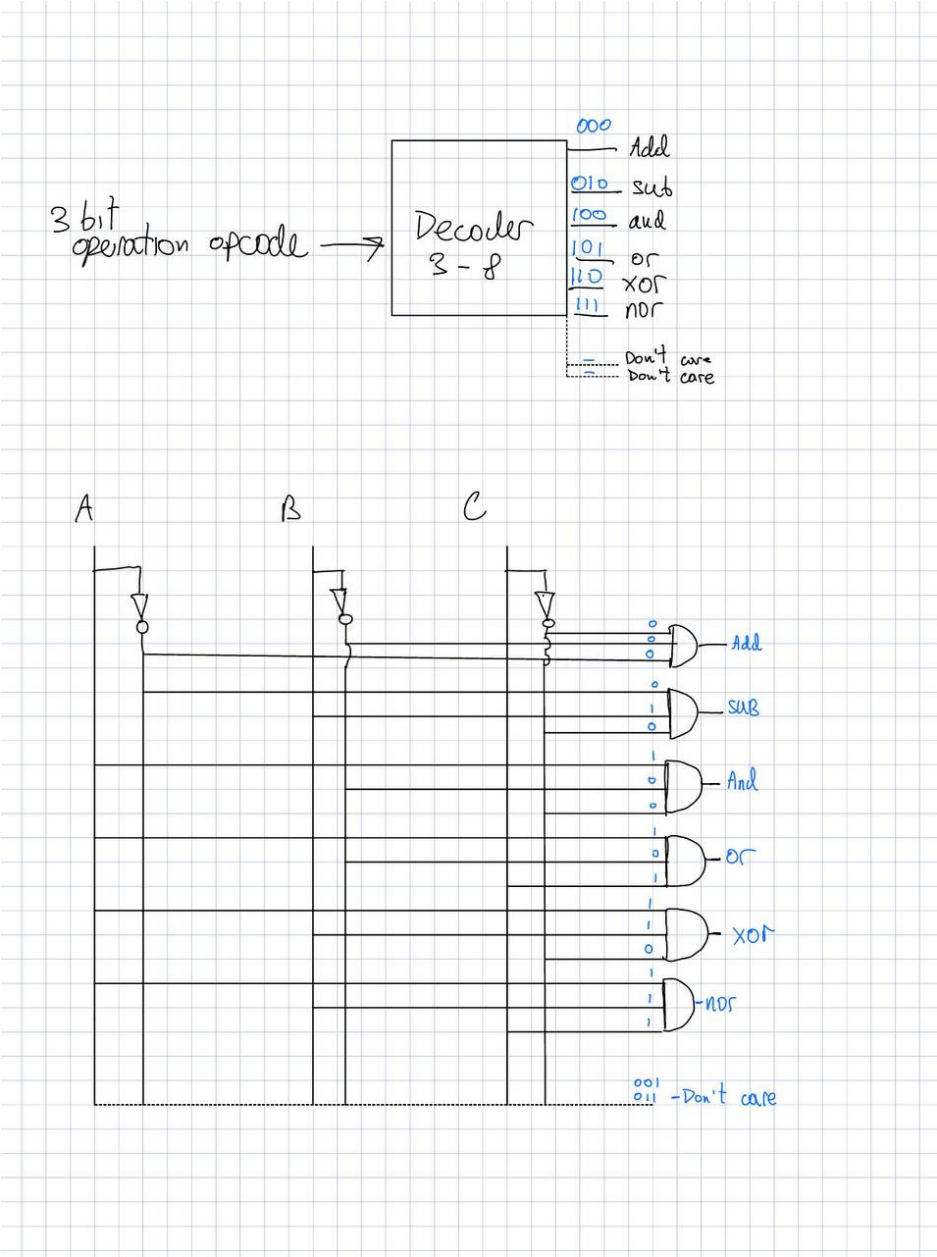

Figure 8: 3-8 decoder.

AluOp	Mnemonic	Result	Description
000	`add`	A + B	Addition
010	`sub`	A – B	Subtraction
100	`and`	A and B	Bitwise AND
101	`or`	A or B	Bitwise OR
110	`xor`	A xor B	Bitwise XOR
111	`nor`	A nor B	Bitwise NOR
Others	-	Don't Care	-

Figure 9: operations.

