Arithmetic-Logical Unit (ALU)

Student: Borzan Călina-Annemary

Structure of Computer Systems Project

Technical University of Cluj Napoca

Contents

Introduction 4

1.1 Context 4

1.2 Objectives 4

Bibliographic Research 5

2.1 What is two's complement? 5

2.2 How can the addition and subtraction be done in two's complement? 5

2.3 How can I implement multiplication? 5

2.4 How can I implement division? 7

Analysis 9

3.1 Project Proposal 9

3.2 Project Analysis 9

3.2.1 How to handle overflow, carry out, division by zero detection 9

3.2.2 Use Case and State Diagram 11

Design 13

4.1. Black Box 13

4.2. Schematic 14

4.3. Resources 15

Implementation 21

5.1 Addition and Subtraction Unit 21

5.1.1 1-bit Full Adder 21

5.1.2 Carry Block 21

5.1.3 4-bit Carry Lookahead Adder 22

5.1.4 32-bit Carry Lookahead Adder 22

5.2 Logical Operations Unit 24

5.3 Multiplication Unit 25

5.4 Division Unit 28

5.5 ALU Top Unit.....31
6. Testing and Validation.....35
6.1 32-Bit Carry Lookahead adder.....35
6.2 Multiplication Unit.....35
6.3 Division Unit.....36
6.4 Alu Top Unit.....36
Bibliography 38

Introduction

1.1 Context

The aim of this project is to design and implement an Arithmetic-Logical Unit (ALU) capable of performing multiple arithmetic and logical operations required by modern processors. The ALU is a fundamental component frequently used in systems that support arithmetic, logic, and data manipulation. It is the primary part of the CPU and GPU [1], playing a central role in processing instructions, enabling systems to perform tasks ranging from simple data comparisons to complex mathematical operations.

This project focuses on designing a versatile ALU that can be flexible as to be able to be integrated into more complex processing units, such as microprocessors or custom-designed hardware circuits. The inclusion of the accumulator register is to make the sequential operations with minimal overhead easier.

1.2 Objectives

The unit will be designed using VHDL and included in a Xilinx Vivado Project. The design will be programmed on a FPGA board where, with the help of the debouncer and the seven segment display, we will be able to see the operations performed right in front of us. The ALU is supposed to perform basic arithmetic and logical operations including:

- Addition and subtraction in two's complement (C2)
- Increment and decrement operations
- Bitwise operations (AND, OR, NOT)
- Negation and rotation (left and right)
- Multiplication and division as separate circuits, both using complex algorithms

We will also have the accumulator register that will hold the intermediate results, enabling sequential operations to be performed efficiently. A one-bit full adder will be used tin implementation, along with a multiplexer to select the desired operation. Also, I need to have an error detection to make sure that in the moment we have division by 0 we will get an error on the board and the operation will not be performed. Nonetheless, for addition and subtraction I want to use flags in case overflow occurs, or we have situations in which we need to handle carry out.

Bibliographic Research

2.1 What is two’s complement?

Two's complement (C2) is a widely used method for representing signed integers in binary systems [4]. It simplifies the design of arithmetic circuits in processors, as it allows for a unified approach to addition and subtraction. In C2, the most significant bit (MSB) serves as the sign bit, with 0 indicating positive numbers and 1 indicating negative numbers.

2.2 How can the addition and subtraction be done in two’s complement?

To perform addition in two's complement, the binary values of the numbers are simply added as if they were unsigned integers. If an overflow occurs—meaning the result exceeds the bit-width—or if the carry in most significant bit differs from the carry out it is discarded. This approach allows addition to handle both positive and negative numbers seamlessly [4].

Subtraction is achieved by adding the negative of the number to be subtracted. This is done by taking the two's complement of the number being subtracted, which involves inverting its bits and adding 1 to the least significant bit (LSB). The subtraction can be expressed as: $A-B=A+(-B)$ where $-B$ is calculated using the two's complement method. This consistent method for both operations eliminates the need for separate circuitry for subtraction, enhancing efficiency.

2.3 How can I implement multiplication?

Multiplication can be a tricky operation especially when we want to implement this on 32 bits numbers. One of the easiest and most natural algorithms is the Shift and Add Multiplication[1]. The Shift-and-Add multiplication method is analogous to how multiplication is performed manually using paper and pencil. The basic idea is to multiply a number (the multiplicand) by each bit of the second number (the multiplier) and then shift and add the intermediate results to obtain the final product. Here’s a theoretical breakdown of how it works:

Initialize Registers:

- Register B (multiplicand register) is loaded with the value of the multiplicand (X).
- Register Q (multiplier register) is loaded with the value of the multiplier (Y)
- Register A (accumulator) is initialized to 0.
- Counter N is set to the bit length of the multiplier, n.

Loop Through Multiplier Bits:

- The least significant bit of the multiplier (Q_0) is checked:
- If $Q_0 = 1$, the multiplicand (B) is added to the accumulator (A).
- If $Q_0 = 0$, no addition is performed.

Shift and Update:

- After checking and possibly adding, the combined value in registers B is shifted once to the left and Q is shifted one position to the right. This shift prepares the next bit of the multiplier for processing in the following iteration.
- The counter N is decremented by 1 after each shift.

Repeat:

- Steps 2 and 3 are repeated until N reaches 0, indicating that all bits of the multiplier have been processed.

Result:

- The product of the multiplication operation is stored across registers A and Q.

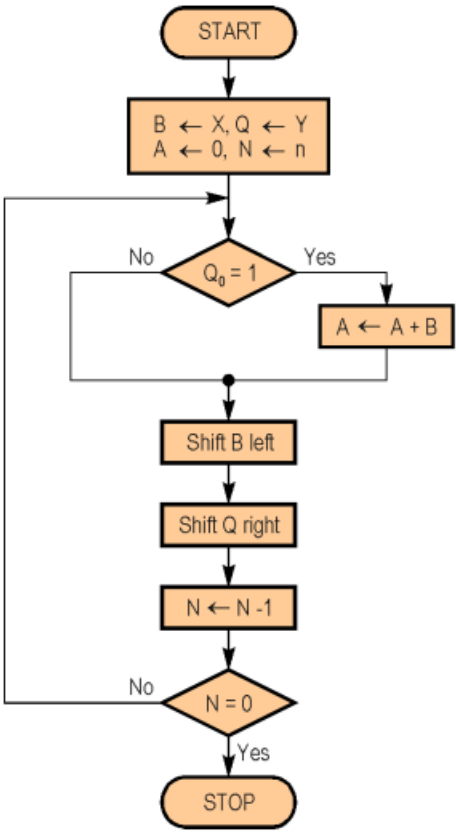


Figure 1 ,,The first version of the multiplication algorithm'' SCS Baruch 2002

2.4 How can I implement division?

The division algorithm chosen for this project is a variation of the Restoring Division Algorithm[1] , applied to 32-bit numbers. This method involves shifting the divisor to the left by powers of 2 and comparing it with the dividend to determine each bit of the quotient [2]. Here we have a theoretical breakthrough:

Initialize Registers:

- Register A (Accumulator) is initialized with 0.
- Register Q (Quotient) is loaded with the dividend (e.g., X).
- Register B (Divisor) is loaded with the divisor (e.g., Y).
- N (Counter) is set to the number of bits in Q

Loop Until Counter N is Zero:

- Shift the combined A_Q register left by one bit.
- Subtract the divisor B from A and store the result in A.
- Check the result of the subtraction:
    - **If A is Negative**: This means the divisor was too large.
        - Restore A by adding B back to it.
        - Set the least significant bit (Q_0) of Q to 0.
    - **If A is Non-negative**: This means the division was successful.
        - Set the least significant bit (Q_0) of Q to 1.

Decrement Counter N:

- If N is not zero, repeat the loop from **Step 1**.
- Otherwise, proceed to the next step.

End of Division:

- When N reaches zero, the contents of the Q register represent the quotient, and the A register holds the remainder.

