Design of an Arithmetical Logic Unit

Student: Codruta-Elen Jucan

Structure of Computer Systems Project

Technical University of Cluj-Napoca

Introduction

1.1 Context

The aim of this project is to design a standalone Arithmetic Logic Unit (ALU). The ALU is a fundamental digital circuit used for executing arithmetic and logical operations, which are essential in various computational processes. Designing an efficient ALU is crucial for ensuring fast and reliable handling of basic operations such as addition, subtraction, multiplication, division, and logical comparisons.

This project focuses on creating a versatile ALU that can be integrated into different digital systems, such as embedded devices, programmable logic controllers, or custom hardware designs. The resulting ALU design enhances the system's ability to perform specific computational tasks with greater accuracy and efficiency, making it suitable for a wide range of applications.

1.2 Objectives

The ALU will be designed in VHDL using Vivado, with an accumulator register as input and result storage for operations. To enhance user interaction, the design will integrate a seven-segment display unit for real-time output visualization. A debouncer will be added to handle input from a mechanical numpad, ensuring stable signals. These components will make the design interactive and efficient when programmed on an FPGA board.

The operations to be performed:

- Addition and subtraction in C2
- Increasing and decreasing
- AND, OR, NOT
- Negation
- Rotation (both left and right)
- Multiplication and division (implemented as distinct circuits)

Bibliographic Research

2.1 ALU general information and modular implementation

An **Arithmetic Logic Unit** is a critical digital circuit used within processors to perform a variety of arithmetic and logical operations. It executes tasks which are essential for data manipulation in computing systems.

Implementing an ALU **modularly** means breaking down the design into smaller, manageable subunits, where each module handles a specific function. This approach allows for better scalability and flexibility, as each module can be optimized or modified independently without affecting the entire system. In VHDL, these modules can be coded separately, simulated, and verified before being connected through a control unit that directs which operation to perform based on the given instruction. Modular design also means that the ALU will be flexible and scalable, ensuring that it handles data efficiently. [1]

2.2 FPGA implementation of ALU

FPGAs (Field-Programmable Gate Arrays) are ideal platforms for implementing ALUs due to their reconfigurability and parallel processing capabilities.These boards allow developers to test various ALU designs in real-time, using hardware simulation for verification before final implementation. The flexibility of the FPGA means that arithmetic operations such as addition and subtraction in two's complement form can be efficiently executed in parallel.

FPGA-based ALUs can outperform traditional ALUs in terms of speed and efficiency, especially when handling complex operations such as multiplication and division. Using dedicated multiplier circuits within the FPGA is important, reducing the computational load of these operations. It shows that a well-designed ALU can handle a wide range of operations with minimal delay, making FPGAs ideal for applications that require high processing power, such as embedded systems. [2]

2.3 Enhancing ALU Operations

The design focuses on optimizing performance by minimizing resource usage while maintaining the ability to execute various arithmetic and logical tasks. Two's complement arithmetic, which is essential for handling both positive and negative numbers, can be efficiently integrated into the design using minimal hardware resources. This is particularly useful for achieving a balance between complexity and performance. [3]

Also, bitwise logic operations can be optimized using VHDL to ensure fast execution times. Optimization techniques, such as reducing the number of logic gates needed for each operation, can help reduce power consumption and processing time, making the ALU more efficient when implemented on an FPGA. [4]

A set of numbers are stored in a memory, in the two's complement format. The operations performed will be efficient for this representation and for the 32-bit length:

- The addition and subtraction are implemented using a *carry lookahead adder*. [5]
- The multiplication uses *Shift and Add* for efficiency. [6]
- The division uses the *Restoring Division algorithm*, controlled by a finite state machine. [5]

Analysis

3.1 Project Proposal

The final design of the Arithmetic Logic Unit (ALU) will incorporate a range of essential operations and features, described in the list below.

1. **Addition and Subtraction** performed in two's complement (C2) to handle both positive and negative values.
2. **Increase and Decrease** operations, allowing efficient incrementing and decrementing of values stored in the accumulator.
3. **Logical Operations** including AND, OR, and NOT, enabling the ALU to perform bitwise logic on inputs.
4. **Negation**, allowing the ALU to compute the two's complement of a value efficiently.
5. **Rotation Operations** (left and right), shifting bits for specialized arithmetic and data manipulation.
6. **Multiplication and Division**, implemented as separate circuits, providing support for more complex arithmetic operations.
7. Multiple test programs to verify the correctness and performance of the ALU across various use cases.

3.2 Project Analysis

3.2.1 Control Unit

The control unit is the central component of a VHDL ALU with memory, responsible for orchestrating the entire operation. It receives the opcode from the user and decodes it to generate control signals that govern the data flow within the ALU.

These signals control the selection of operands from memory, the operation performed by the ALU, the storage of intermediate results, and the overall timing of the process. By effectively managing these control signals, the control unit ensures that the ALU executes the desired operation accurately and efficiently.

3.2.2 Algorithms

The ***Carry-Lookahead Adder*** (CLA) is a high-speed adder that significantly reduces the propagation delay compared to a ripple-carry adder. It achieves this by calculating carry bits in parallel rather than sequentially. The CLA employs a sophisticated logic network that analyzes the input bits to determine the carry-in and carry-out signals for each bit position simultaneously. This parallel calculation eliminates the need for the carry signal to ripple through each stage, leading to a significant speedup, especially for larger adders. By precomputing the carry bits, the CLA enables faster addition, making it a crucial component in high-performance digital systems.

The **Shift-and-Add algorithm** performs binary multiplication by iteratively examining each bit of the multiplier. If a bit is 1, the multiplicand is added to an accumulator. The multiplicand is then shifted left, and the multiplier is shifted right for the next bit. This process repeats until all bits of the multiplier are processed. The final value in the accumulator is the product.

Restoring division is a fundamental division algorithm that involves repeatedly subtracting the divisor from the dividend. In each iteration, the divisor is shifted left, and a subtraction is performed. If the result of the subtraction is positive, a '1' is placed in the quotient bit, and the result becomes the new dividend. If the result is negative, the original dividend is restored by adding back the divisor, and a '0' is placed in the quotient bit. This process continues until the quotient is fully formed. While straightforward, restoring division can be less efficient than other algorithms due to the potential need for restoration steps.

