CIE 239: Digital design and computer architecture project

PREPARED FOR:
Dr Amr hefny

PREPARED BY:

Yara mahmoud 202300177
s-yara.abdalrazek@zewailcity.edu.eg
Hagar alsherbiny 202301124
s-hagar.atallah@zewailcity.edu.eg
Matthew nader 202300103
s-matthew.matta@zewailcity.edu.eg

Date of submission: 10, August 2024.

CONTENTS

COVER PAGE	[1]
CONTENTS	[2]
EXECUTIVE SUMMARY	[3]
INTRODUCTION	[3-4]
THE DESIGN STEPS AND WAVE OUTPUTS	[4-16]
CODE	[16-20]
The schematics	[20-24]
CONCLUSION AND RECOMMENDATIONS	[25]

EXECUTIVE SUMMARY

This project involves designing a 3-bit Arithmetic and Logic Unit (ALU) that performs eight operations: 2's complement, addition, subtraction, increment, XOR, NOT, OR, and AND. The ALU outputs a 3-bit result (F) and two flags: overflow (V) and zero (Z).

Challenges and Solutions:

Handling overflow was challenging. Initially, we tried comparing F to 3'b111 and using a 4-bit F, both of which proved problematic. The final solution involved using a 4-bit temporary variable (temp) to perform calculations and set F as the rightmost 3 bits of temp, with V set based on the fourth bit.

Operations Implementation:

2's complement, addition, subtraction, and increment operations include overflow checks.
XOR, NOT, OR, and AND operations are straightforward without overflow concerns.
Flag Handling:

- V is set to 1 if the result exceeds 3 bits.
- Z is set to 1 if F equals 3'b000.

This design effectively addresses the initial challenges, ensuring accurate operation and flag handling.

INTRODUCTION

An Arithmetic and Logic Unit (ALU) is a fundamental building block of a computer's central processing unit (CPU). It performs arithmetic and logical operations on binary data. The ALU takes in data from the CPU's registers, processes it according to the specified operation, and returns the result back to the registers. This project focuses on designing a 3-bit ALU capable of performing eight distinct operations, essential for basic computational tasks.

The 3-bit ALU in this project is designed to handle operations on 3-bit inputs, providing outputs that include a 3-bit result (F) and two status flags: the overflow flag (V) and the zero flag (Z). These flags are crucial for indicating special conditions resulting from the operations, such as overflow and zero results. The specific operations implemented in this ALU include two's complement, addition, subtraction, increment, XOR, NOT, OR, and AND.

The goal of this project is to create a robust and efficient ALU that can accurately perform these operations and handle overflow and zero conditions effectively.
including:

1. 000: Negate A and add 1 (2's complement)
2. 001: Add A and B
3. 002: Subtract A from B
4. 003: Increment A
5. 004: XOR between A and B

6. 005: NOT of B
7. 006: OR between A and B
8. 007: AND between A and B

By addressing the challenges encountered during the design process, this project demonstrates a thorough understanding of digital logic design principles and the practical application of Verilog for hardware description. The design and implementation of a 3-bit Arithmetic and Logic Unit (ALU) have been successfully completed.

THE DESIGN STEPS AND WAVE OUTPUTS:

We have 8 operations we made binary code for each operation

- 000: Negate A and add 1 (2's complement).
- 001: Add A and B.
- 002: Subtract A from B.
- 003: Increment A.
- 004: XOR between A and B.
- 005: NOT of B.
- 006: OR between A and B.
- 007: AND between A and B.

And also we have flags

- V: Overflow flag.
- Z: Zero flag (set when the output F is zero).

For the inputs we have A , B and op"operation" and for the output we have F , V and Z .

For all inputs and outputs we have 3 bits. [2:0]

```
module ALU (
   input  logic [2:0] A,
   input  logic [2:0] B,
   input  logic [2:0] Op,
   output reg [2:0] F,
   output logic V,
   output logic Z
);
```

problems faced while performing the code:

In the overflow part we were unable to decide which approach to implement:

1-we used inequalities to compare if F>3’b111 then set V to be equal 1 **But** when the value overflow 3 bits it resets F to be 000 because it’s decided in code to be 3 bits

2-so we made F to be logic [3:0] instead of logic [2:0]But F was displayed as 4-bit number and it is required to be 3-bit number

3-we tried to save F in some variable called x and then check if x is greater than 111 But this why was too long to deal with

4- we used a simple way where we have some variable temp that is 4-bit number

```
logic [3:0] temp;
```

Then In the structural code, the ALU module contains submodules for each operation. Each submodule is responsible for its specific operation and handles overflow (V) as needed. The zero flag (Z) is calculated based on the final output F.

5-the method mentioned in [4] was really effective in two’s complement and complement but in addition for example if you are adding 2 positive number the result might be 3 bit but a negative number so that is counted as overflow, in order to consider these we have changed the addition and subtraction like that:

1. in addition :

   When adding two binary numbers, an overflow occurs if the result exceeds the maximum value that can be represented with the given number of bits. For example, in a 3-bit system, the maximum positive value is 011 (which is 3 in decimal), and the minimum negative value (in 2's complement) is 100 (-4 in decimal). Overflow occurs in the following cases:

   - Positive Overflow: When adding two positive numbers results in a negative number. This happens if both A[2] and B[2] are 0 (indicating both numbers are positive) and the most significant bit (MSB) of the result F[2] is 1 (indicating a negative result).
   - Negative Overflow: When adding two negative numbers results in a positive number. This happens if both A[2] and B[2] are 1 (indicating both numbers are negative) and the MSB of the result F[2] is 0 (indicating a positive result).

   Breaking Down the Expression:

   A[2]: The most significant bit (MSB) of input A. This bit determines the sign of A (1 for negative, 0 for positive in 2's complement).

B[2]: The MSB of input B. This bit determines the sign of B.