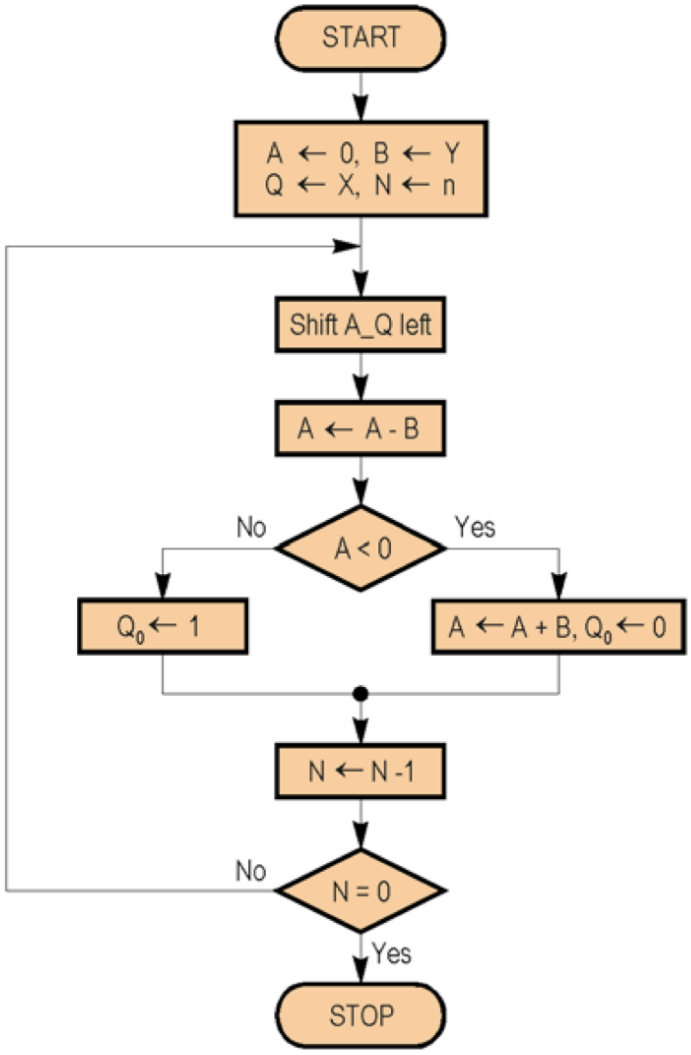


Figure 2 ,,The final version of the division algorithm'' SCS Baruch 2002

Analysis

3.1 Project Proposal

The final Arithmetic Logical Unit will encompass the following features:

- The possibility of introducing some numbers and choosing one of the following operations: ADD, SUBTRACT, MULTIPLY, DIVIDE, ROTATE, NEGATE, OR, AND, NOT and seeing the result on the seven-segment display
- The user will be able to perform successive operations, the previous result being stored in the accumulator register.
- Detection of negative numbers being illustrated on the seven segment with a '-' symbol and the numbers will be represented in Two's Complement.
- Different circuits for the multiplication and division and complex algorithms that will make the implementation easier
- Flags- handle the situations of overflow, carry-out, division by zero
- Tests that show the correction of the operations

3.2 Project Analysis

3.2.1 How to handle overflow, carry out, division by zero detection

3.2.1.1 Overflow

Overflow occurs when the result in an arithmetic operation exceeds the maximum limit that can be represented with the help of the specific number of bits, in this case with 32 bits. Overflow can occur when adding two positives number and the result becomes a negative number, or when adding two negative numbers and the result is a positive number. An example of overflow could be if we try to add 2 numbers in 4 bits: +3 (0011) and +5 (0101). The result will be (1000) which is -0 in signed magnitude. This is invalid because +3+5= +8 which can't be represented by the 4 bits.

Also, overflow when it comes to subtraction happens when the signs of operand differs and the result's signs matches the subtracted operand. When it comes to subtraction we must keep in mind that the negative number will have to be transformed in 2's complement. As example we have +3 (0011) and -5 . We represent +5 in binary (0101) and want to compute in 2's complement. Inver the bits:1010 and add 1 => 1010+1=1011 which will be -5 in two s

complement. Now, if we try to compute +3−(−5), it becomes +3+(+5) because subtracting a negative number is equivalent to adding the positive version of it. And is the same as the previous example where +8 can not be represented on 4 bits.

We use a dedicated output line to signal overflow. If an overflow is detected a led will be started.

3.2.1.2 Carry-out Handling

Carry-out tells us that the limit of 32 bits had exceeded so the result will not be the exact one. We will have for this a flag that will inform us.

3.2.1.3 Division by zero Detection

This is a critical error in arithmetical operations. It occurs when the divisor is zero which makes the operation impossible to perform. This will be checked at the start of the operation and if the divisor is 0 flag in my circuit will become 1 and this will initialize a message on the FPGA the inform the user of the error.

3.2.2 Use Case and State Diagram

3.2.2.1 Block Diagram of Control Unit

I made this block diagram that helps us understand how will the flow of this project be. It represents the first step of the implementation.

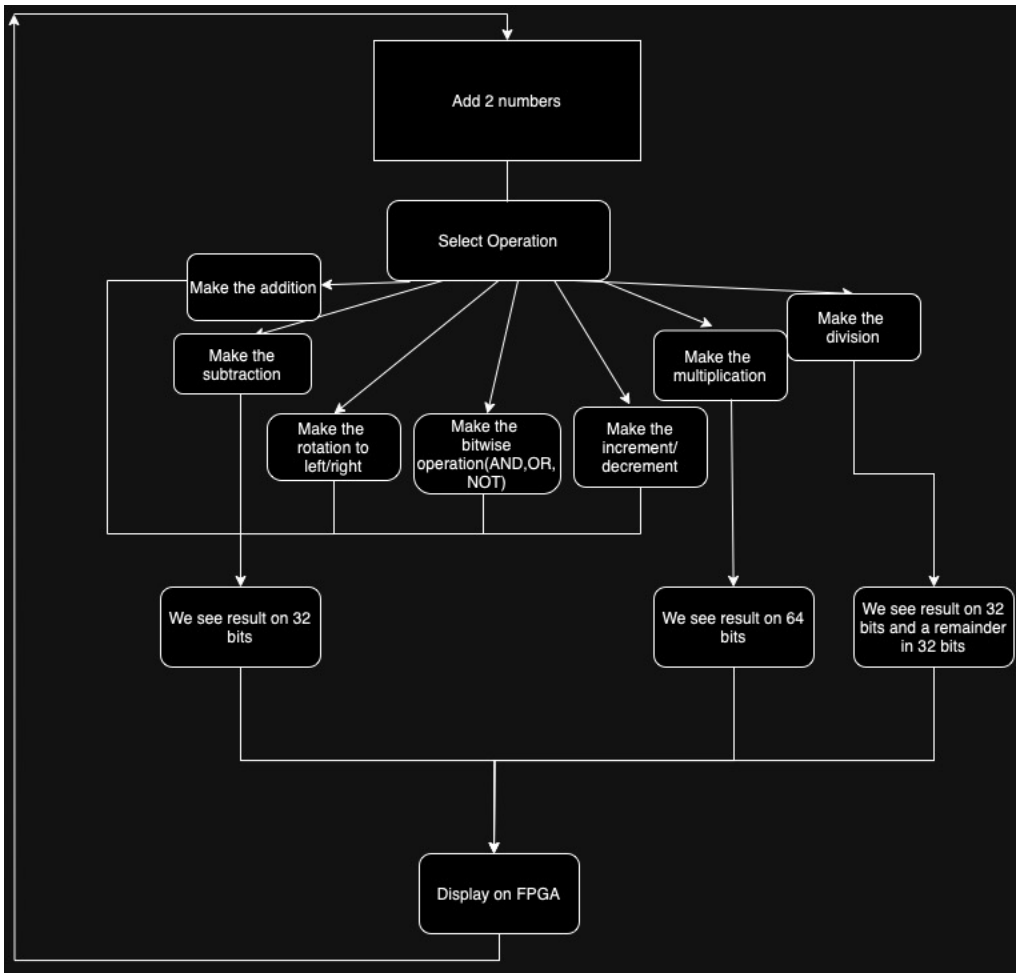


Figure 3 „Block diagram" implemented in draw.io

3.2.2.2 Use case

Actor: User

Precondition:

- Alu is powered and initialized
- The user is ready to input the number and understand what he will see

Basic Flow:

1. **Input Numbers:**

- The user inputs the first number (Operand 1).
- The user inputs the second number (Operand 2).

1. **Select Operation :**

- The user select an operation (addition, subtraction, multiplication, division, negate, increment, decrement, bitwise operations, rotate).

2. **Perform Operation:**

- The system performs the selected operation.
- The result is stored in the accumulator.

3. **Show Result:**

- The system displays the result.

4. **Input Additional Number:**

- The user is prompted to input another number if they want to perform additional operations.

5. **Select Operation Again:**

- The user can select another operation using the accumulator's value as Operand 1 and the new number as operand 2.

6. **Repeat steps 3-6:**

- The user can continue performing operations, chaining them as needed.