3.2.3 Use-case diagram

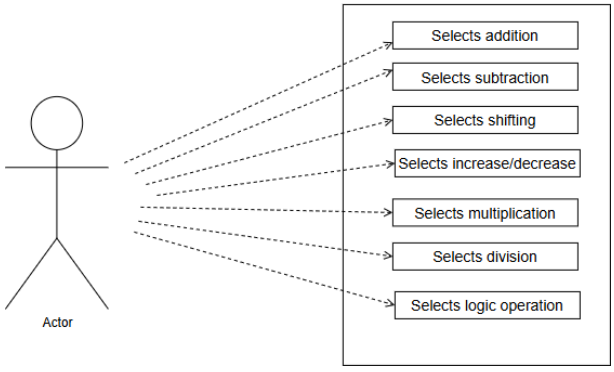

Figure 1 - Use-case diagram

- Use-case: Selects operation
  - Primary Actor: User
  - Main Success Scenario:
    1. The user selects the numbers they want the operation to be performed on
    2. The user selects the operation
    3. The user clicks on a “Confirm” button

4. The ALU performs the operation and displays the result

- Alternative Sequence: The user presses on a wrong button and the scenario returns to step 1.

3.2.4 State diagram

For better understanding the state diagram, this is the list of opCodes for each operation:

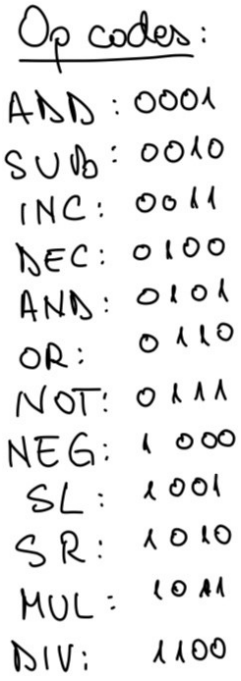

Figure 2 - Operation codes

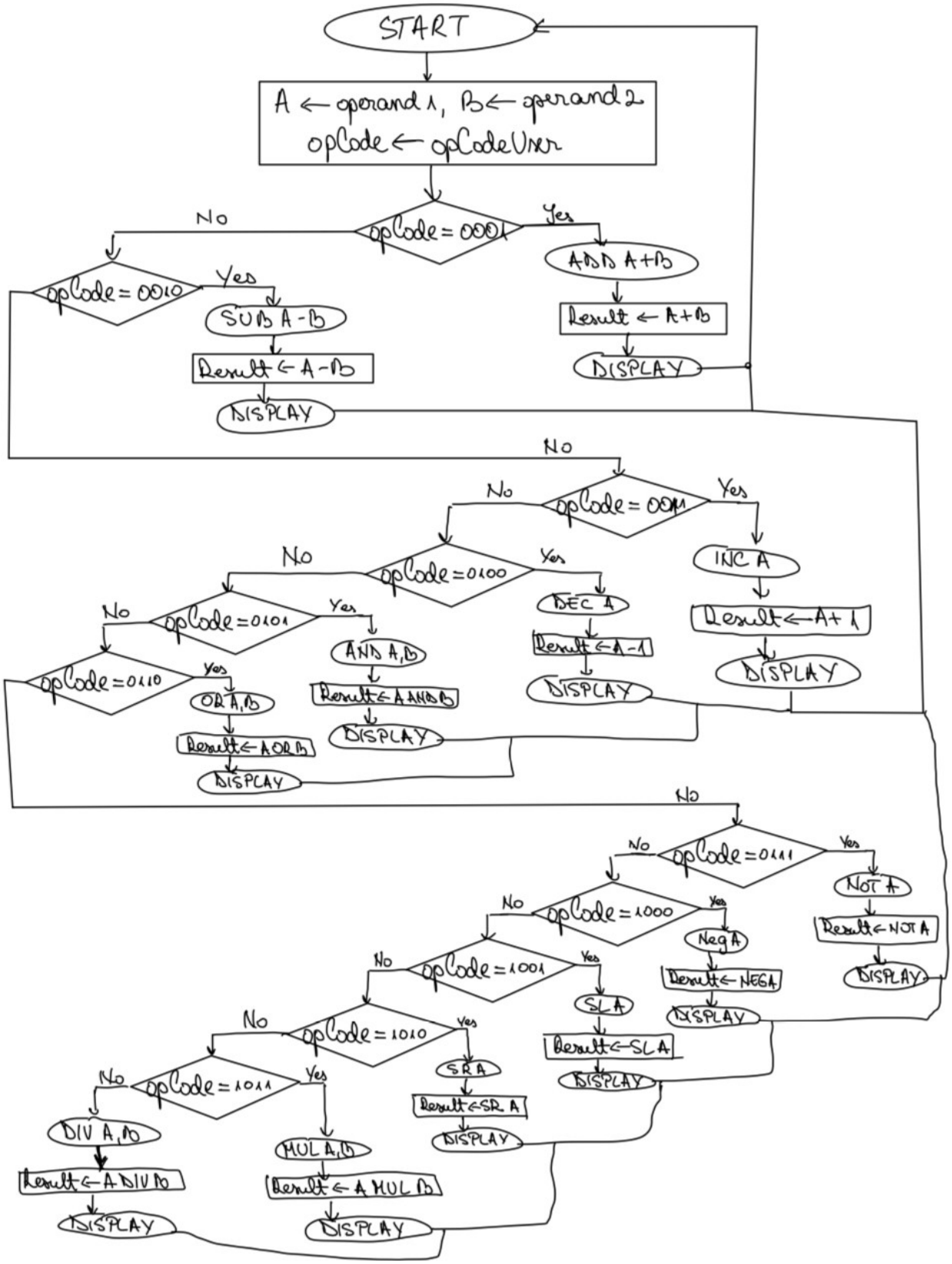

Figure 3 - State diagram

Design

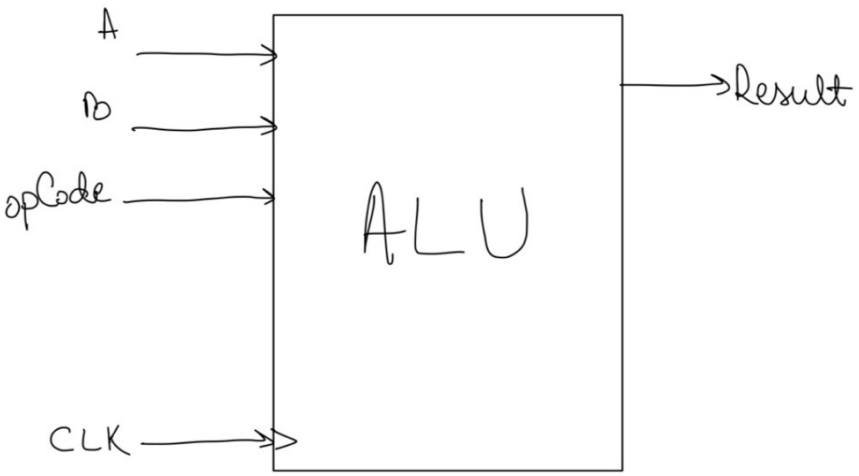

Figure 4 - ALU Blackbox

In the design shown in figure 5, the logic operations (AND, OR, NOT) are implemented in the same component, same with the arithmetic operations (addition, subtraction, increase, decrease), for efficiency.