Design Summary

Several key design decisions shaped the structure of this ALU. A 6-bit ripple-carry adder was used for both addition and subtraction, with subtraction handled by inverting the `B` input and setting the initial carry-in, following the two's complement method. Logic operations such as `AND`, `OR`, `XOR` and `NOR` were computed in parallel, and a MUX selection was built using only `AND` and `OR` gates to choose the correct result. A simple 3-to-8 decoder logic controlled which operation was active based on a 3-bit input, with two unused combinations treated as don't care states.

In Figure 10, the diagram has been simplified a bit to make it easier to understand. It does not show the full details of the multiplexer or the complete logic for the arithmetic and logic operations. In actual design, each operation such as addition, subtraction, or logic gates has its own AND gate that runs at the same time as the others. These AND gates check if their operation was selected and only the correct one sends its result to the final output. This setup works like a multiplexer but is made using only basic logic gates. Also in my code I have made a redundant error which is now changed, however it was like this in my diagram this is where the ANDs and one or gate could have been replaced by one xor gate instead.

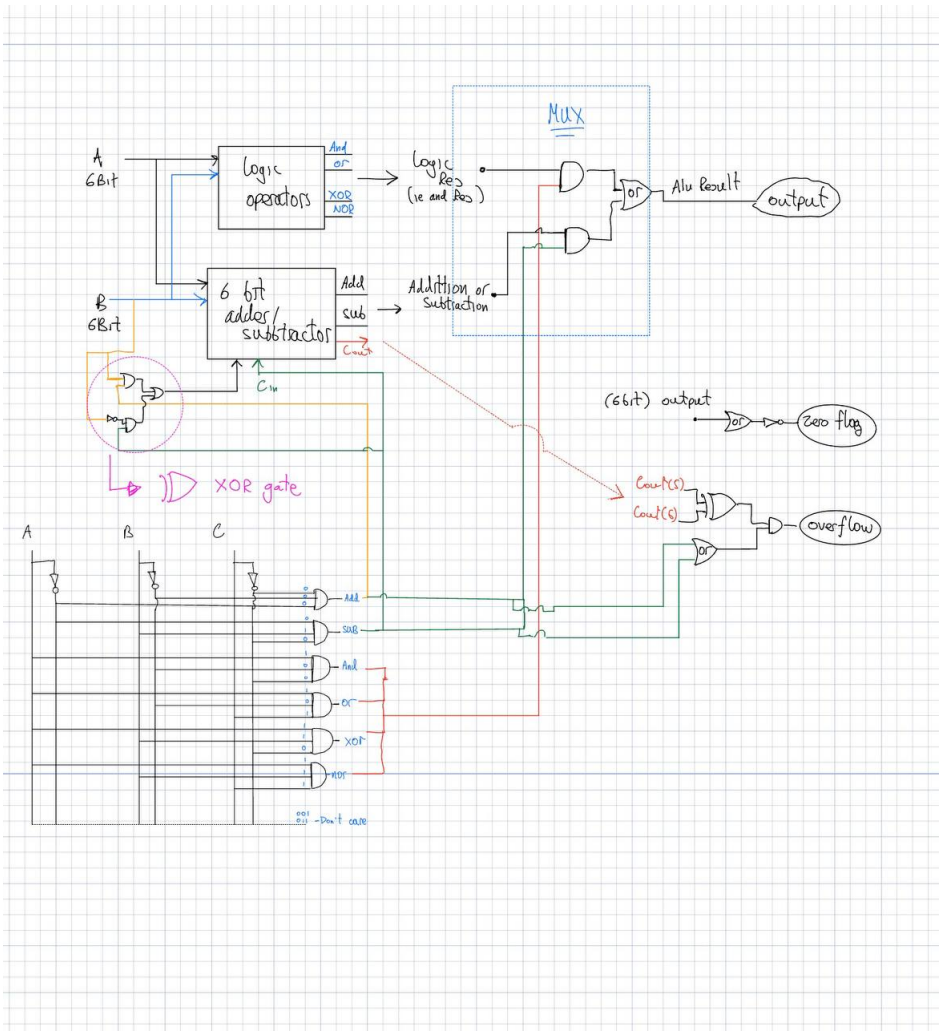

Figure 10: Final ALU

In the final ALU design, two important status flags were included: the zero flag and the overflow flag. The zero flag is calculated by first passing all six bits of the result through a series of OR gates. If none of the output bits are high meaning the result is `000000` the combined OR output is 0, which is then inverted with a NOT gate to produce 1, activating the zero flag. This confirms that the ALU's output is zero.