Design

4.1. Black Box

This is the black box of a 32 bit Alu. As input we have:

- **Addr1 and Addr2** which are address inputs from the memory unit. They are locations from which the ALU retrieves the first and second operand, and they will be implemented on switches.
- **Op_select** which will be the selection for the operation wanted. The selection of operations will be like this: 0010- Addition, 0011- Subtraction, 0100-Multiplication, 0101- Division, 1001 - AND, 1010 - OR, 1011 -NOT, 1111 - Negation, 0111 - Rotate Left, 0110 - Rotate right, 1100 - Increment, 1101 - Decrement. This will most likely be implemented on switches.
- **Clk** is clock signal for sequential operations and synchronization.
- **Reset** is a signal to clear the accumulator and reset the state, used if we want to start again from the start the operations, this might also be a button.
- **CIN t**he carry-in input, typically used in addition or subtraction operations.
- **Load_acc** is a control signal to determine if the ALU should continue storing the results in the accumulator to continue performing operations.

As outputs we have:

- **Result** which is the final result of the operation.
- **Overflow_flag** a flag that indicates if an arithmetic overflow has occurred.
- **Cout_flag** which indicates if there is a carry out from the most significant bit during an addition or subtraction operation.
- **Div_by_zero_flag** is a flag that is set when a division by zero is attempted. This is a safeguard to handle undefined operations and avoid erroneous results.

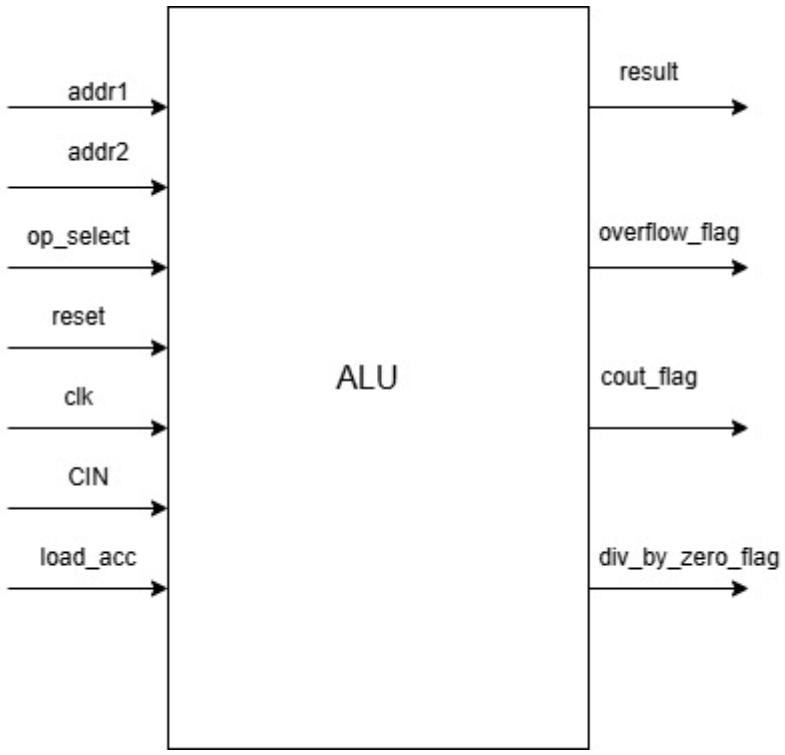


Figure 4 ,,Black Box”

4.2. Schematic

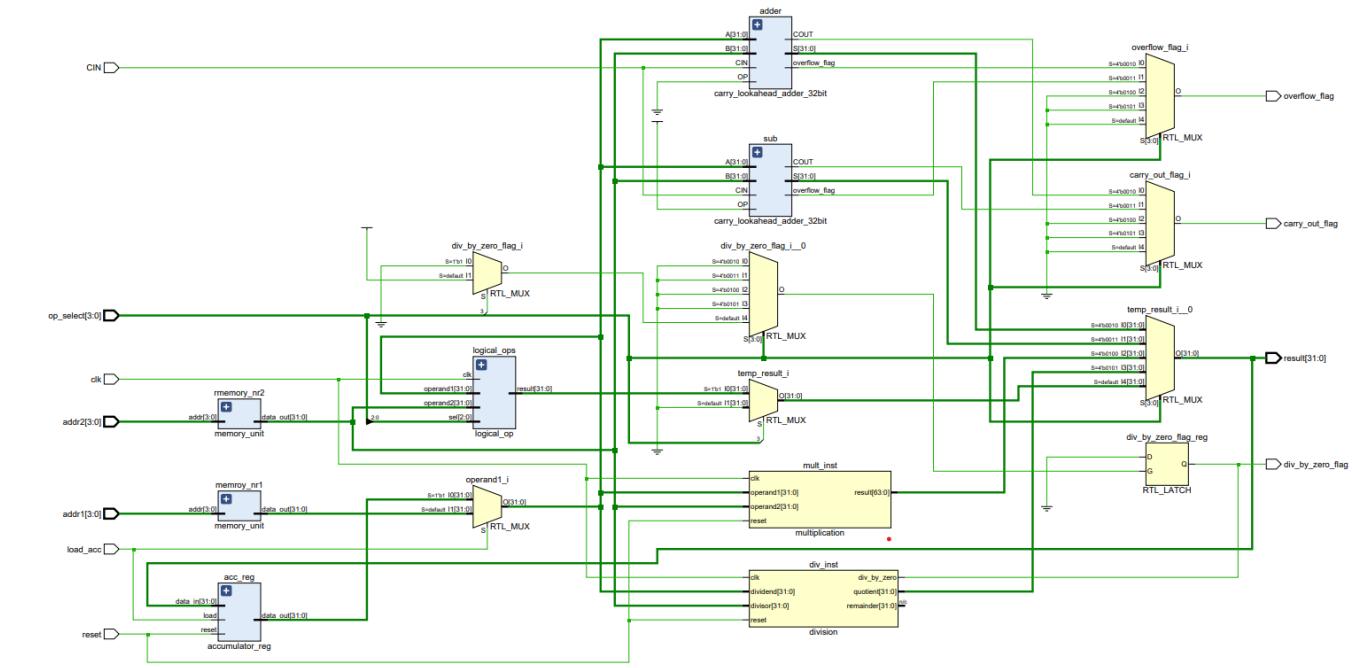
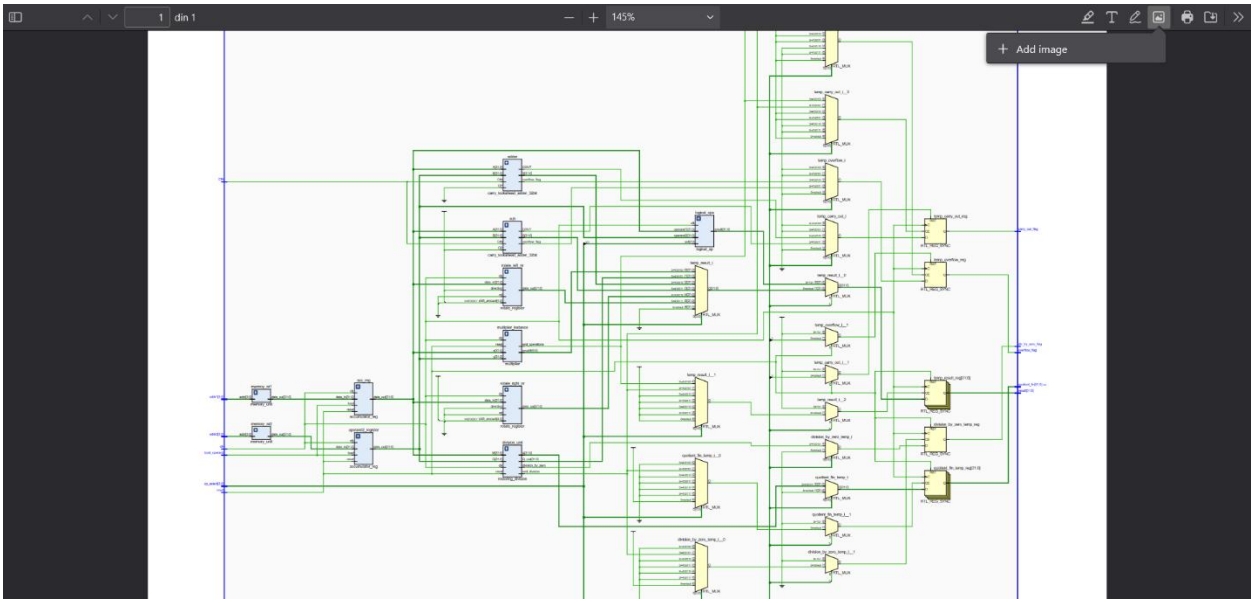


Figure 5 ,,Schematic”



4.3. Resources

In order to establish the design of my ALU and the links between the Control Unit and Execution Unit, we must first identify the resources on the basis of which we make decisions. The EU has the role of executing some instructions to execute mathematical operations. Any decision-making information must come from a resource that generates that information and passes it to the Control Unit. The EU contains the resources that realize the implementation of the operations which we can also see on the schematic. They are:

- **Memory Units** (memroy_nr1 and memory_nr2): These are memory components used to provide the operands (operand1_mem and operand2) based on the addr1 and addr2 inputs. Each memory unit has an address input (addr) and a 32-bit data output (data_out).