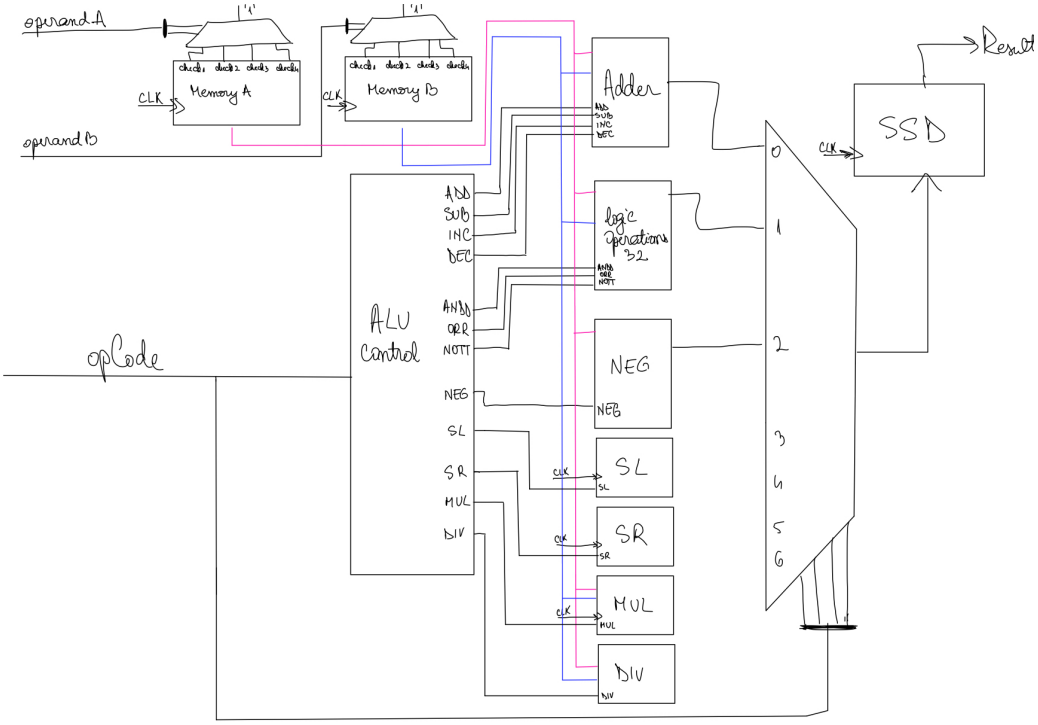

Figure 5 - Internal schematic

The logic operations are implemented this way, always checking the signal given by ALU control:

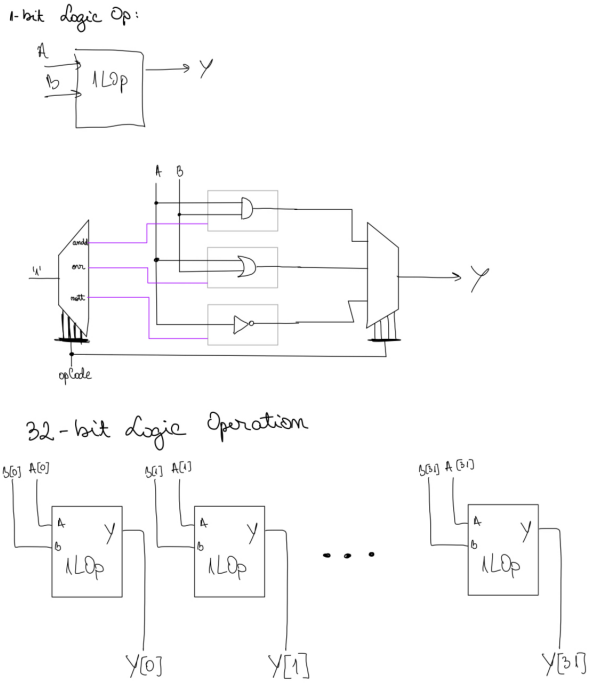

Figure 6 - Internal schematic of the Logic Operations component

The 32-bit adder is the main component used for implementing, besides addition, also subtraction, negation, increasing and decreasing:

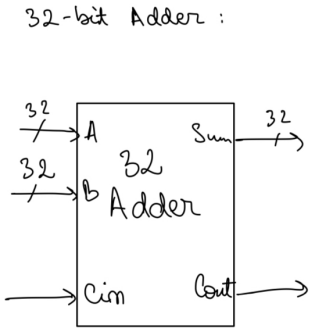

Figure 7 – Blackbox of the 32-bit Adder

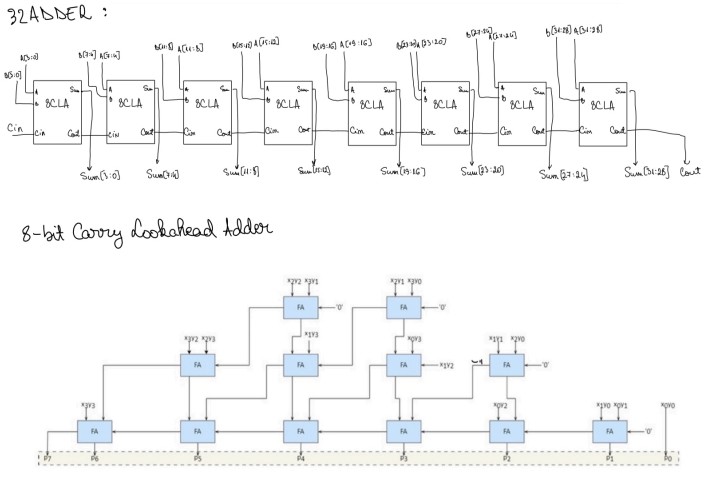

Figure 8 - Internal schematic of the 32-bit Adder

The negation is implemented using this adder, by negating the number and adding one to it:

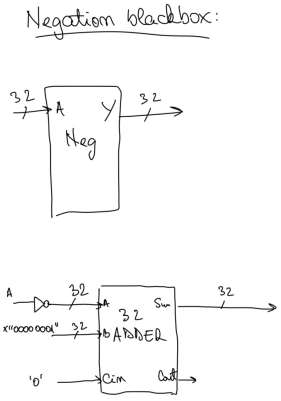

Figure 9 - Internal schematic of the Negation Component

Increasing is implemented by adding 1 to the number, subtraction is implemented by adding the negated number to the first one and decreasing is implemented by adding 1 negated to the first number, all of them using the 32-bit adder.