The overflow flag, on the other hand, is used during arithmetic operations such as addition and subtraction. It checks for overflow by comparing the carry into the most significant bit (`carry(5)`) and the carry out (`carry(6)`). If these two values differ, as detected by an XOR gate, it means the sign bit has flipped unexpectedly, indicating a signed overflow.

To ensure the overflow flag is only active during arithmetic modes, the XOR result is ANDed with the logic that enables either addition or subtraction.

Implementation

The switches on the Basys3 board were connected to the inputs `A(5:0)`, `B(5:0)`, and `AluOp(2:0)`, while LEDs were used to display the 6-bit result and the status flags (`Zero`, `Carry-out`, `Overflow`). These connections were defined in the constraints file using the proper pin assignments. This allowed for testing the ALU live on hardware using physical toggles and LED indicators.

Signal	Type	Bits / Flags	Basys3 Pins	Mapped To (Switch/LED)
`A(0) – A(5)`	Input	6-bit Input A	V17, V16, W16, W17, W15, V15	SW0 – SW5
`B(0) – B(5)`	Input	6-bit Input B	W14, W13, V2, T3, T2, R3	SW6 – SW11
`AluOp(0–2)`	Input	3-bit Operation Selector	W2, U1, T1	SW12 – SW14
`Cin`	Input (optional)	Carry-in (for subtraction)	R2	SW15 (if used)
`res(0) – res(5)`	Output (LED)	6-bit Result	U16, E19, U19, V19, W18, U15	LD0 – LD5
`zero`	Output (LED)	Zero Flag	U14	LD6
`cout`	Output (LED)	Carry-out Flag	V14	LD7
`ofl`	Output (LED)	Overflow Flag	V13	LD8

Figure 11: Switch and LED configuration for Basys3 board used in testing the ALU.

1-Bit Adder Code

The 1-bit adder is the base of the 6-bit ripple adder. It takes two input bits and a carry-in, then outputs a sum and a carry-out. The logic is implemented with basic gates:

- `sum_bit = A_bit XOR B_bit XOR carry_in`
- `carry_out = (A_bit AND B_bit) OR (carry_in AND (A_bit XOR B_bit))`

Ripple Carry Chain Logic

- `carry(0)` = 0 (for addition) or 1 (for subtraction)
- `sum(0) = A(0) XOR B(0) XOR carry(0)`
- `carry(1)` = generated by logic from previous bit
- ...
- `sum(5) = A(5) XOR B(5) XOR carry(5)`
- `carry(6)` = final carry used for flags (overflow, carry-out)

Subtraction using Two's Complement

Subtraction is done by inverting each bit of `B` and setting the initial carry-in to 1. This creates the two's complement form of B.

- `B_sub(i) = NOT B(i)`
- `carry(0) = 1`

MUX Logic (Result Selection)

Instead of using `if` or `case`, all possible results (add/sub, and, or, xor, nor) are calculated in parallel. Then only one result is selected using pure logic gates (`AND`, `OR`):

Example for a single bit:

```
result(0) = (sum(0) AND add_or_sub_active)
          OR (andRes(0) AND and_active)
          OR (orRes(0) AND or_active)
          OR ...
```

Operation Mode Selection

Each operation (ADD, SUB, AND, OR, XOR, NOR) is activated using combinational logic from the 3-bit `AluOp` selector:

```
opAdd = NOT op2 AND NOT op1 AND NOT op0   -- 000
opSub = NOT op2 AND op1 AND NOT op0       -- 010
...
opNor = op2 AND op1 AND op0               -- 111
```

AND Logic Implementation

The `AND` operation is performed bit-by-bit:

- `andRes(i) = A(i) AND B(i)` for i = 0 to 5

Overflow Flag Logic

The overflow flag is computed using the XOR of the carry into and out of the most significant bit:

- `ofl = carry(5) XOR carry(6)`
- `ofl = ofl AND (opAdd OR opSub)`

  This ensures overflow is only flagged for arithmetic operations.