- **Accumulator Register** (acc_reg): Stores the result of operations if load_acc is active, allowing the ALU to perform sequential calculations using the previous result as operand1.

- **Full Adder**: This component adds two single-bit binary numbers along with a carry-in bit. It generates a sum bit and a carry-out bit.

  The expressions of the outputs are:

  Si= xi xor yi xor Ci

Ci+1= xi*yi + (xi + yi) *Ci

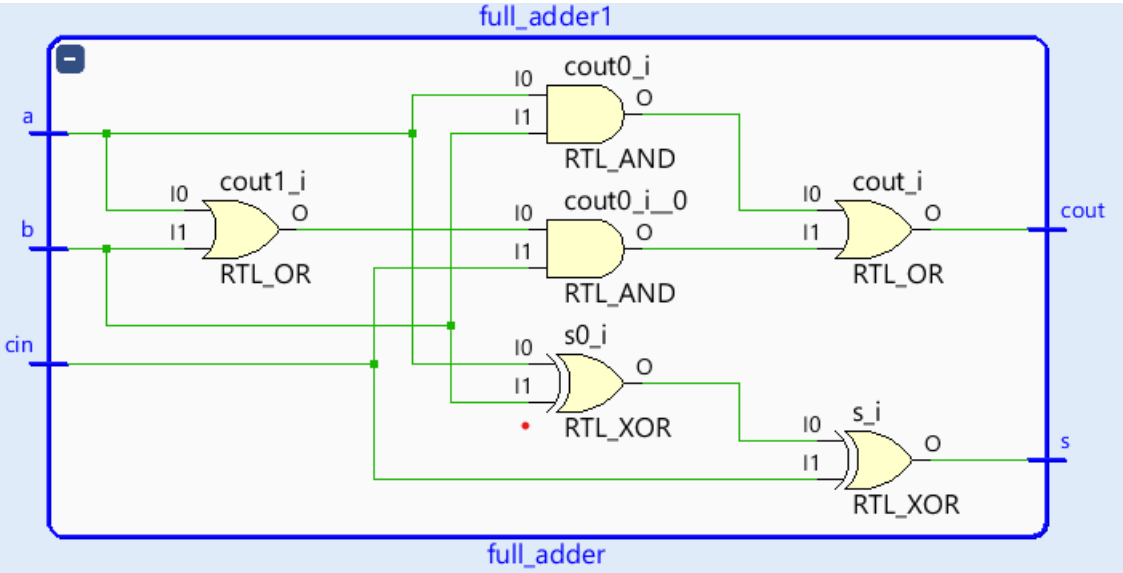


Figure 6 ,, Full Adder ”

- **Arithmetic Components (Adder and Subtractor):** Adder (adder) and Subtractor (sub) components are instances of a carry_lookahead_adder_32bit module. These components take operand1, operand2, and CIN as inputs and produce add_result and sub_result. This is implemented with the help of 8 4 bit Carry Lookahead Adders. A Carry Lookahead Adder is a type of digital adder used for performing binary addition efficiently. Unlike the Ripple Carry Adder, where carry outputs are computed sequentially from one stage to the next, the CLA calculates all carry outputs simultaneously, reducing latency significantly. It is formed on 4 full adders and the 4 carry blocks which is based on these two signals: generate (g) and propagate (p).

  Each bit pair (xi,yi)(xi,yi) produces:
  gi=xi·yigi=xi·yi (Generate)
  pi=xi+yipi=xi+yi (Propagate)

  The carry output for each stage is computed as: Ci+1=gi+pi·CiCi+1=gi+pi·Ci . This expression allows the carry output to be calculated directly based on the current inputs and the previous carry input, eliminating the need to wait for prior carries to be calculated. This adder will be used to create the 32 bit adder/subtractor by cascading 8 of such adders. This component will be a major part of my project because it will be used for addition, subtraction, increment, decrement, negation, multiplying and division.

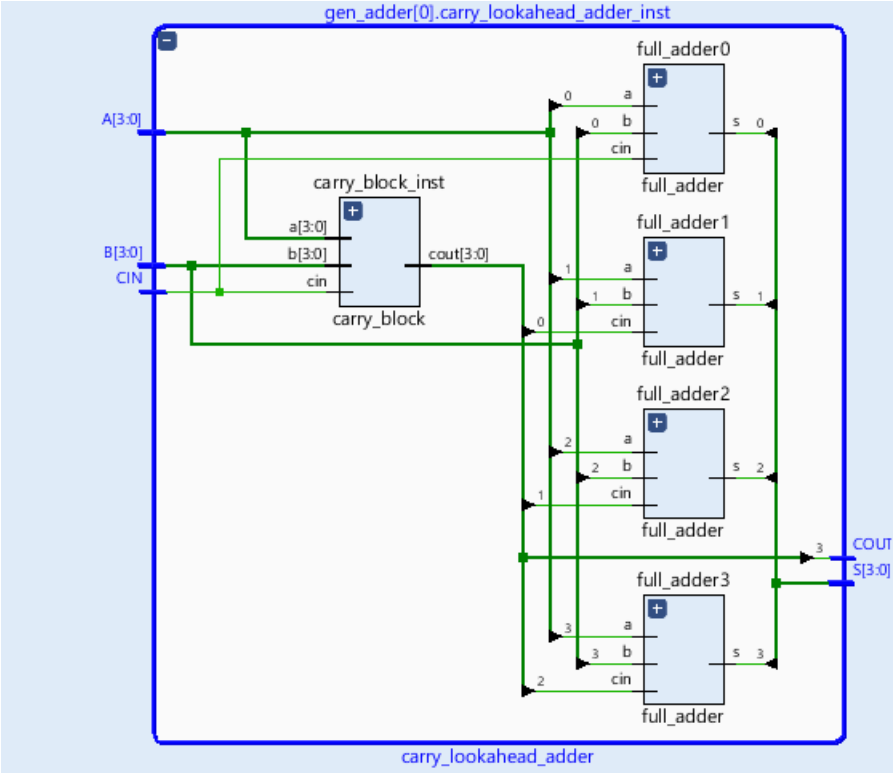


Figure 7 ,, 4 bit Carry Look Ahead Adder ”

- Logical Operations (logical_ops): Handles bitwise operations (AND, OR, NOT) and Increment, Decrement, Negation based on the lower bits of op_select when the MSB is 1. Takes operand1 and operand2 as inputs and produces logical_result as output. It also uses the 32 bit Carry Lookahead adder for the: decrement, increment and negate operation. For the rotate we use a shift register which based on the bit that we trnasmit through direction:1 or 0, we choose the direction. 1 is for left, 0 is for right. We also use the bitwise gates for the AND, OR, NOT.

AND: R= AB OR: R=A + B NOT: R= A’