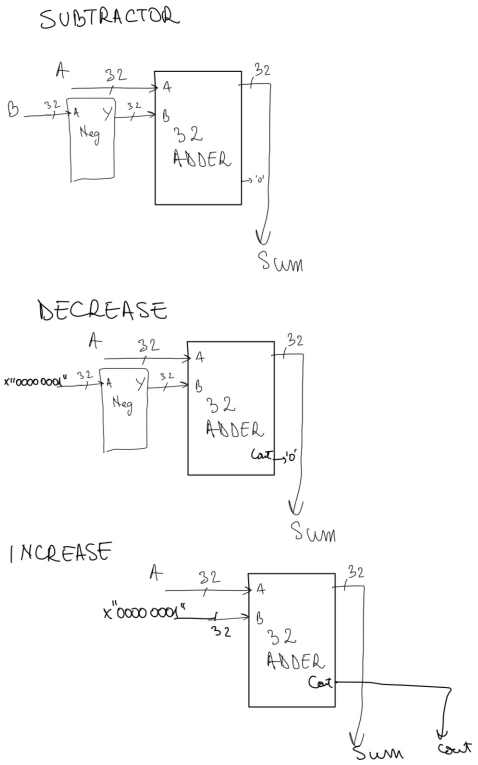

Figure 10 - Internal schematic of arithmetic operations

Implementation

The implementation stage starts with designing and testing each operation, to see that they work individually properly. After that, the SSD, multiplexers and memories are implemented, helping to get to the usable ALU by connecting all the pieces together.

5.1. ALU Control

ALU Control

- opCode: in std_logic_vector(3 downto 0)
- add, sub, inc, dec, neg, andd, orr, nott, sl, sr, mul, div: out std_logic

ALU Control generates enable signals for each operation, depending on the opcode given by the user.

```
architecture Behavioral of aluControl is
begin
 process(opCode)
  begin
        add <= '0';
        sub <= '0';
        inc <= '0';
        dec <= '0';
        neg <= '0';
        nott <= '0';
        andd <= '0';
        orr <= '0';
        sl <= '0';
        sr <= '0';
        mul <= '0';
        div <= '0';

        case opCode is
          when "0001" => add <= '1';
          when "0010" => sub <= '1';
          when "0011" => inc <= '1';
          when "0100" => dec <= '1';
          when "0101" => andd <= '1';
          when "0110" => orr <= '1';
          when "0111" => nott <= '1';
          when "1000" => neg <= '1';
          when "1001" => sl <= '1';
          when "1010" => sr <= '1';
          when "1011" => mul <= '1';
          when "1100" => div <= '1';
          when others => null; -- Do nothing for undefined opCodes
        end case;
  end process;
end Behavioral;
```

Figure 11 - ALU Control Architecture

5.2. Logic operations (AND, OR, NOT)

32-bit Logic Operations entity

- A, B: in std_logic_vector(31 downto 0)
- andd, orr, nott: in std_logic
- Y: out std_logic_vector(31 downto 0)

The andd, orr and nott inputs are the enables transmitted by the ALU_Control, permitting the component to perform one of the operations. Using the 1-bit component that verifies what operation to perform, the final 32-bit component to implement logic operations will simply generate 32 such components for each position in the 32-bit vectors.

5.3. Negation

32-bit Negation entity

- A: in std_logic_vector(31 downto 0)
- Y: out std_logic_vector(31 downto 0)

This entity's result is the opposite of the input in two's complemented format. It's implemented by negating all bits and adding 1.

```
architecture Behavioral of negation is
begin

  Y <= std_logic_vector(signed(NOT A) + 1);

end Behavioral;
```

Figure 12 - Negation Architecture

5.4. Adder

32-bit Adder entity

- add, sub, inc, dec: in std_logic
- A, B: in std_logic_vector(31 downto 0)
- Cin: in std_logic
- Sum: out std_logic_vector(31 downto 0)
- Cout: out std_logic

This component performs a part of the required arithmetic operation (addition, subtraction, increase, decrease), using the enable signals generated by ALU Control. For every operation, the component modifies the inputs:

- The inputs stay the same for ***addition***
- The second input is negated for ***subtraction***
- The second input is one for ***increasing***
- The second input is one negated for ***increasing***

After modifying the inputs such that this set of operations can be performed, the inputs will be given to a 32-bit adder.

The 32-bit adder used for performing the arithmetic operations is implemented by cascading 8 4-bit Carry Lookahead adders this way:

```
CLA0: carryLookaheadAdder port map(A(3 downto 0), B(3 downto 0), Cin,
     Sum(3 downto 0), c(0));
CLA1: carryLookaheadAdder port map(A(7 downto 4), B(7 downto 4), c(0)
    , Sum(7 downto 4), c(1));
CLA2: carryLookaheadAdder port map(A(11 downto 8), B(11 downto 8), c
    (1), Sum(11 downto 8), c(2));
CLA3: carryLookaheadAdder port map(A(15 downto 12), B(15 downto 12),
    c(2), Sum(15 downto 12), c(3));
CLA4: carryLookaheadAdder port map(A(19 downto 16), B(19 downto 16),
    c(3), Sum(19 downto 16), c(4));
CLA5: carryLookaheadAdder port map(A(23 downto 20), B(23 downto 20),
    c(4), Sum(23 downto 20), c(5));
CLA6: carryLookaheadAdder port map(A(27 downto 24), B(27 downto 24),
    c(5), Sum(27 downto 24), c(6));
CLA7: carryLookaheadAdder port map(A(31 downto 28), B(31 downto 28),
    c(6), Sum(31 downto 28), Cout);
```

Figure 13 - Cascading Carry Lookahead Adders

5.5. Shift Left, Shift Right

Shift Register

- **clk:** *in* STD_LOGIC
- **dir:** *in* STD_LOGIC
- **data_in:** *in* STD_LOGIC_VECTOR(31 downto 0)
- **ready:** *out* STD_LOGIC
- **data_out:** *out* STD_LOGIC_VECTOR(31 downto 0)

The register works by storing a 32-bit input value in a register on a rising clock edge. If a reset signal is active, the register is loaded with the data_in value. When the enable signal is active, the register performs a right or left shift operation, where the least significant bit (LSB), respectively the most significant bit (MSB) is shifted out, and a 0 is inserted at the most significant bit (MSB), respectfully the least significant bit. This process effectively shifts the contents of the register to the right/ left by one bit each time the clock pulses.