F[2]: The MSB of the result F. This bit indicates the sign of the result.

The expression assign V = (A[2] & B[2] & ~F[2]) | (~A[2] & ~B[2] & F[2]); detects overflow by checking the following conditions:

(A[2] & B[2] & ~F[2]): This part detects a negative overflow. It checks if both A and B are negative (A[2] & B[2] are both 1), but the result F is positive (F[2] is 0).

(~A[2] & ~B[2] & F[2]): This part detects a positive overflow. It checks if both A and B are positive (A[2] & B[2] are both 0), but the result F is negative (F[2] is 1).

Combining the Conditions:

The | (OR) operation combines these two overflow detection conditions.

If either condition is true, the overflow flag V is set to 1, indicating that an overflow has occurred.

If neither condition is true, V is 0, indicating no overflow.

This expression effectively checks whether the result of adding two 3-bit signed numbers has overflowed beyond the representable range, setting the overflow flag accordingly.

2. In subtraction:

   The expression assign V = (B[2] ^ A[2]) & (B[2] ^ F[2]); is used to detect overflow in a 3-bit subtraction operation by comparing the signs of the operands and the result.

   Breaking Down the Expression:

   B[2] ^ A[2]:

   XOR (^) Operation: The XOR operation compares the most significant bits (MSBs) of B and A.

   If B[2] and A[2] are different (one is 1 and the other is 0), the XOR result will be 1.

   This part of the expression checks if B and A have opposite signs. Overflow in subtraction can occur when the signs of the two operands are different.

   B[2] ^ F[2]:

   This part checks if the sign of the result F[2] is different from the sign of B[2].

   If B[2] and F[2] are different, the XOR result will be 1.

This is important because an overflow occurs in subtraction if the sign of the result is opposite to the sign of B.

Combining the Two Conditions with AND (&):

The AND operation combines the two conditions.

Overflow occurs in subtraction if:

The signs of A and B are different.

The sign of the result F is different from the sign of B.

Example of Overflow:

Suppose A is positive and B is negative. Subtracting a large negative number from a positive number could produce a result that's too large to be represented as a positive number in 3 bits, causing overflow.

In this case, B[2] ^ A[2] would be 1, and if the result's sign changes, B[2] ^ F[2] would also be 1, setting V to 1.

Example of No Overflow:

If both A and B are positive, or both are negative, and the result doesn't change the expected sign, V will remain 0, indicating no overflow.

The expression detects whether the subtraction of two 3-bit numbers results in a sign change that indicates overflow. If the signs of A and B are different and the result has an unexpected sign change, it sets the overflow flag V to 1.

```
module ALU (
  input  logic [2:0] A,
  input  logic [2:0] B,
  input  logic [2:0] Op,
  output logic [2:0] F,
  output logic V,
  output logic Z
);
```

```
logic [2:0] F_add, F_sub, F_2scmp, F_inc, F_xor, F_cmp, F_or, F_and;
logic V_add, V_sub, V_2scmp, V_inc;
```

Internal Wires: These wires connect the outputs of the submodules to the main ALU module.
F_add, F_sub, F_2scmp, F_inc, F_xor, F_cmp, F_or, F_and are 3-bit wires for the results of each operation.

V_add, V_sub, V_2scmp, V_inc;, are single-bit wires for the overflow flags of operations that can cause overflow.

```
assign V = (Op == 3'b000) ? V_2scmp :
        (Op == 3'b001) ? V_add :
        (Op == 3'b010) ? V_sub :
        (Op == 3'b011) ? V_inc :
        1'b0;
```

Overflow Flag Assignment:
The V output is assigned based on the operation code (Op).
For Op = 000 (2's complement), V is set to V_2scmp.
For Op = 001 (addition), V is set to V_add.
For Op = 010 (subtraction), V is set to V_sub.
For Op = 011 (increment), V is set to V_inc.
For other operations, overflow is not relevant, so V is set to 0.

```
assign F = (Op == 3'b000) ? F_2scmp :
        (Op == 3'b001) ? F_add :
        (Op == 3'b010) ? F_sub :
        (Op == 3'b011) ? F_inc :
        (Op == 3'b100) ? F_xor :
        (Op == 3'b101) ? F_cmp :
        (Op == 3'b110) ? F_or :
        (Op == 3'b111) ? F_and :
        3'b000;
```

Result Assignment:

- The F output is determined by the operation code (Op).
- The corresponding submodule's output is selected and assigned to F.
- If an invalid operation code is provided, F is set to 3'b000.

```
assign Z = (F == 3'b000) ? 1'b1 : 1'b0;
```

Zero Flag Assignment:
The zero flag Z is set to 1 if the output F is zero (3'b000), indicating that the result is zero.
Otherwise, Z is set to 0.

```
TwosCmp twos_cmp (
  .A(A),
  .F(F_2scmp),
  .V(V_2scmp)
);
```

2's Complement Submodule (TwosCmp):
This submodule calculates the 2's complement of A and outputs the result to F_2scmp and V_2scmp.

```
 Add add (
  .A(A),
  .B(B),
  .F(F_add),
  .V(V_add)
```

```
);
```

Addition Submodule (Add):

This instantiates the Add submodule.

The inputs A and B are connected to the submodule, and the results are output to F_add and V_add.

```
    Sub sub (
    .A(A),
    .B(B),
    .F(F_sub),
    .V(V_sub)
);
```

Subtraction Submodule (Sub):

Similar to the addition submodule, the subtraction submodule computes B - A and outputs the result to F_sub and V_sub.

```
Inc inc (
    .A(A),
    .F(F_inc),
    .V(V_inc)
);
```

Increment Submodule (Inc):

This submodule increments A by 1 and outputs the result to F_inc and V_inc.

```
XOR xor_op (
    .A(A),
    .B(B),
    .F(F_xor)
);
```

XOR Submodule (XOR):

This submodule computes the bitwise XOR between A and B and outputs the result to F_xor.