Carry-Out Flag

The carry-out flag simply reflects the final carry:

- `cout = carry(6) AND (opAdd OR opSub)`

Zero Flag Logic

The zero flag checks if all output bits are zero:

- Combine all 6 output bits with `OR` gates
- Invert the result with a `NOT` gate
- Output is 1 if the result is `000000`

Schematic

A schematic is a visual representation of the digital circuit design, showing how each component in the ALU is connected using logic gates such as `AND`, `OR`, `XOR`, and `NOT`. Instead of writing out code, the schematic uses standard symbols to illustrate the flow of data from inputs (like `A`, `B`, and `AluOp`) through the internal logic blocks including the adder/subtractor, logical operation units, and selection logic all the way to the outputs such as the result vector and status flags (`zero`, `carry-out`, `overflow`).

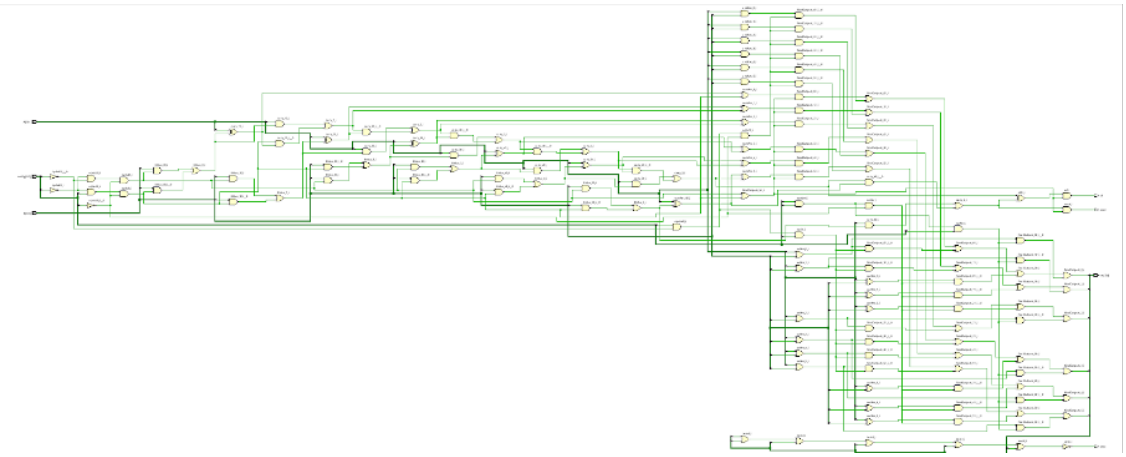

Figure 12: Full schematic.

Testing and Verification

#	A (hex)	B (hex)	ALUop	Operation	Result (hex)	Carry-out	Overflow
1	35	25	000	ADD	1A	1	1
2	0F	1F	000	ADD	22	0	1
3	21	3E	010	SUB	00	0	0
4	35	3E	010	SUB	37	0	0
5	12	28	111	NOR	05	0	0
6	3F	0C	101	XOR	33	0	0
7	12	28	110	OR	3A	0	0
8	3F	0C	100	AND	0C	0	0

Figure 13: Testbench used to verify the ALU's output across multiple operations.

This is the testbench for the ALU. Each result was compared against manual calculations to verify correctness. The outputs from simulation matched expectations exactly, confirming the logic and structure of the code.

1. **ADD Operation (ALUop = 000)** A = 35 (100101), B = 25 (011001)

- Binary sum: 100101 + 011001 = 111110
- Final result (6 bits): 111110 = 0x3E
- Carry-out = 1, Overflow = 1, Zero = 0

2. ADD Operation (ALUop = 000) A = 0F (001111), B = 1F (011111)

- Sum: 001111 + 011111 = 101110 (0x2E)
- Carry-out = 0, Overflow = 1, Zero = 0

3. SUB Operation (ALUop = 010) A = 21 (100001), B = 3E (111110)

- Two's complement of B = 000010
- Result: 100001 + 000010 = 100011 = 0x23
- Carry-out = 0, Overflow = 0, Zero = 0