5.6. Multiplication

Shift & Add Multiplier

- **clk:** *in* STD_LOGIC
- **x, y:** *in* STD_LOGIC_VECTOR(31 downto 0)
- **ready:** *out* STD_LOGIC
- **result:** *out* STD_LOGIC_VECTOR(63 downto 0)

The ***Shift and Add*** multiplier sequentially computes the product of two 32-bit signed integers (x and y) based on the shift-and-add algorithm. It involves initializing partial products and iteratively adding the multiplicand (B) to the accumulator (A) whenever the least significant bit of the multiplier (Q) is 1. Each iteration shifts B left and Q right while decrementing the iteration count (N) until all bits of the multiplier have been processed. The design includes state-based control for handling initialization, computation, and result generation. Two 32-bit carry-lookahead adders are utilized for efficient addition, and signed multiplication is managed by converting inputs to two's complement when necessary. The final result is output in the result signal, with a resultReady signal indicating computation completion.

```
when Handle_inputs =>
    if x(31) = '1' then
        nr1 <= std_logic_vector(unsigned(not x) + 1);
    else
        nr1 <= x;
    end if;

    if y(31) = '1' then
        nr2 <= std_logic_vector(unsigned(not y) + 1);
    else
        nr2 <= y;
    end if;

    current_state <= state_init;

when state_init =>
    B <= "00000000000000000000000000000000" & nr1;
    A<= (others => '0');
    Q <= nr2;
    N <=32;
    current_state <= state_calc;

when state_calc =>
    if (N > 0) then
          if Q(0) = '1' then
                A <= Sum2 & Sum1;
           end if;
     B <= B(62 downto 0) & '0';
     Q <= '0' & Q(31 downto 1);
     N <= N - 1;
    else
      current_state <= state_done;
    end if;

when state_done =>
    if (x(31) XOR y(31)) = '1' then
        A<= std_logic_vector(unsigned(not A) + 1);
    end if;
    ready <= '1';
    current_state <=Idle;
```

Figure 14 - Multiplication process

5.7. Division

Divider

- x, y: in std_logic_vector(31 downto 0)
- q, r: out std_logic_vector(31 downto 0)

The implementation defines a 32-bit divider module that computes both the quotient and remainder of two input values (x and y). The algorithm follows a step-by-step binary division process, similar to the manual long division method, extended for digital logic. The dividend is initialized by appending 32 zeros to the 32-bit input x to create a 64-bit value. The divisor is simply the 32-bit input y. Two 32-bit variables, quotient and remainder, are used to store the intermediate results during the computation.

The division algorithm works iteratively over 32 cycles, where in each cycle, the current remainder is shifted left, and the most significant bit of the dividend is appended. The algorithm then subtracts the divisor from this updated remainder. If the subtraction yields a negative result, the operation is undone by adding the divisor back, and a 0 is appended to the quotient. Otherwise, a 1 is appended to the quotient. After the loop completes, the quotient and remainder contain the final results of the division, which are then assigned to the outputs q and r as std_logic_vector signals.

This design ensures that both the quotient and remainder are computed accurately using a hardware-friendly iterative approach. It is particularly suited for FPGA and ASIC implementations, where resource efficiency and deterministic timing are critical. The use of the numeric_std library enables seamless arithmetic operations with unsigned values, ensuring the division logic adheres to VHDL's best practices.

```
PROCESS(x, y)
    VARIABLE dividend : unsigned(63 downto 0);
    VARIABLE divisor  : unsigned(31 downto 0);
    VARIABLE quotient : unsigned(31 downto 0);
    VARIABLE remainder : unsigned(31 downto 0);
  BEGIN
    -- Initialize variables
    dividend := unsigned(x) & x"00000000"; -- Extend dividend to 64 bits
    divisor := unsigned(y);
    quotient := (others => '0');
    remainder := (others => '0');

    -- Perform division algorithm
    FOR i IN 31 DOWNTO 0 LOOP
      -- Shift remainder left and bring down the next bit from dividend
      remainder := (remainder(30 downto 0) & dividend(63)) - divisor;
      dividend := dividend(62 downto 0) & '0'; -- Shift dividend left

      -- Check if subtraction result is negative
      IF remainder(31) = '1' THEN
        remainder := remainder + divisor; -- Undo subtraction
        quotient := quotient(30 downto 0) & '0'; -- Append 0 to quotient
      ELSE
        quotient := quotient(30 downto 0) & '1'; -- Append 1 to quotient
      END IF;
    END LOOP;

    -- Assign outputs
    q <= std_logic_vector(quotient);
    r <= std_logic_vector(remainder);
  END PROCESS;
```

Figure 15 - Division process

5.8. SSD

Seven Segment Display

- **digit0, digit1, digit2, digit3:** *in* STD_LOGIC_VECTOR(3 downto 0)
- **clk:** *in* STD_LOGIC
- **cat:** *out* STD_LOGIC_VECTOR(6 downto 0)
- **an:** *out* STD_LOGIC_VECTOR(3 downto 0)

This component implements a 7-segment display driver capable of cycling through four 4-bit digits (digit0, digit1, digit2, digit3) and displaying them sequentially on a common-anode 7-segment display. A 16-bit counter (cnt) is incremented at every clock cycle, and the two most significant bits of the counter (cnt(15 downto 14)) determine which digit is active and which anode signal (an) is enabled.

The selected digit is decoded into its corresponding 7-segment representation (cat) using a with-select statement that maps hexadecimal values to segment patterns. This design allows for multiplexing four digits on a single 7-segment display, refreshing them fast enough to appear steady to the human eye. The approach ensures efficient use of hardware resources and precise control of display output.

5.9. Memory

Memory

- **clk:** *in* STD_LOGIC
- **address:** *in* STD_LOGIC_VECTOR(4 downto 0)
- **data_out:** *out* STD_LOGIC_VECTOR(31 downto 0)

The memories serve as storage units for the operands used in ALU operations. Each memory contains preloaded 32-bit values, which can be accessed using specific addresses. Memory1 provides the first operand (address1), and Memory2 provides the second operand (address2). By using these memories, the user can efficiently test various operations with predefined values, enabling consistent and repeatable simulations while simplifying the input handling for your ALU design.

MEMORY1

Address	Number
00000	x7FFFFFFF
00001	x12345678
00010	x000F FFFF
00011	xFFFF FFFF
00100	x55555555
00101	x77359400

Figure 16 - Hardcoded memory for the first operand

MEMORY2

Address	Number
00000	x89ABCDEF
00001	x12345671
00010	x0000FFFF
00011	x00000002
00100	xAAAAAAAA
00101	x00000000
00110	x7FFFFFFF
00111	x59682F00

Figure 17 - Hardcoded memory for the second operand

5.10. Top Level

The top level integrates memory access and arithmetic logic operations, functioning as a simple processing unit. It reads two inputs from memory, decodes an operation code (opcode), and executes the corresponding operation—such as arithmetic, logical, shifting, multiplication, or division—using dedicated components. Results are routed to output ports (result1, result2), along with a carry flag (cout) for arithmetic operations. The design combines modular components like memory, ALU, and control logic to perform a wide range of operations in a synchronized, clock-driven manner.