A	B	R
0	0	0
0	1	0
1	0	0
1	1	1

A	B	R
0	0	0
0	1	1
1	0	1
1	1	1

A	R
0	1
1	0

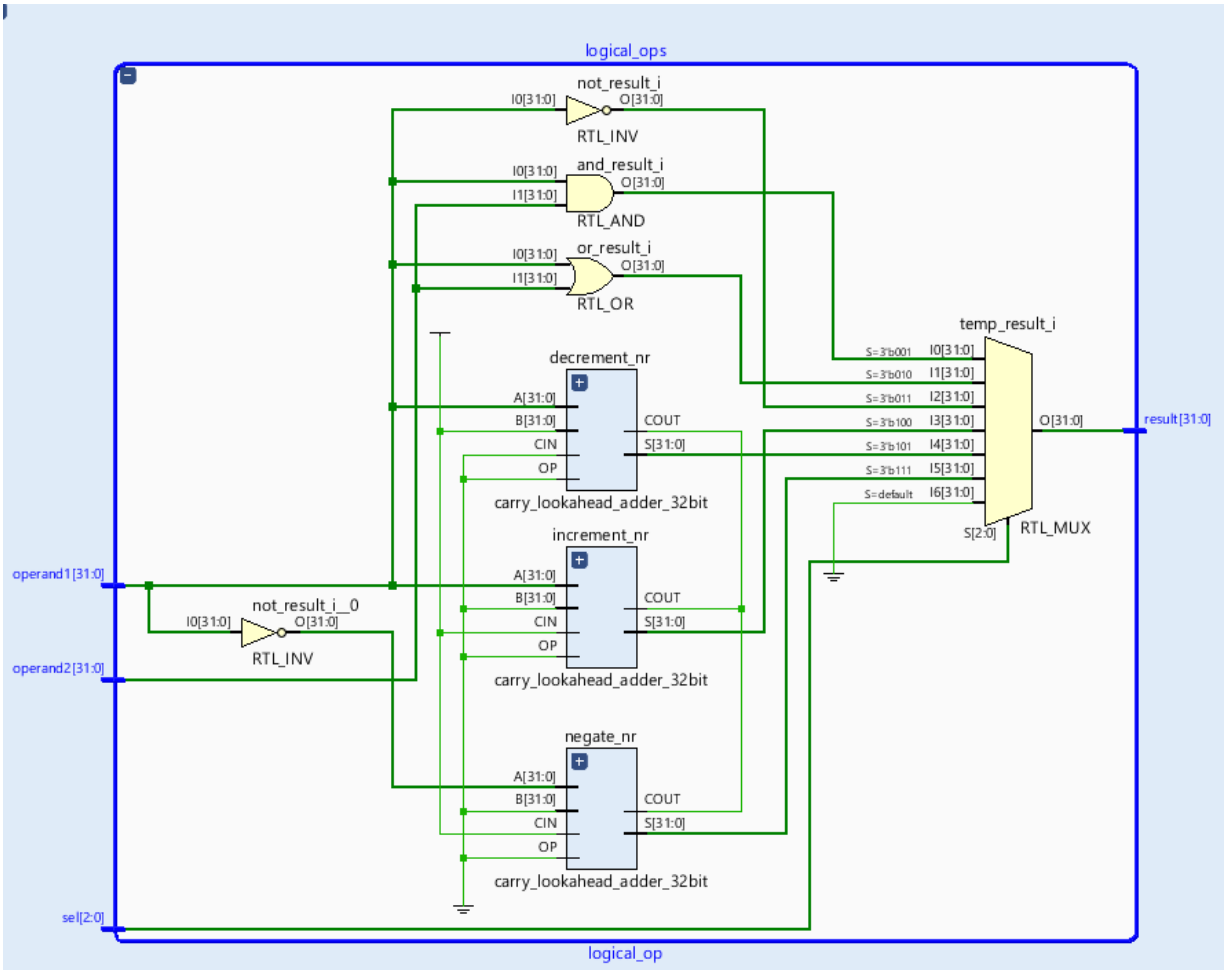


Figure 6 „ Logical Op Unit ”

- Rotate Registers (rotate_right_nr, rotate_left_nr): This component performs the rotation to the left / right of the 32 bit operand.
-

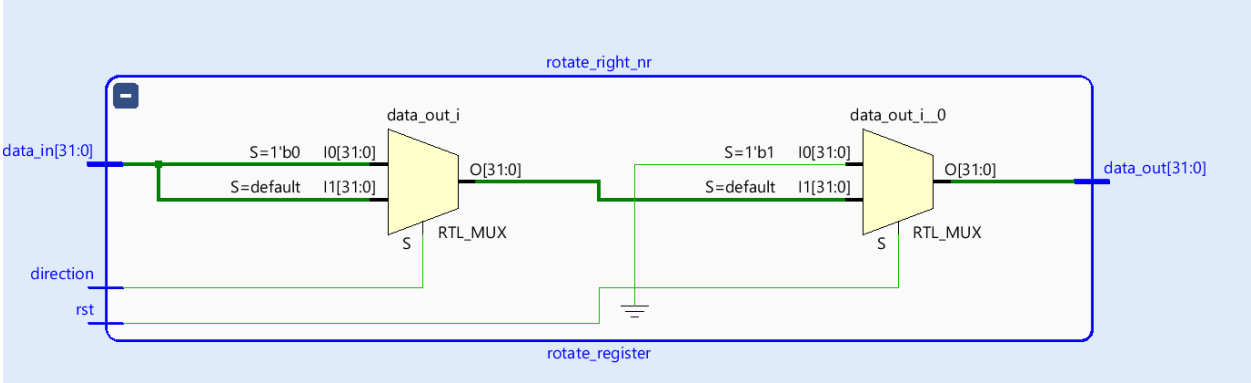


Figure 7 ,, Rotate Register"

- Multiplication (multiplier_instance): The multiplication component performs 32-bit multiplication, producing a 64-bit result (result).

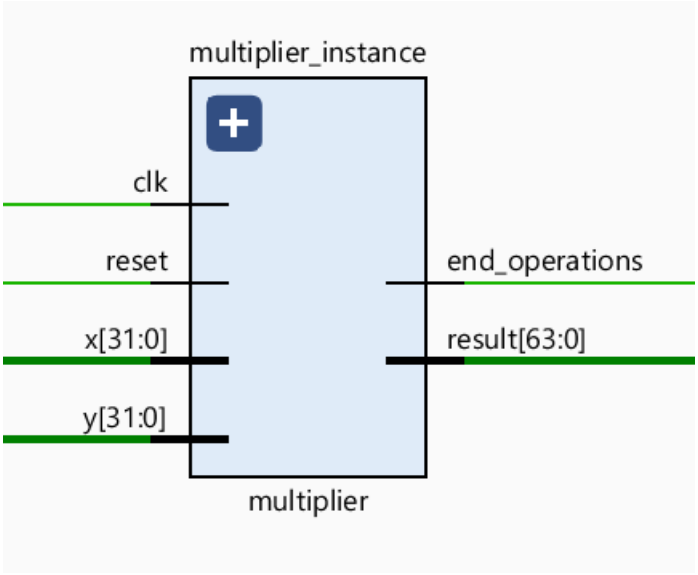


Figure 8 ,,Multiplication circuit)

- Division (division_unit): The division component performs division on operand1 and operand2, producing a 32-bit quotient (Q) and a remainder(A).

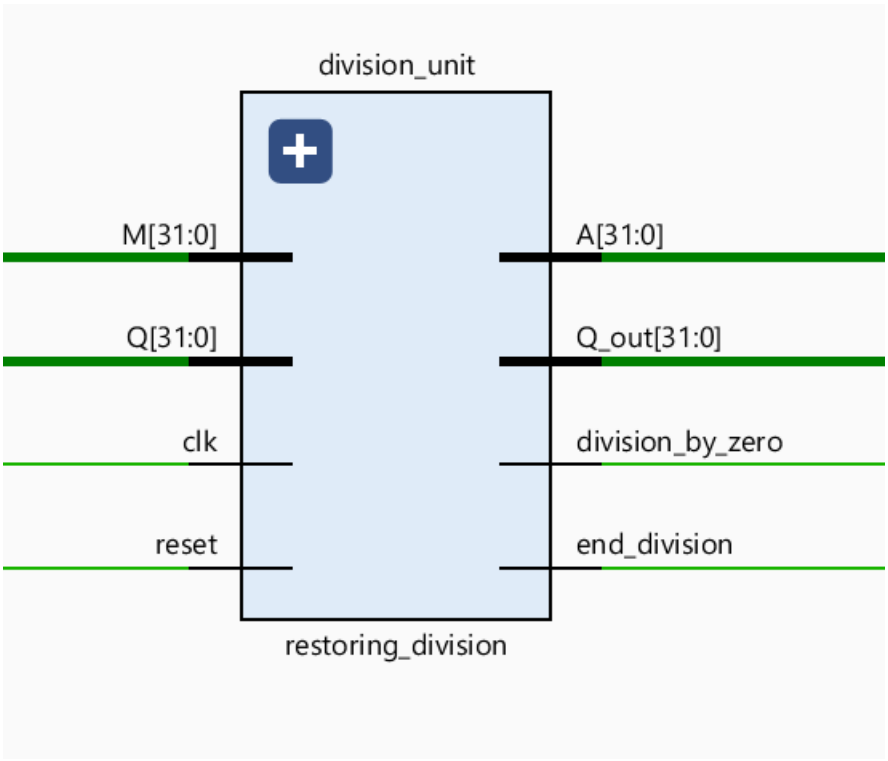


Figure 9 ,,Division Circuit”

- Multiplexers: Are used to select between different results (addition, subtraction, logical, multiplication, and division) based on op_select. A latch (RTL_LATCH) stores the div_by_zero_flag to ensure it is available as output if division by zero is detected.

Implementation

The implementation is the part where we start to put our design and components in the VHDL language with the help of simple processes and multiplexers.

5.1 Addition and Subtraction Unit

We have 2 operations to perform with the help of one big component: the Carry Lookahead Adder. To do this we must first implement the 1-bit full adder, the carry block, the 4 bit Carry Lookahead Adder and then cascade 8 such adders.