No overflow flag is needed for logical operations.

```
Cmp cmp (
    .B(B),
    .F(F_cmp)
);
```

NOT Submodule (Cmp):

This submodule computes the bitwise NOT of B and outputs the result to F_cmp.

```
OR or_op (
    .A(A),
    .B(B),
    .F(F_or)
);
```

OR Submodule (OR):

This submodule computes the bitwise OR between A and B and outputs the result to F_or.

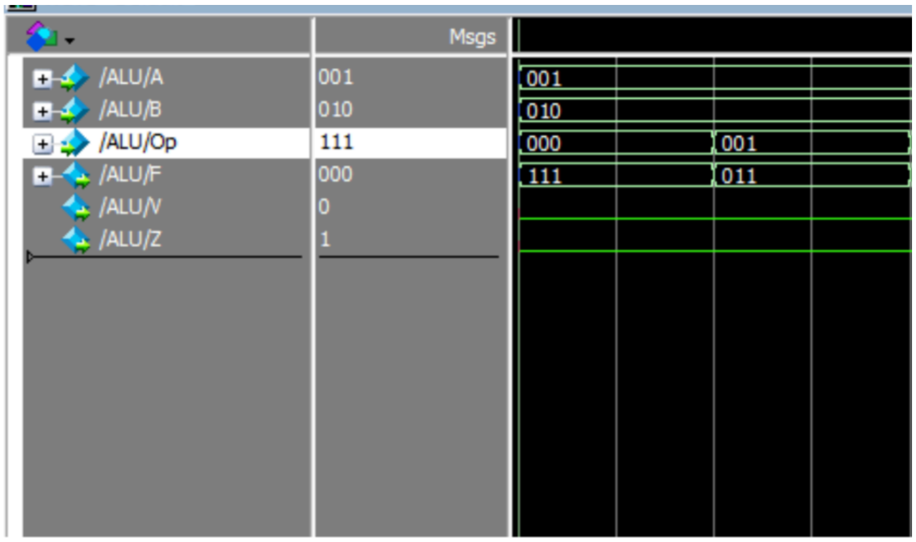
```
 AND and_op (
    .A(A),
    .B(B),
    .F(F_and)
);
```

AND Submodule (AND):
This submodule computes the bitwise AND between A and B and outputs the result to F_and.
In operation 4,5,6,7 there is no need to put the if condition of the overflow since it will never exceed 3 bits since A and B are both 3 bits and the operations won't add extra bits or something.

```
module Add (
    input  logic [2:0] A,
    input  logic [2:0] B,
    output logic [2:0] F,
    output logic V
);
    logic [3:0] temp;
    assign temp = {1'b0, A} + {1'b0, B};
    assign F = temp[2:0];
    assign V = (A[2] & B[2] & ~F[2]) | (~A[2] & ~B[2] & F[2]);
endmodule
```

The expression assign V = (A[2] & B[2] & ~F[2]) | (~A[2] & ~B[2] & F[2]); detects overflow in a 3-bit addition. Checks: Negative Overflow: When adding two negative numbers results in a positive number.
Positive Overflow: When adding two positive numbers results in a negative number.
If either condition is true, V is set to 1, indicating an overflow.

Wave execution example:



```
        Msgs
/ALU/A  001    001
/ALU/B  010    010
/ALU/Op 111    000        001
/ALU/F  000    111        011
/ALU/V  0
/ALU/Z  1
```

Fig (1).

```
module Sub (
    input  logic [2:0] A,
    input  logic [2:0] B,
    output logic [2:0] F,
    output logic V
);
    logic [3:0] temp;
    assign temp = {1'b0, B} - {1'b0, A};
    assign F = temp[2:0];
    assign V = (B[2] ^ A[2]) & (B[2] ^ F[2]);
endmodule
```

The expression assign V = (B[2] ^ A[2]) & (B[2] ^ F[2]); detects overflow in a 3-bit subtraction operation.
Here's how it works:
B[2] ^ A[2]: Checks if the signs of B and A are different. Overflow can occur if one is positive and the other is negative.
B[2] ^ F[2]: Checks if the sign of the result F is different from the sign of B. Overflow occurs if the result's sign is not what was expected based on B's sign.
Combining with &: The & (AND) operation ensures both conditions must be true for overflow to be detected:
The signs of A and B are different.
The sign of the result F is different from B.
If both conditions are true, V is set to 1, indicating an overflow has occurred.
Wave execution example:

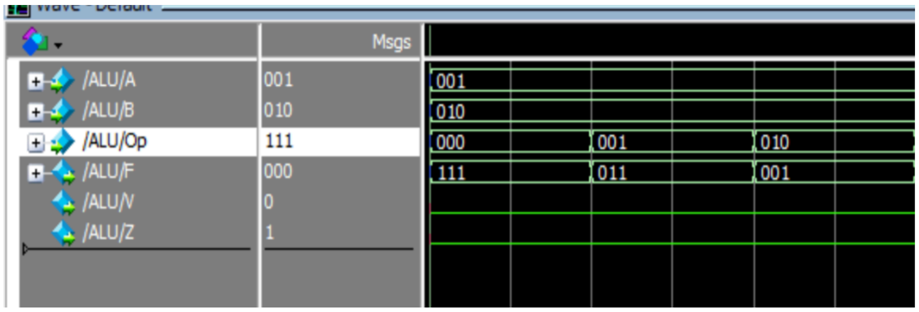
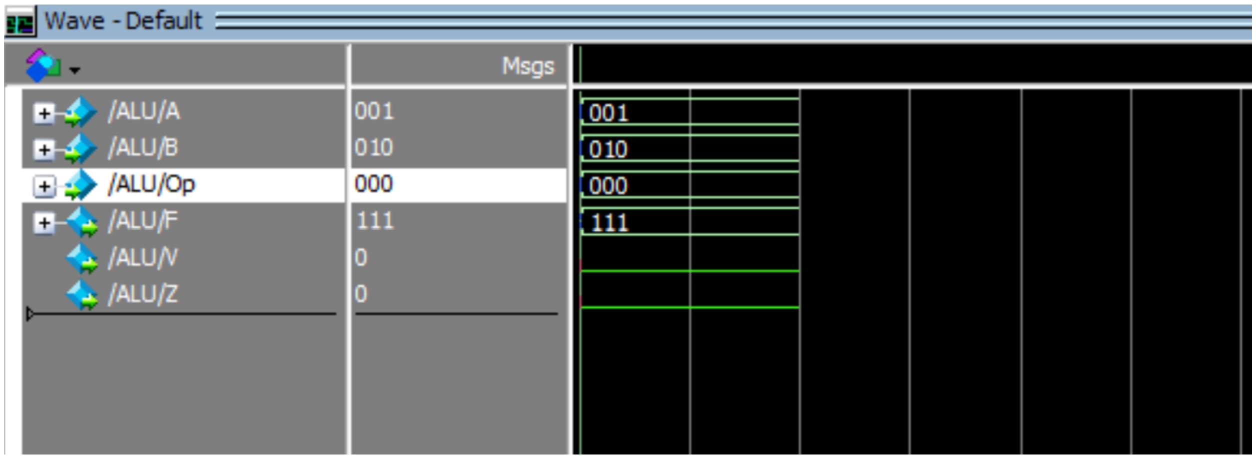


Fig (2).

```
module TwosCmp (
    input  logic [2:0] A,
    output logic [2:0] F,
    output logic V
);
```

The lower 3 bits of temp are assigned to F.
The 4th bit of temp indicates overflow and is assigned to V.
Wave execution example:

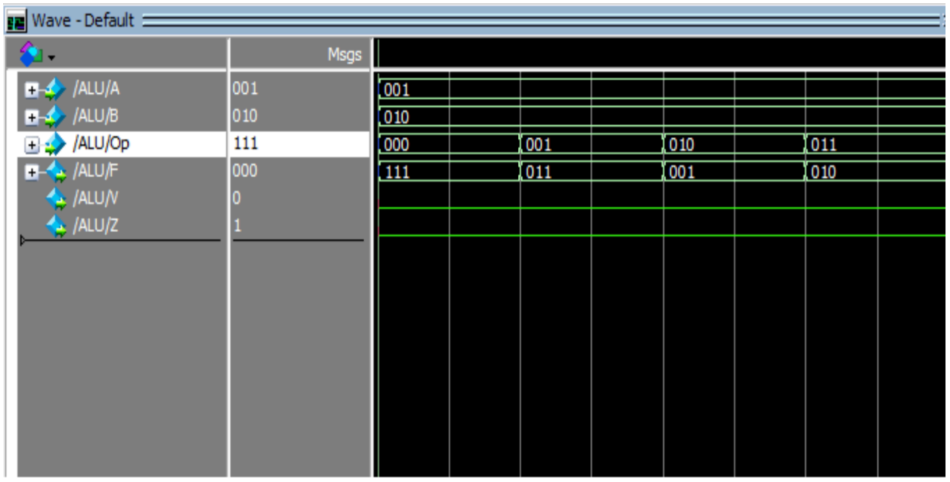


```
/ALU/A    001
/ALU/B    010
/ALU/Op   000
/ALU/F    111
/ALU/V    0
/ALU/Z    0
```

Fig (3).

```
logic [3:0] temp;
   assign temp = {1'b0, ~A} + 4'b0001;
   assign F = temp[2:0];
   assign V = temp[3];
endmodule

module Inc (
   input  logic [2:0] A,
   output logic [2:0] F,
   output logic V
);
   logic [3:0] temp;
   assign temp = {1'b0, A} + 4'b0001;
   assign F = temp[2:0];
   assign V = temp[3];
endmodule
```

The lower 3 bits of temp are assigned to F.
The 4th bit of temp indicates overflow and is assigned to V.
Wave execution example:

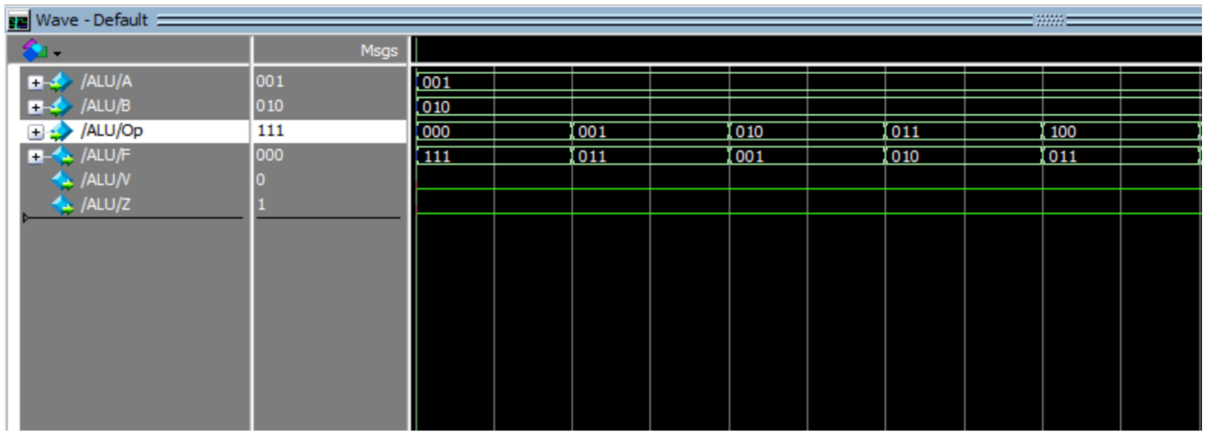


```
/ALU/A    001    001
/ALU/B    010    010
/ALU/Op   111    000  001  010  011
/ALU/F    000    111  011  001  010
/ALU/V    0
/ALU/Z    1
```