When loaded on the FPGA board, the user writes the addresses of the operands and the opcodes, using switches. To see the full results, there are 3 buttons to use, each displaying 16-bit parts of the output. In order to see the special cases which can occur in this design, there are two LEDS signaling the overflow and the division by 0.

Testing and validation

Tests are made for each operation, using the top level, accordingly to the table shown below:

Operation	Opcode	Operand 1	Address 1	Operand 2	Address 2	Result
ADD	0001	x7FFFFFFF	00000	x12345671	00001	x92345670
ADD	0001	x 47359400	00101	x59682F00	00011	LED
SUB	0010	x12345678	00001	x89ABCDEF	00000	x88888889
INC	0011	x12345678	00001	—	—	x12345679
DEC	0100	x12345678	00001	—	—	x12345677
AND	0101	x55555555	00100	xAAAAAAAA	00100	x00000000
OR	0110	x55555555	00100	xAAAAAAAA	00100	xFFFFFFFF
NOT	0111	x12345678	00001	—	—	xEDCBA987
NEG	1000	—	—	x7FFFFFFF	00110	x80000001
MUL	1011	x0000FFFF	00010	x0000FFFF	00010	xFFFE0001
DIV	1100	xFFFFFFFF	00011	x00000002	00011	x7FFFFFFF
DIV	1100	xFFFFFFFF	00011	x00000000	00101	LED
MUL	1011	x12345678	00001	x12345671	00001	01AB66A8 9E86 7AF8
SL	1001	—	—	x00000002	00011	x00000004
SR	1010	—	—	x00000002	00011	x00000001