5.1.1 1-bit Full Adder

This has as inputs:

- **a** : in std_logic
- **b** : in std_logic
- **cin** : in std_logic

This is how the outputs: **s** and **cout** are computed with the help of the formulas.

```
s <= a xor b xor cin;
cout <= (a and b) or ((a or b) and cin);
```

Figure 10 ,, The computation of outputs "

5.1.2 Carry Block

This has as inputs:

- a : in std_logic_vector(3 downto 0)b :
- b: in std_logic_vector(3 downto 0)
- cin : in std_logic

The carry output for each bit position in a carry lookahead adder is generated by leveraging two key signals, Generate (G) and Propagate (P), which simplify the process of carry calculation and make it faster than in a ripple-carry adder. The carry-out for each bit position, Cout, is calculated by combining the generate and propagate signals with any previous carries. By using these

generate and propagate signals, the carry lookahead adder quickly determines the carry for each position without waiting for sequential propagation, allowing all carries to be calculated almost simultaneously. This approach significantly reduces the delay associated with carry calculation, making the carry lookahead adder much faster than traditional adders in performing multi-bit additions.

```
signal p, g: std_logic_vector(3 downto 0);
signal cout_aux: std_logic_vector(3 downto 0);
begin
g <= a and b;
p <= a or b;
cout_aux(0) <= g(0) or (p(0) and cin);
cout_aux(1) <= g(1) or (p(1) and cout_aux(0));
cout_aux(2) <= g(2) or (p(2) and cout_aux(1));
cout_aux(3) <= g(3) or (p(3) and cout_aux(2));
cout <= cout aux;
```

Figure 11 ,, Use of generate and propagate signal and how the output is generated"

5.1.3 4-bit Carry Lookahead Adder

This has as inputs:

- a: in std_logic
- b: in std_logic
- cin: in std_logic

In here we simply use 4 full adders and the carry block. The carry block calculates the carry in signals for each bit position in advance. Each full adder then uses these pre-calculated carry signals to quickly add the corresponding bits of the two numbers without waiting for the previous bit's carry out to propagate. This way, each bit in the sum is calculated independently. The carry out remains open, we don't use them at all. The final carry-out from the most significant bit is set as the overall carry-out of the entire addition.

5.1.4 32-bit Carry Lookahead Adder

This has as inputs:

- A: in std_logic_vector(31 downto 0)
- B: in std_logic_vector(31 downto 0)
- OP: in std_logic
- CIN: in std_logic

The circuit has an OP signal that decides if the operation will be addition or subtraction. If OP is set to 1, it indicates subtraction, and the code inverts B and adds 1 to create the signal B_modified. If OP is 0, it simply assigns B to B_modified for addition. CIN_temp is an internal carry chain used to connect the carry outputs of each 4-bit adder to the carry inputs of the next one. The initial carry input, CIN, is stored in CIN_temp(0). I use a loop to generate eight instances of the 4-bit carry lookahead adder, connecting them in sequence to handle all 32 bits. Each instance of the 4-bit adder takes a chunk of 4 bits from A and B_modified, as well as the carry from the previous adder (stored in CIN_temp). Each instance outputs a 4-bit sum and the carry to the next instance, like we would do for cascading normal adders. After all 4-bit sections are processed, the combined sum for all 32 bits is stored in S_internal, which is then assigned to S as the final result. The final carry-out, COUT, is taken from the carry output of the last 4-bit adder (CIN_temp(8)). An overflow_flag is set based on the exclusive OR (XOR) of the last two carry signals, CIN_temp(8) and CIN_temp(7). This flag detects overflow by checking if the carry into the most significant bit differs from the carry out, which would indicate a sign overflow in two’s complement arithmetic.

5.2 Logical Operations Unit

There are 6 operations to be performed in this unit: AND, OR, NOT, Decrement, Increment, Negation. There implementation, inputs and meaning of outputs is shown below.

These are the inputs:

- **operand1**: in std_logic_vector(31 downto 0)
- **operand2**: in std_logic_vector(31 downto 0)
- **sel**: in std_logic_vector(2 downto 0) – used for the operation selection
- **clk** :in std_logic
- **start**: in std_logic

We use a Finite State Machine with 3 states:Idle, Execute, Finish. In Idle we check if the operations should have already started, and if it did we move to the next state. In the Finish state we have the selection process, where we store the **result** in the final state, and set ready to '1'.

```
when Finish =>
    case sel is
        when "001" =>  -- AND operation
            result <= and_result;
        when "010" =>  -- OR operation
            result <= or_result;
        when "011" =>  -- NOT operation
            result <= not_result;
        when "100" =>  -- Increment
            result <= incremented;
        when "101" =>  -- Decrement
            result <= decremented;
        when "111" =>  -- Negation
            result <= negate_result;
        when others =>
            result <= (others => '0');
    end case;
    ready <= '1';
    state <= Idle;
```

Figure 12 „ Process for final output"

5.3 Multiplication Unit

This unit implements a Shift-and-Add Multiplication Algorithm for 32-bit signed integers, producing a 64-bit result. It operates over multiple states, systematically handling initialization, computation, and finalization of the multiplication process.

It has as inputs:

- **x** : in std_logic_vector(31 downto 0);
- **y** : in std_logic_vector(31 downto 0);
- **clk** : in std_logic;
- **reset** : in std_logic;
- **start**: in std_logic ;

As output we have the result on 64 bits and a flag(end_operations) that will keep count when each operation is done so we can take the valid result.

The multiplication process is state-driven. Each state corresponds to a step in the multiplication algorithm:

1. **Initialization (initial_state)**

Objective: Check if the operation was started or not.

Steps:

- Set the end_division to 0.
- Check start
- Set the state to check_sign_x.

2. **Check Sign X (check_sign_x)**

Objective: Check the sign of X and use the adder to negate if it is negative.

Steps:

- Check the most significant bit of x
- If is ‘1’ it means it is negative, and we set the adder to negate the number by doing not(x) + 1
- If is ‘0’ it means we have positive number and we keep the value as it is

3. **Check Sign Y (check_sign_y)**

Objective: Check the sign of Y and use the adder to negate if it is negative.

Steps:

- Check the most significant bit of y
- If is ‘1’ it means it is negative, and we set the adder to negate the number by doing not(y) + 1
- If is ‘0’ it means we have positive number and we keep the value as it is

3. Setup multiplier (setup_multiplier)

Objective: Prepare the registers and signals for multiplication.

Steps:

- Set the accumulator (A) to 0.
- Load the absolute values of x and y into B and Q, respectively.
- Initialize the counter (N) to the bit-width of the multiplier (32).
- Set end_operation to ‘0’.
- Set the state to add_shift.

4. Add and Shift (add_shift)

Objective: Perform addition and prepare for the next iteration.

Steps:

- Check the least significant bit of Q (Q(0)):
- If Q(0) = 1, add B to A using the carry_lookahead_adder_32bit modules, two of them to add the first 32 bits of the A and B and the last 32 bits of A and B. This will be done in 2 states: add_lower, add_upper.
- If Q(0) = 0, skip the addition step.
- Transition to the shift_B state.

5. Add upper and add lower states

Objective: Perform addition.

Steps:

- Prepare the adder to add the first 32 bits of A and B and then the last 32 bits of A and B.
- Transition to the shift_B state.

6. Shift B (shift_B)

Objective: Shift the multiplicand (B) left by one bit.

Steps:

- Perform a left shift on B.
- Transition to the shift_Q state.

7. Shift Q (shift_Q)

Objective: Shift the multiplier (Q) right by one bit.

Steps:

- Perform a right shift on Q.
- Transition to the decrement_counter state.

8. Decrement Counter (decrement_counter)

Objective: Reduce the iteration count and check for completion.

Steps:

- Subtract 1 from the counter (N) using the carry_lookahead_adder_32bit module (subtractor_N).

- We go to the decrement_N_start state.

9. Decrement N (decrement_N_start)

Objective: Verify the value of the counter.

Steps:

- Check if N is ‘0’
- If is ‘0’ go to the final state
- If is not go back to add_shift state

10. Finalize Result (finalize_result)

Objective: Compute the final result and handle the sign of the product.

Steps:

- Check the XOR of the sign bits of x and y to determine the sign of the result (result_sign).
- If the result is negative, negate the contents of A using the adder32bits_negateA_upper and adder32bits_negateA_lower modules in 2 states: negate_64bit_lower_start, negate_64bit_upper_start.
- Store the final result in the result signal.
- Transition to the delay_state.

11. Delay State (delay_state)

Objective: Check if N is 0

Steps:

- Ensure proper timing by introducing a delay before resetting the multiplier.
- Reset end_operations to 1 and transition to done_state.

12. Done state

Objective: Reset the operation

Steps:

- If start is ‘0’ go back to initial state

5.4 Division Unit

The division unit implements a Restoring Division Algorithm for 32-bit signed integers, yielding a 32-bit quotient (Q) and a 32-bit remainder (A).

The algorithm is state-based, ensuring sequential execution of division steps with intermediate results managed effectively.