Fig(4)

```
module XOR (
   input  logic [2:0] A,
   input  logic [2:0] B,
   output logic [2:0] F
);
   assign F = A ^ B;
Endmodule
```

Wave execution example:

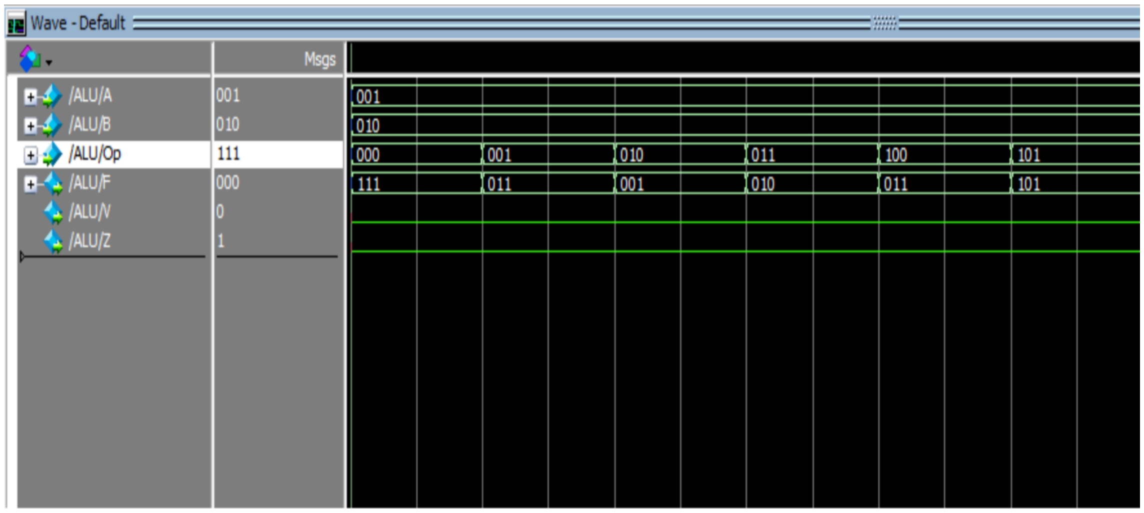


```
/ALU/A    001    001
/ALU/B    010    010
/ALU/Op   111    000  001  010  011  100
/ALU/F    000    111  011  001  010  011
/ALU/V    0
/ALU/Z    1
```

Fig(5)

```
module Cmp (
   input  logic [2:0] B,
   output logic [2:0] F
);
   assign F = ~B;
endmodule
```

Wave execution example:

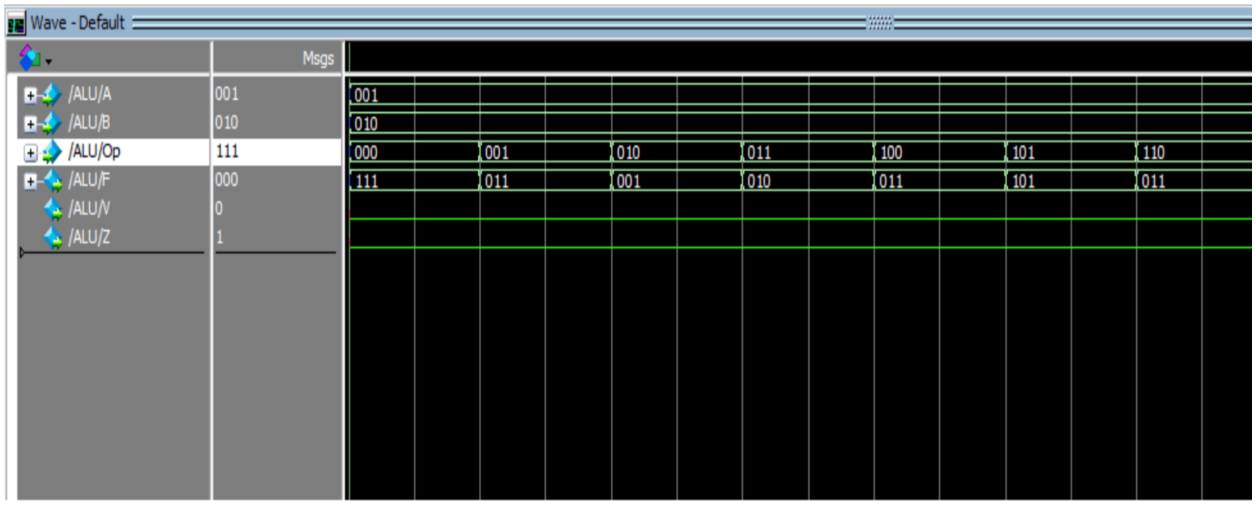


```
/ALU/A   001   001
/ALU/B   010   010
/ALU/Op  111   000  001  010  011  100  101
/ALU/F   000   111  011  001  010  011  101
/ALU/V   0
/ALU/Z   1
```

Fig(6)

```
module OR (
   input  logic [2:0] A,
   input  logic [2:0] B,
   output logic [2:0] F
);
   assign F = A | B;
endmodule
```

Wave execution example:



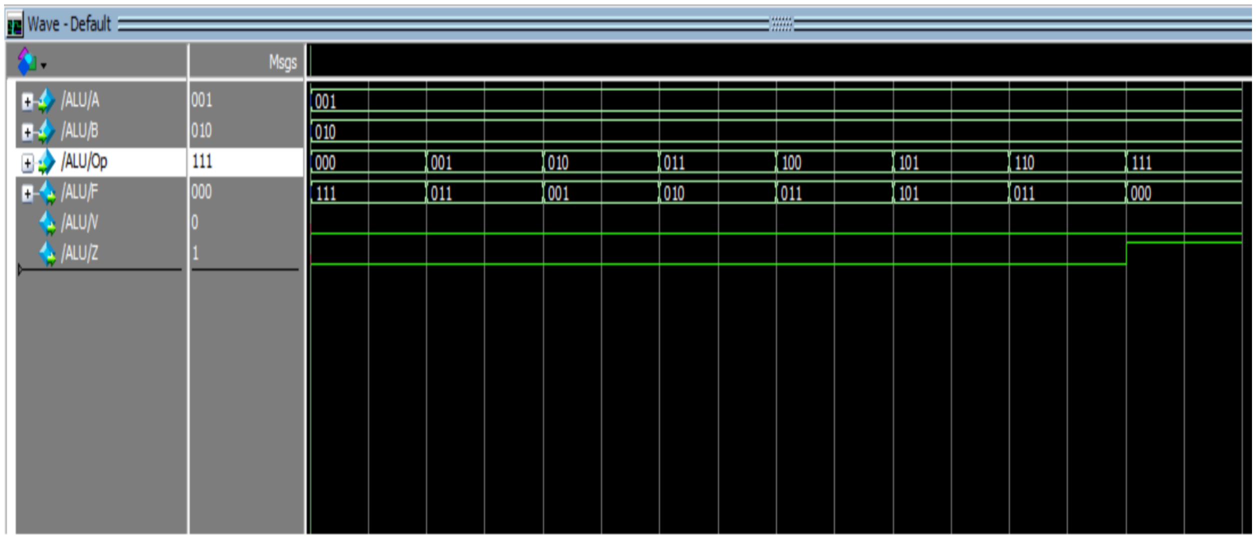
```
/ALU/A   001   001
/ALU/B   010   010
/ALU/Op  111   000  001  010  011  100  101  110
/ALU/F   000   111  011  001  010  011  101  011
/ALU/V   0
/ALU/Z   1
```

Fig(7)

```
module AND (
   input  logic [2:0] A,
   input  logic [2:0] B,
   output logic [2:0] F
);
   assign F = A & B;
Endmodule
```

Note that the value was 000 so Z became true “1”.

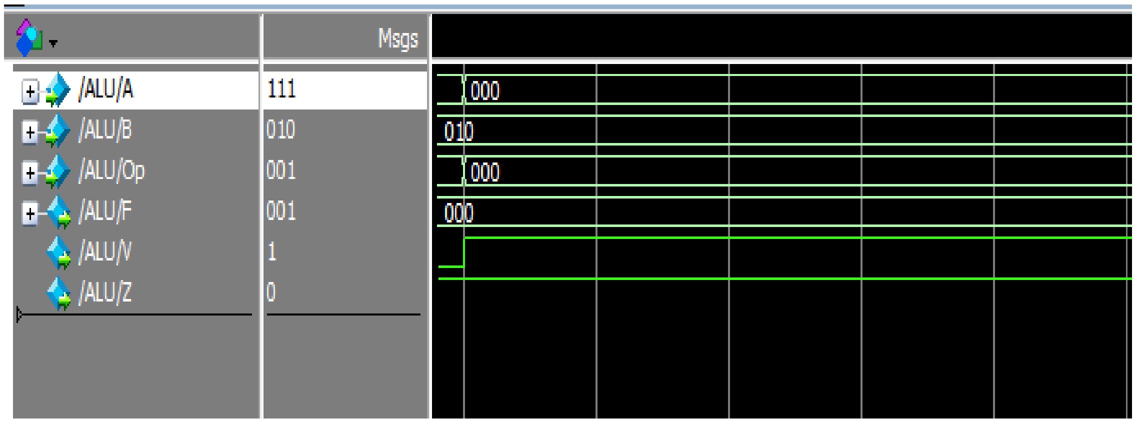
Wave execution example:



Fig(8)

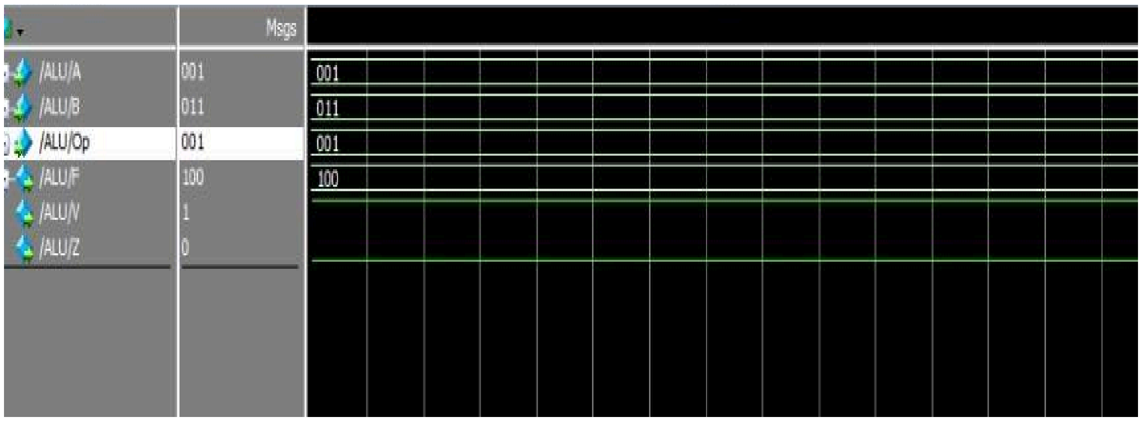
Two’s complement Overflow wave execution example:

We set A to 000 so the two’s complement will be 1000



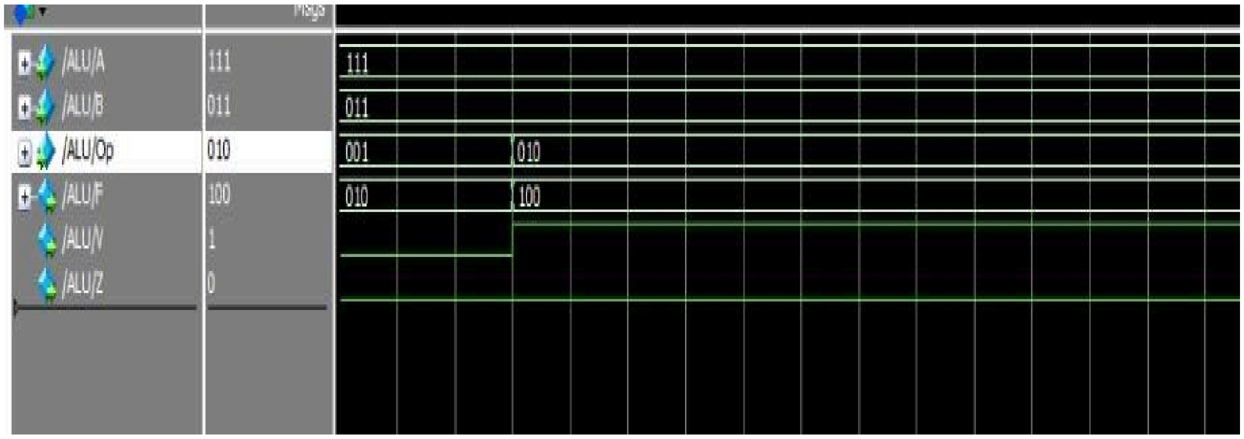
Fig(9)

Addition Overflow wave execution example:



Fig(10)

Subtraction Overflow wave execution example:



Fig(11)

OUR FULL CODE

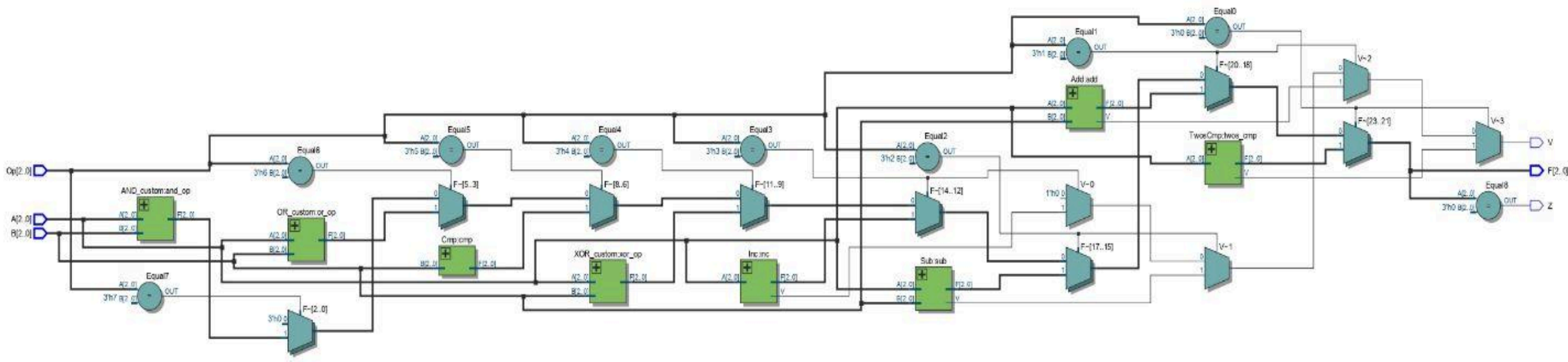
```
module ALU (
   input  logic [2:0] A,
   input  logic [2:0] B,
   input  logic [2:0] Op,
   output logic [2:0] F,
   output logic V,
   output logic Z
);

logic [2:0] F_add, F_sub, F_2scmp, F_inc, F_xor, F_cmp, F_or, F_and;
logic V_add, V_sub, V_2scmp, V_inc;

assign V = (Op == 3'b000) ? V_2scmp :
         (Op == 3'b001) ? V_add :
         (Op == 3'b010) ? V_sub :
         (Op == 3'b011) ? V_inc :
         1'b0;

assign F = (Op == 3'b000) ? F_2scmp :
         (Op == 3'b001) ? F_add :
         (Op == 3'b010) ? F_sub :
         (Op == 3'b011) ? F_inc :
         (Op == 3'b100) ? F_xor :
         (Op == 3'b101) ? F_cmp :
         (Op == 3'b110) ? F_or :
```

```
        (Op == 3'b111) ? F_and :
        3'b000;

assign Z = (F == 3'b000) ? 1'b1 : 1'b0;

Add add (
   .A(A),
   .B(B),
   .F(F_add),
   .V(V_add)
);

Sub sub (
   .A(A),
   .B(B),
   .F(F_sub),
   .V(V_sub)
);

TwosCmp twos_cmp (
   .A(A),
   .F(F_2scmp),
   .V(V_2scmp)
);

Inc inc (
   .A(A),
   .F(F_inc),
   .V(V_inc)
);

XOR xor_op (
   .A(A),
   .B(B),
   .F(F_xor)
);

Cmp cmp (
   .B(B),
   .F(F_cmp)
);
```

```
OR or_op (
   .A(A),
   .B(B),
   .F(F_or)
);

AND and_op (
   .A(A),
   .B(B),
   .F(F_and)
);

endmodule

module Add (
   input  logic [2:0] A,
   input  logic [2:0] B,
   output logic [2:0] F,
   output logic V
);
   logic [3:0] temp;
   assign temp = {1'b0, A} + {1'b0, B};
   assign F = temp[2:0];
   assign V = ((A[2] & B[2] & ~F[2]) | (~A[2] & ~B[2] & F[2]));
endmodule

module Sub (
   input  logic [2:0] A,
   input  logic [2:0] B,
   output logic [2:0] F,
   output logic V
);
   logic [3:0] temp;
   assign temp = {1'b0, B} - {1'b0, A};
   assign F = temp[2:0];
   assign V = (B[2] ^ A[2]) & (B[2] ^ F[2]);
endmodule

module TwosCmp (
   input  logic [2:0] A,
   output logic [2:0] F,
   output logic V
```

```
);
    logic [3:0] temp;
    assign temp = {1'b0, ~A} + 4'b0001;
    assign F = temp[2:0];
    assign V = temp[3];
endmodule

module Inc (
    input  logic [2:0] A,
    output logic [2:0] F,
    output logic V
);
    logic [3:0] temp;
    assign temp = {1'b0, A} + 4'b0001;
    assign F = temp[2:0];
    assign V = temp[3];
endmodule

module XOR (
    input  logic [2:0] A,
    input  logic [2:0] B,
    output logic [2:0] F
);
    assign F = A ^ B;
endmodule

module Cmp (
    input  logic [2:0] B,
    output logic [2:0] F
);
    assign F = ~B;
endmodule

module OR (
    input  logic [2:0] A,
    input  logic [2:0] B,
    output logic [2:0] F
);
    assign F = A | B;
endmodule

module AND (
    input  logic [2:0] A,
    input  logic [2:0] B,
    output logic [2:0] F
);
    assign F = A & B;
endmodule
```