Figure 18 - Table showing the hardcoded values and their addresses

Addition

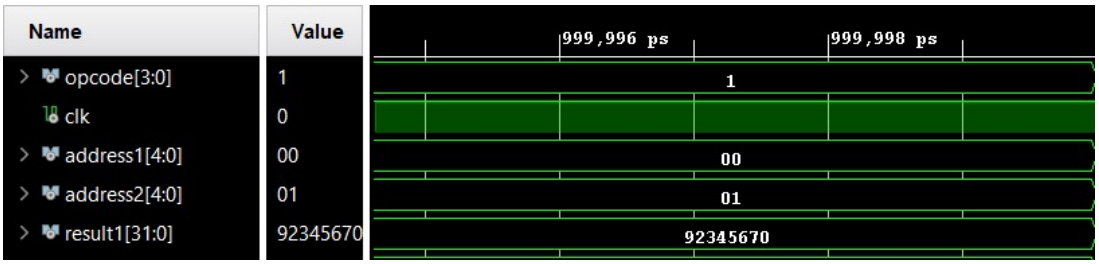

Figure 19 - Test1 for addition

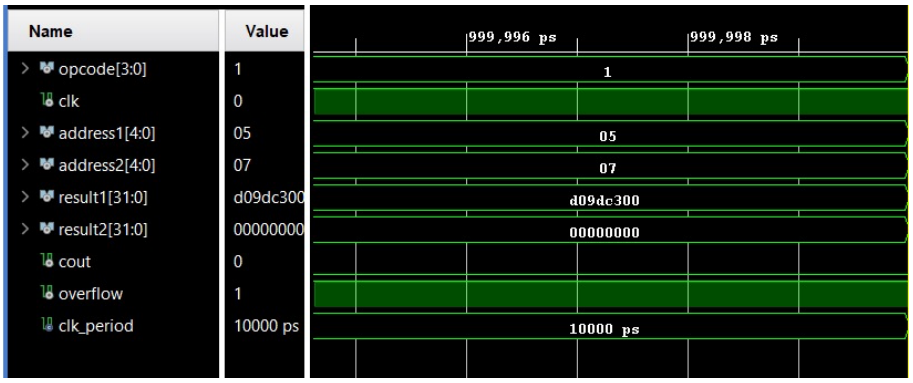

Figure 20 - Test2 for addition

Subtraction

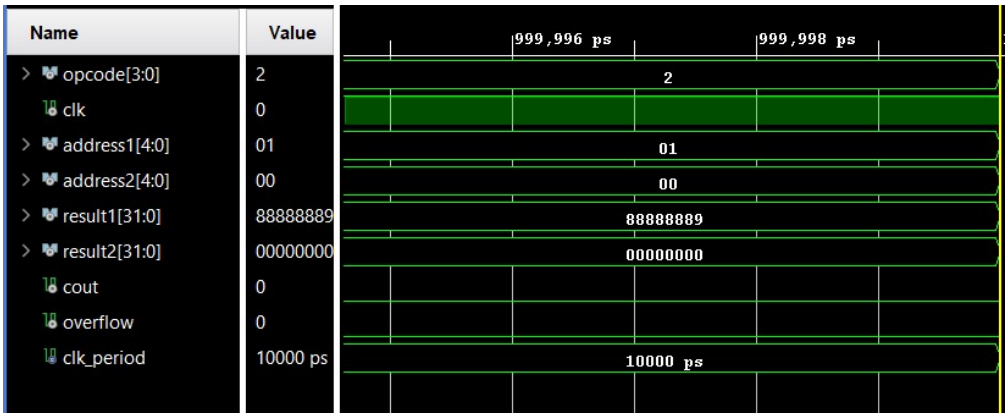

Figure 21 - Test for subtraction

Incrementing

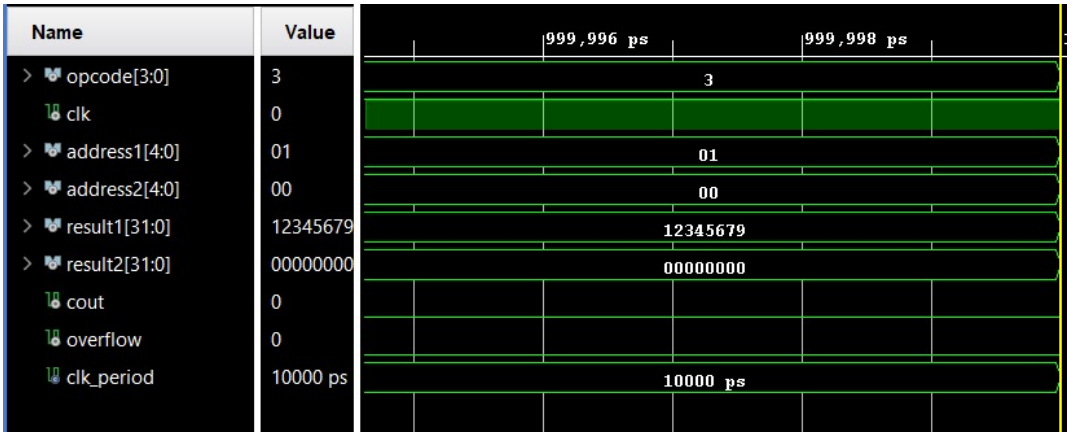

Figure 22 - Test for incrementing

Decrementing

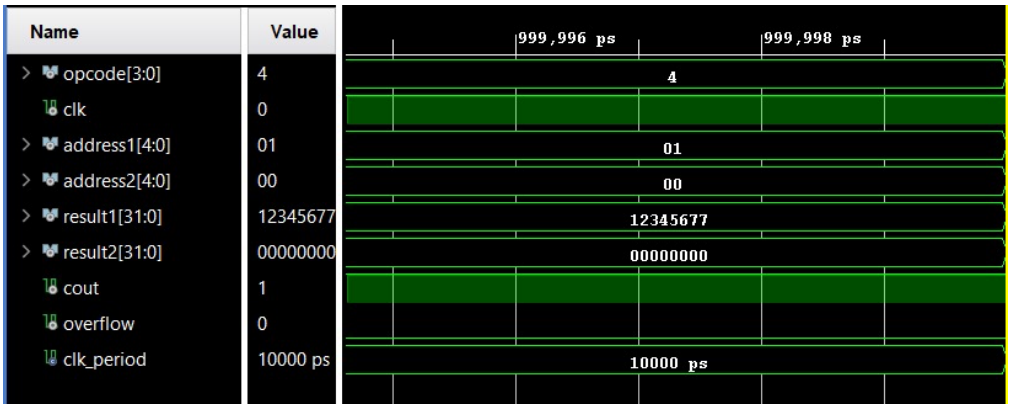

Figure 23 - Test for decrementing

AND

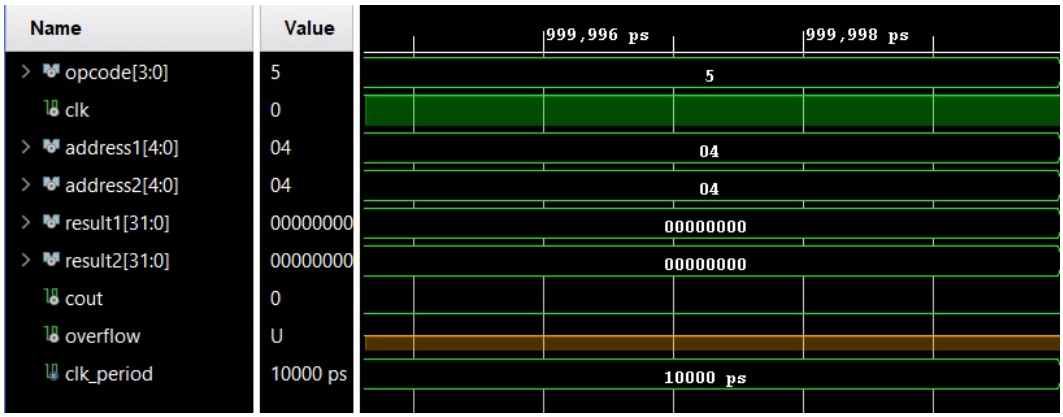

Figure 24 - Test for AND

OR

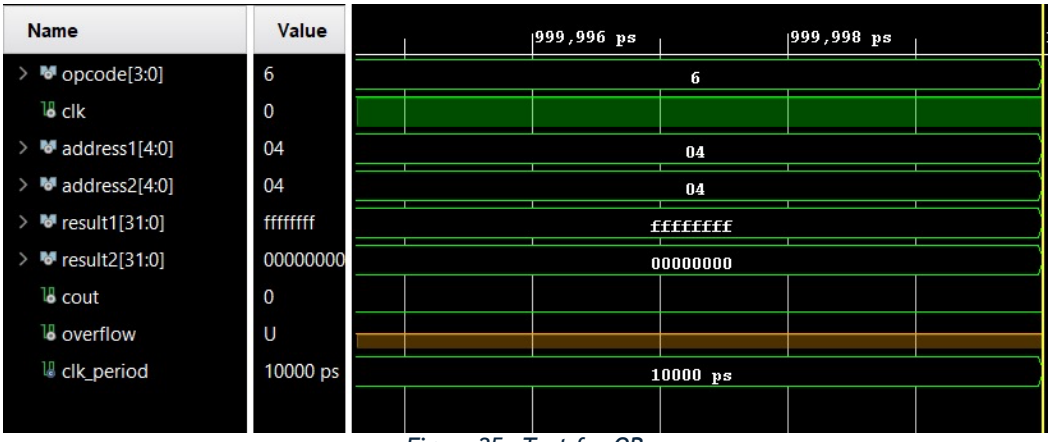

Figure 25 - Test for OR

NOT

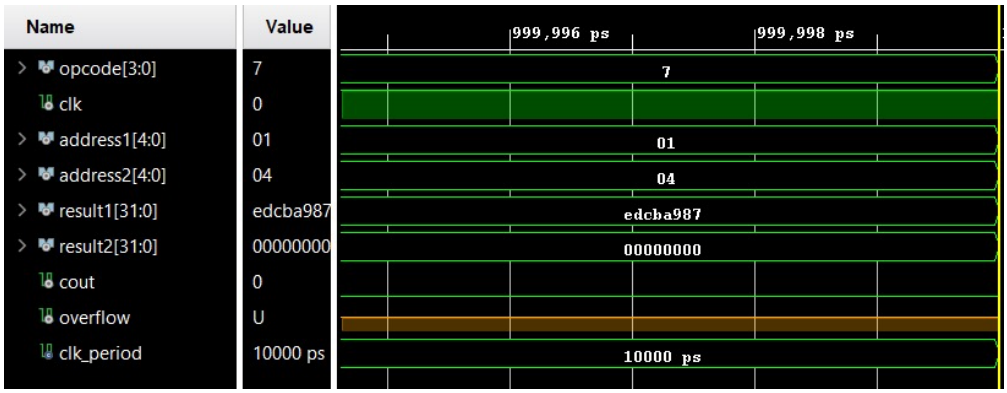

Figure 26 - Test for NOT

Negation

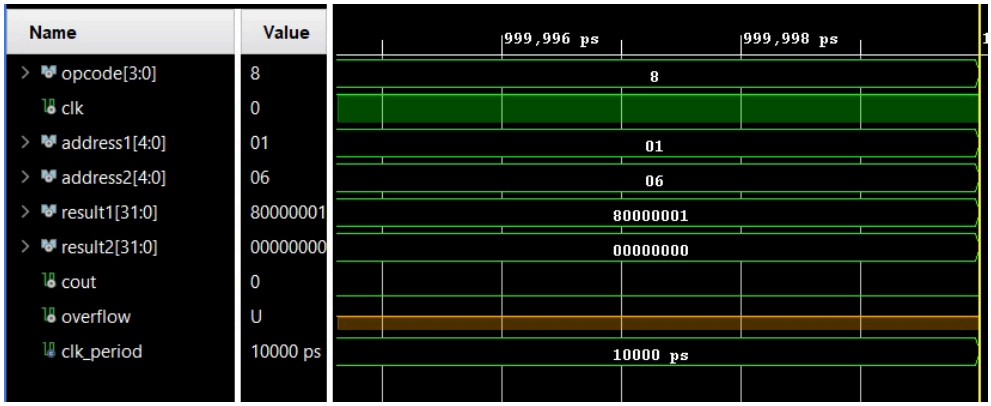

Figure 27 - Test for negation

Multiplication

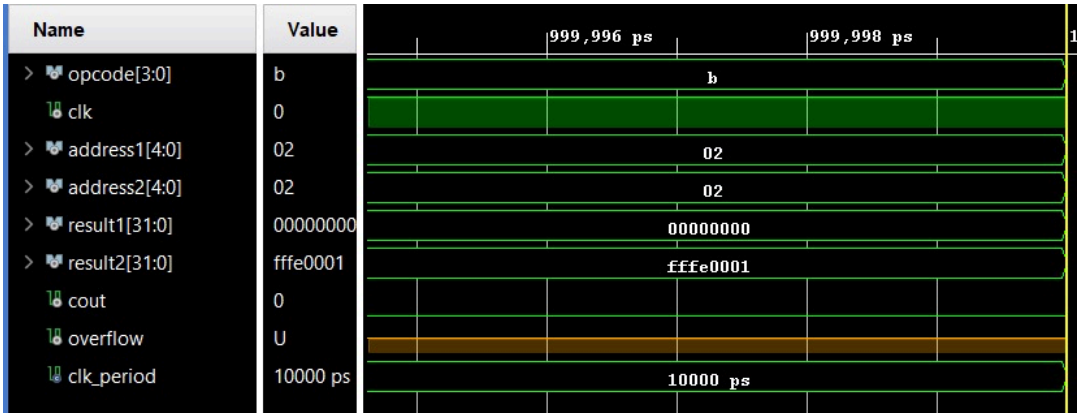

Figure 28 - Test for multiplication

Division

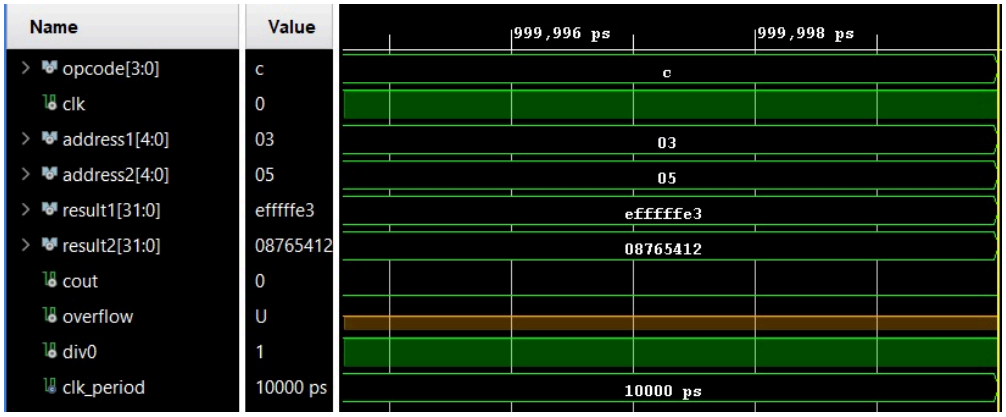

Figure 29 - Test for division by 0

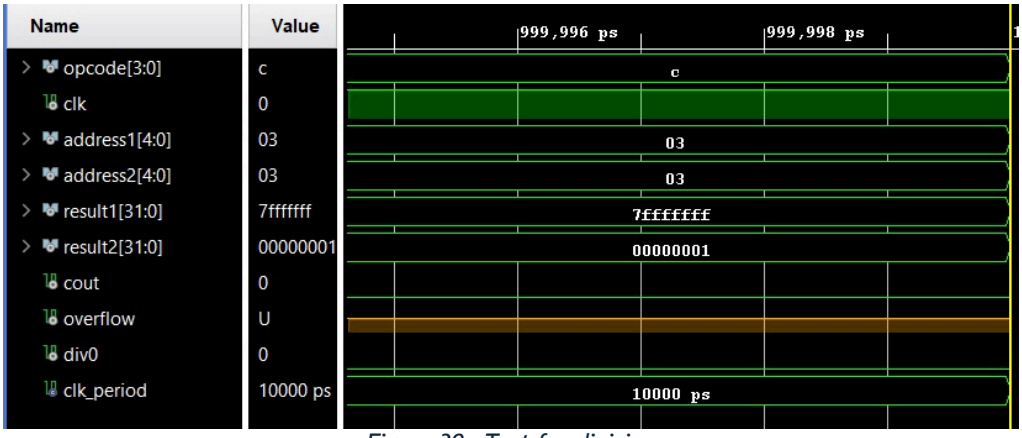

Figure 30 - Test for division

Conclusion

At first glance, designing an ALU for a fully functional system may seem like a straightforward task. However, as the process of analysis, design, and implementation unfolds, the complexity of addressing all possible operations and edge cases becomes apparent. Each function added to the ALU introduces new challenges in integration, control, and verification.