It has as inputs:

- **Q** : in std_logic_vector(31 downto 0);

- **M**: in std_logic_vector(31 downto 0);
- **clk** : in std_logic;
- **reset** : in std_logic;

As outputs it will have the quotient Q_out on 32 bits, the remainder A on 32 bits, the end_division flag to be sure we get the correct result and division_by_zero flag with which we check that we don't have division by 0.

1. **Initialization (initial_state)**

Objective: Reset all flags and signals and check if the operation should start.

Steps:

- Initialize to 0 all the flags.
- Check is start is 1 and if it is check M to see if is negative or positive.
- If M is negative prepare the adder to negate it
- If M is positive store the value and go to the next state negate_Q

2. **Check zero**

Objective: Check if we have division by zero.

Steps:

- Check if M is '0'
- If it is set the flag to '1'.
- If it is not we set the A register to '0' and the Q register and the counter.

3. **Shift Left Part (iteration_shift)**

Objective: Shift the A_Q register left by one bit.

Steps:

- Perform a left shift on A and Q (similar to multiplying by 2).
- Transition to the subtract_M state.

4. **Subtraction (subtract_M)**

Objective: Subtract the divisor from the accumulator.

Steps:

- Compute A = A - M using the carry_lookahead_adder_32bit module (subtractor).
- Transition to check_remainder

5. **Check remainder(check_remainder_sign)**

Objective: Check is we will need to restore A

Steps:

- If the result is negative (A(31) = 1), proceed to the restore_A state.
- Otherwise, set Q to 1 and transition to the decrement_N_setup.

6. **Restore Accumulator (restore_A)**

Objective: Undo subtraction if the divisor is larger than the current value of A.

Steps:

- Add the divisor back to A using the carry_lookahead_adder_32bit module (adder).
- We go to a wait state after this in which we capture the result passed by the adder and set Q to 0.
- Transition to the the decrement_N_setup.

7. **Decrement Counter (**decrement_N_setup**)**

Objective: Reduce the iteration counter and check for completion.

Steps:

- Subtract 1 from the counter (N) using the carry_lookahead_adder_32bit module (subtractor_N).
- Transition to final_check.

8. **Final check state (final_check)**

Objective: Here we check the flag for division by ‘0’ and result sign.

Steps:

- We check the flag for division by ‘0’ and if it is ‘1’ we transition to delay_state.
- If not we check the sign for the final result

- If the final result should be negative we start to negate the result and A with the help of adder.

9. Delay State (delay_state)

Objective: Introduce a delay to reset and prepare for subsequent operations.

Steps:

- Wait for the counter to reset.
- Transition to done_state for the next operation.

10. Done state (done_state)

Objective: Reset the process for next operation

Steps:

- Check is start is ‘0’ and if it is we transition to initial_state.
- If not we remain in this done state

5.5 ALU Top Unit

Alu Top Unit represent the main component of the ALU, the top level component, in which all the components are combined to implement the complete functionality of an ALU.

It has as inputs:

- **clk** : in std_logic
- **reset** : in std_logic
- **addr1** : in std_logic_vector(3 downto 0)
- **addr2** : in std_logic_vector(3 downto 0)
- **op_select** : in std_logic_vector(3 downto 0)
- **CIN :** in std_logic
- **load_acc** : in std_logic
- **start**: in std_logic

The operation type is chosen using op_select, a 4-bit signal that determines which arithmetic or logical function to perform. CIN is an initial carry-in signal, used in addition or subtraction. Load_acc is a control signal that determines whether the result should be stored in the accumulator register for use in subsequent operations. A memory unit fetches the values of operands from specific addresses (addr1 and addr2). In this memory unit we have 16 numbers: negative and positive, represented in two's complement.

We have 2 signals: operand1_mem and operand2, that store the values chosen from memory with the help of addr1 and addr2. We use a multiplexer to choose the value of operand1: if load_acc is active, the value stored in the accumulator is used for operand1 instead of loading directly from memory. This feature allows sequential operations to use the previous result without resetting. Then the output of this multiplexer is transmitted to the components for operations.

The carry_lookahead_adder_32bit component is used twice, once to handle addition and once for subtraction:

- For addition, OP is set to 0, and the result is stored in add_result.
- For subtraction, OP is set to 1, and the result is stored in sub_result

The logical_op component has the result stored in logical_result, and no carry-out or overflow flags are needed for logical operations since they do not produce overflow.

First accumulator register stores intermediate results, allowing sequential operations to be performed without reloading operands from memory. Second accumulator register will store the upper part of the multiplication result or the remainder of the previous division to be used in the next operations.

A process block selects the appropriate operation based on op_select:

- "0010" selects addition.
- "0011" selects subtraction.
- "0110" selects rotate right.
- "0111" selects rotate left.
- "0100" select multiplication.
- "0101" selects division.
- If the highest bit in op_select is 1, a logical operation is chosen

The chosen result is stored in temp_result, which feeds into the accumulator and becomes the final output in result. The final output flags (carry_out_flag, overflow_flag, and div_by_zero_flag) are set based on the selected operation's results, helping to indicate errors or special conditions like overflow or division by zero.

Finite State Machine (FSM) Overview

The FSM in the ALU_top_unit governs the execution flow, ensuring structured and error-free operation. It transitions through several states to load operands, execute the operation, and finalize results:

1. **Idle:**
    - Waits for the start signal.
    - Transitions to Load_Registers (for new operations) or Load_Operands (for sequential operations) based on the load_acc signal.
2. **Load_Registers:**
    - Fetches operands from memory (addr1 and addr2) and loads them into operand1_mem and operand2_mem.
3. **Load_Operands:**
    - Prepares operands for sequential operations:
        - If the previous operation was multiplication, loads the upper 32 bits of the result into operand2.
        - If the previous operation was division, loads the remainder into operand2.
        - Otherwise, fetches operand2_mem from memory.
4. **Operation_Start_Loaded:**
    - Finalizes operand loading and transitions to the operation execution stage.
5. **Operation_Start:**
    - Determines the operation type from op_select and triggers the appropriate submodule:
        - Addition and subtraction transition to Execute_Operation.
        - Multiplication and division transition to Wait_For_Done.
        - Logical and rotational operations transition to Wait_For_Result.
6. **Execute_Operation:**
    - Executes addition or subtraction using the carry_lookahead_adder_32bit component.
    - Captures the result in temp_result and updates temp_carry_out and temp_overflow.

7. **Wait_For_Done:**
    - Monitors the completion of multiplication or division operations:
        - Captures the lower 32 bits of the multiplication result.
        - Captures the quotient and remainder for division.
        - Flags division-by-zero errors if applicable.
8. **Wait_For_Result:**
    - Waits for the result of logical or rotational operations.
    - Captures the output in temp_result.
9. **Finalize:**
    - Stores the result in the accumulator if load_acc is enabled, allowing sequential operations.
    - Updates flags (carry_out_flag, overflow_flag, and div_by_zero_flag) and sets the final output in result.
10. **Error_State:**
    - Handles division-by-zero errors by zeroing the result and setting the div_by_zero_flag.

6. Testing and Validation

6.1 32-Bit Carry Lookahead adder

Here we see the waveform for this component in which I have tested every type of operation: addition with positive numbers, addition with negative numbers, subtraction with positive numbers and subtraction with negative numbers. Everything works completely fine.

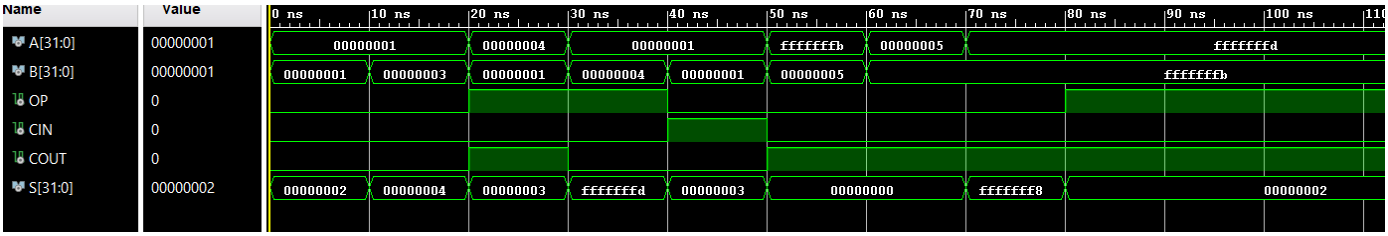


Figure 13 ,, Waveform of 32-bit Lookahead Adder doing all types of operations”

6.2 Multiplication Unit

Here we see the waveform for this component in which we can clearly see all the possible tests: multiplication of 2 positive numbers, of one positive and one negative and of 2 negative numbers.