4. SUB Operation (ALUop = 010) A = 35 (110101), B = 3E (111110)

- Two's complement of B = 000010
- Result: 110101 + 000010 = 110111 = 0x37
- Carry-out = 0, Overflow = 0, Zero = 0

5. NOR Operation (ALUop = 111) A = 12 (010010), B = 28 (101000)

- OR: 111010 $\rightarrow$ NOR: 000101 = 0x05
- Carry-out = 0, Overflow = 0, Zero = 0

6. XOR Operation (ALUop = 101) A = 3F (111111), B = 0C (001100)

- Result: 110011 = 0x33
- Carry-out = 0, Overflow = 0, Zero = 0

7. NOR Operation (ALUop = 110) A = 12 (010010), B = 28 (101000)

- OR: 111010 $\rightarrow$ NOR: 000101 = 0x05
- Carry-out = 0, Overflow = 0, Zero = 0

8. AND Operation (ALUop = 100) A = 3F (111111), B = 0C (001100)

- Result: 001100 = 0x0C
- Carry-out = 0, Overflow = 0, Zero = 0

Operation	A (dec)	B (dec)	A (bin)	B (bin)	Result (dec)	Result (bin)	Cout	Overflow
A + B	31	1	011111	000001	32 (Invalid)	100000	0	1
A + B	-32	-1	100000	111111	-33 (Invalid)	011111	1	1
A - B	-32	1	100000	000001	-33 (Invalid)	011111	1	1
A - B	31	-1	011111	111111	32 (Invalid)	100000	0	1
A - B	15	15	001111	001111	0	000000	1	0
A + B	-32	31	100000	011111	-1	111111	1	0

Figure 14: Simulation results for extreme cases involving overflow and carry-out in 6-bit signed operations.

To ensure the robustness and correctness of the ALU design, a series of extreme test were completed both in simulation and on the Basys3 hardware.

For example, when adding 31 and 1, the expected result is 32, which cannot be represented in 6-bit signed format instead, the ALU output wraps to `100000`, and the overflow flag is correctly raised. Similarly, subtracting -32 and 1 also leads to an invalid result (-33), again correctly flagging an overflow. These cases demonstrate that the overflow detection logic implemented using the XOR of the two most significant carry bits (`carry(5) XOR carry(6)`) performs as expected under critical conditions.

In each test, the result was verified through both the testbench waveform and physical output on the board, using the LED indicators for result and flags. The carry-out and overflow outputs were mapped to dedicated LEDs, and their behaviour during these tests matched the theoretical expectations, confirming the correctness of the ALU's flag logic under edge cases.

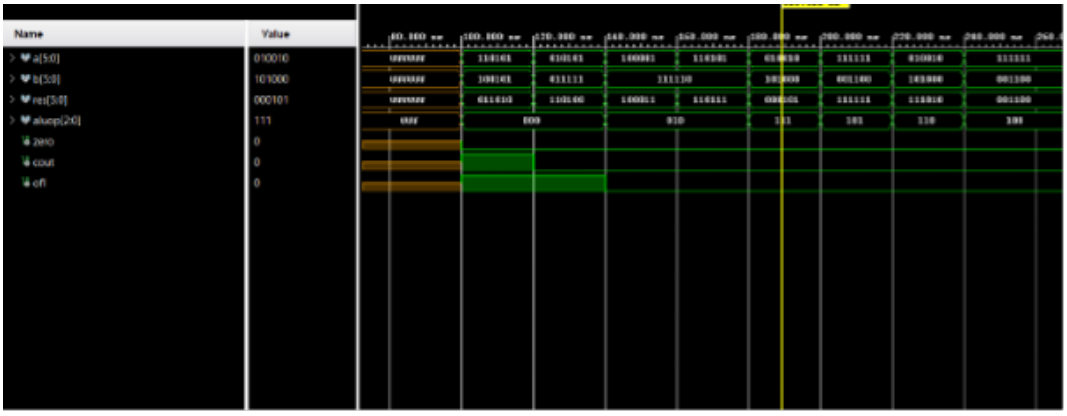

Figure 15: Final simulation waveform showing complete ALU operation verification in Vivado.