Nevertheless, the objective of creating a versatile and efficient ALU has been successfully achieved. The implemented design supports arithmetic, logical, and shift operations, as well as control signal management, enabling seamless integration into a larger system. This makes the ALU a robust component capable of handling a wide range of computational tasks.

Using a simple FPGA board (like Digilent's Basys3), the ALU has been prototyped and tested in a simulated environment, showcasing its functionality and adaptability. If needed, additional functionality could be incorporated, and the design could be scaled to meet specific system requirements. The flexibility and modularity of the design leave ample room for further innovation and enhancements.

Bibliography

[1] Abuelma’atti M. T., "Design of a Scalable ALU Using VHDL." *International Journal of Electronics* (2003)

[2] McCabe S., "FPGA-Based High-Speed ALU for Real-Time Applications." *IEEE Transactions on Circuits and Systems* (2019)

[3] Choudhary Rajat, "Design of ALU using VHDL and FPGA." *International Journal of Computer Applications* (2014)

[4] Koren, "Computer Arithmetic Algorithms." *Prentice Hall* (2002)

[5] G. Sebestyen & A. Hangan, , "Structure of Computer Systems Lab Documentation." *Technical University of Cluj Napoca* (2024)

[6] Gustavo H. Barrionuevo, , "Booth’s Multiplication Algorithm in VHDL" *Github* (2019)