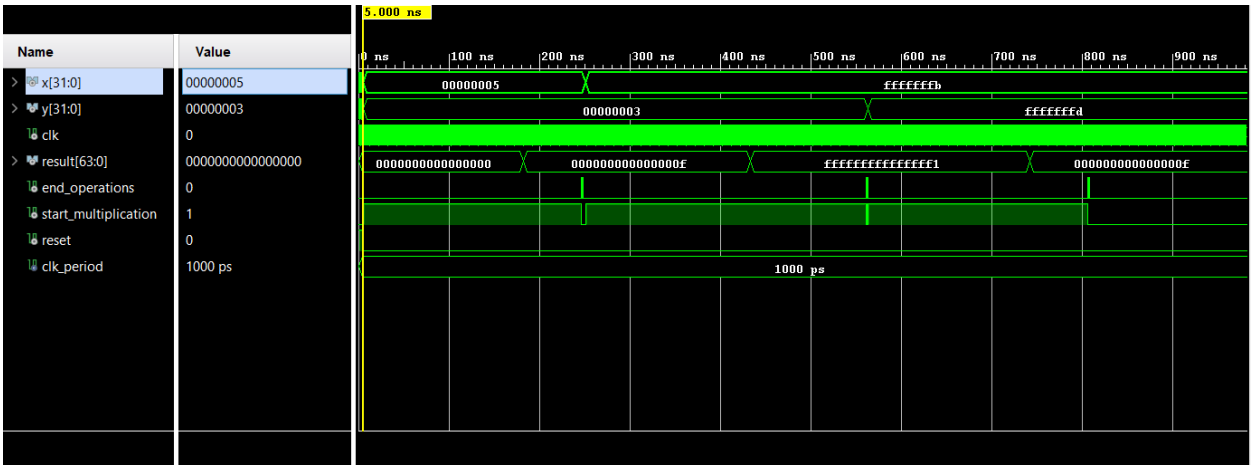


Figure 14 ,,Multiplication Unit doing with all types of operands”

6.3 Division Unit

Here we also have the test bench for the division unit. We tested 4 cases, when we have division by zero, and we can see that the division_by_zero flag becomes 1, when we have the division of 2 positive numbers, of one positive and one negative, and of two negative numbers. Everything works according to how it should do.

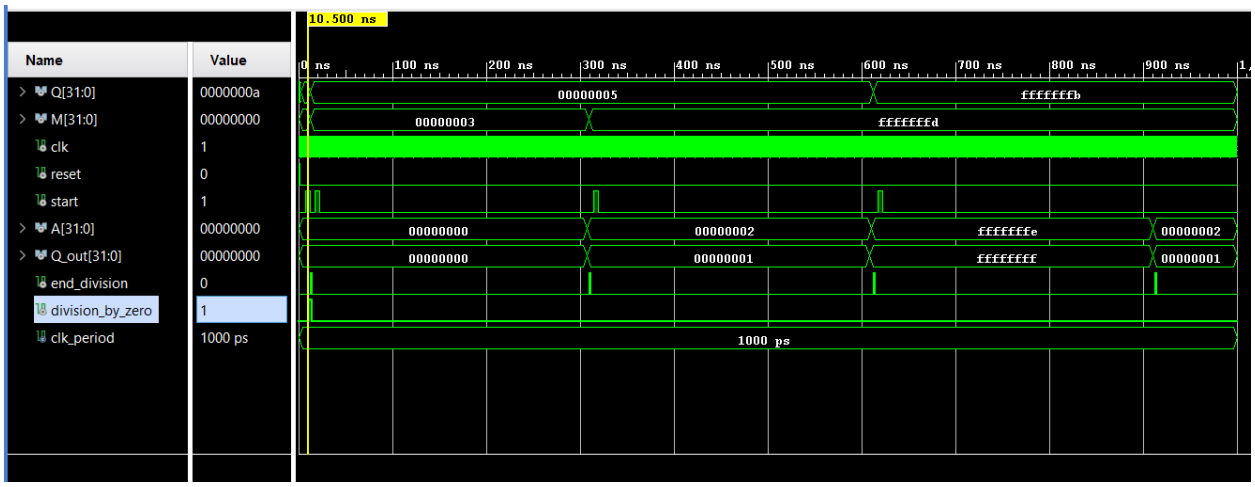


Figure 15 „ Division Unit doing with all types of operands”

6.4 Alu Top Unit

In here I have also tested all the types of operations.

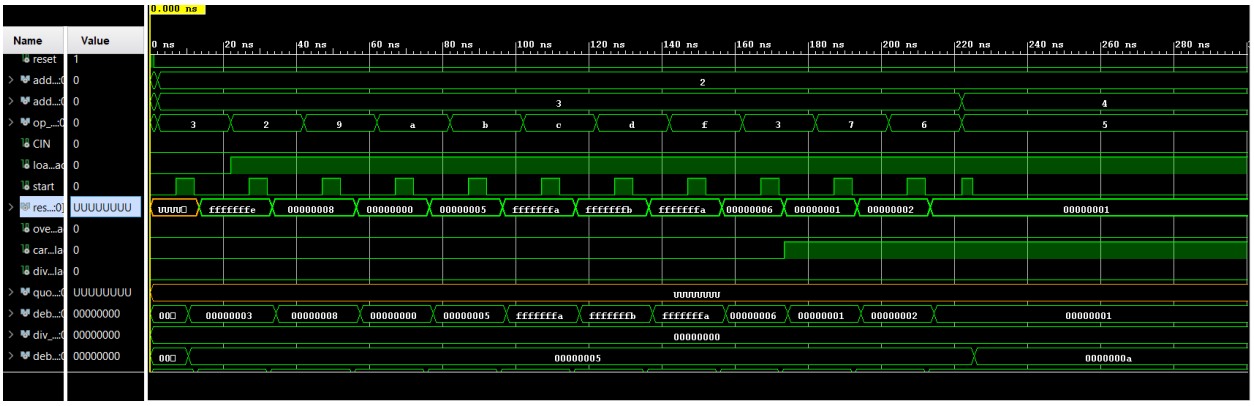


Figure 16 „ Alu Top Unit doing with all types of operands”

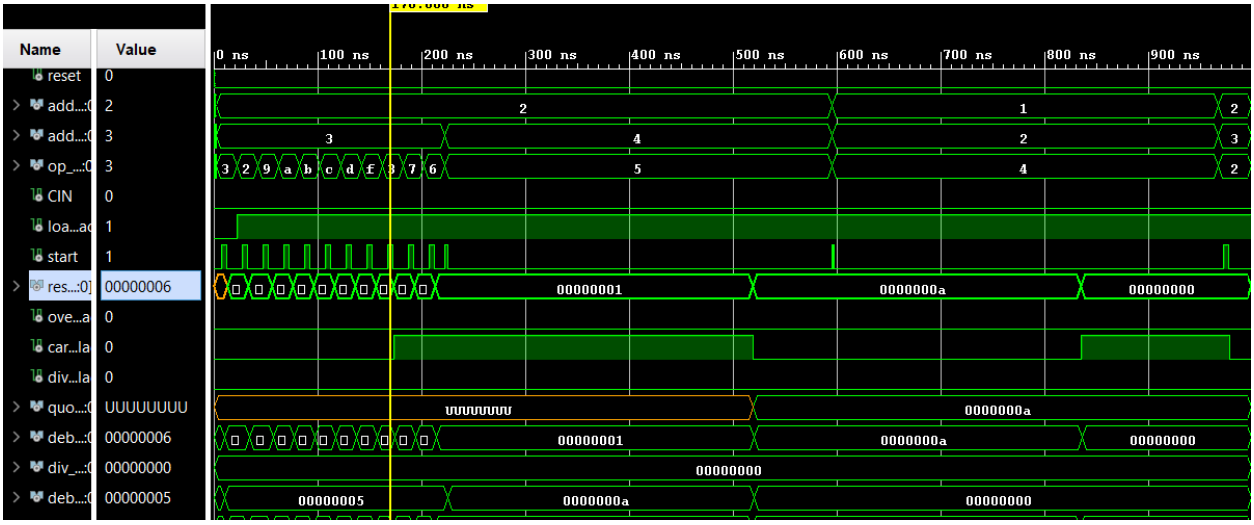


Figure 17 „ Alu Top Unit doing with all types of operands”

Bibliography

[1] *Baruch Zoltan Francisc, ,, Structure of Computer Systems" 2002*

[2] *Laboratory 4 ,,Design of ALU component" Structure of Computer Systems 2024*

[3] *Arithmetic logic unit :https://en.wikipedia.org/wiki/Arithmetic_ logic_unit*

[4] *Octavian Cret, Lucia Vacariu, Aurel Netin ,,Logic Design" 2003*