Schematic



Fig(12)

At first, we searched for how to make a schematic that corresponds to our code. We ended up using Intel Quartus Prime Lite for its ease of use and support for System Verilog files.

We made our schematic through Intel Quartus by changing some minor things in the code to solve some conflicts. These were the names of our functions as some lead to errors during compilation in Intel Quartus. These changes did not change the logic of the code at all. And here is the Intel Quartus code:

```
module ALU (
    input  logic [2:0] A,
    input  logic [2:0] B,
    input  logic [2:0] Op,
    output logic [2:0] F,
    output logic V,
    output logic Z
);

logic [2:0] F_add, F_sub, F_2scmp, F_inc, F_xor, F_cmp, F_or, F_and;
logic V_add, V_sub, V_2scmp, V_inc;

assign V = (Op == 3'b000) ? V_2scmp :
```

```
          (Op == 3'b001) ? V_add :
          (Op == 3'b010) ? V_sub :
          (Op == 3'b011) ? V_inc :
          1'b0;

assign F = (Op == 3'b000) ? F_2scmp :
          (Op == 3'b001) ? F_add :
          (Op == 3'b010) ? F_sub :
          (Op == 3'b011) ? F_inc :
          (Op == 3'b100) ? F_xor :
          (Op == 3'b101) ? F_cmp :
          (Op == 3'b110) ? F_or :
          (Op == 3'b111) ? F_and :
          3'b000;

assign Z = (F == 3'b000) ? 1'b1 : 1'b0;

Add add (
   .A(A),
   .B(B),
   .F(F_add),
   .V(V_add)
);

Sub sub (
   .A(A),
   .B(B),
   .F(F_sub),
   .V(V_sub)
);

TwosCmp twos_cmp (
   .A(A),
   .F(F_2scmp),
   .V(V_2scmp)
);

Inc inc (
   .A(A),
   .F(F_inc),
   .V(V_inc)
);
```

```
XOR_custom xor_op (
   .A(A),
   .B(B),
   .F(F_xor)
);

Cmp cmp (
   .B(B),
   .F(F_cmp)
);

OR_custom or_op (
   .A(A),
   .B(B),
   .F(F_or)
);

AND_custom and_op (
   .A(A),
   .B(B),
   .F(F_and)
);

endmodule

module Add (
   input  logic [2:0] A,
   input  logic [2:0] B,
   output logic [2:0] F,
   output logic V
);
   logic [3:0] temp;
   assign temp = {1'b0, A} + {1'b0, B};
   assign F = temp[2:0];
   assign V = ((A[2] & B[2] & ~F[2]) | (~A[2] & ~B[2] & F[2]));
endmodule

module Sub (
   input  logic [2:0] A,
   input  logic [2:0] B,
   output logic [2:0] F,
  output logic V
);
  logic [3:0] temp;
  assign temp = {1'b0, B} - {1'b0, A};
  assign F = temp[2:0];
  assign V = (B[2] ^ A[2]) & (B[2] ^ F[2]);
endmodule

module TwosCmp (
  input  logic [2:0] A,
  output logic [2:0] F,
  output logic V
);
  logic [3:0] temp;
  assign temp = {1'b0, ~A} + 4'b0001;
  assign F = temp[2:0];
  assign V = temp[3];
endmodule

module Inc (
  input  logic [2:0] A,
  output logic [2:0] F,
  output logic V
);
  logic [3:0] temp;
  assign temp = {1'b0, A} + 4'b0001;
  assign F = temp[2:0];
  assign V = temp[3];
endmodule

module XOR_custom (
  input  logic [2:0] A,
  input  logic [2:0] B,
  output logic [2:0] F
);
  assign F = A ^ B;
endmodule

module Cmp (
  input  logic [2:0] B,
  output logic [2:0] F
);
```

```
    assign F = ~B;
endmodule

module OR_custom (
    input  logic [2:0] A,
    input  logic [2:0] B,
    output logic [2:0] F
);
    assign F = A | B;
endmodule

module AND_custom (
    input  logic [2:0] A,
    input  logic [2:0] B,
    output logic [2:0] F
);
    assign F = A & B;
endmodule
```

Challenges:

During our searching process we tried using many softwares to create our schematic. We tried exporting our code to VHDL to use LogicSim but that did not work. We tried using our original SystemVerilog file along with Yosys and graphviz but that ended up with an unsatisfactory result (fig 13). Hence with intel Quartus as it provided the most satisfying result out of all tested softwares.

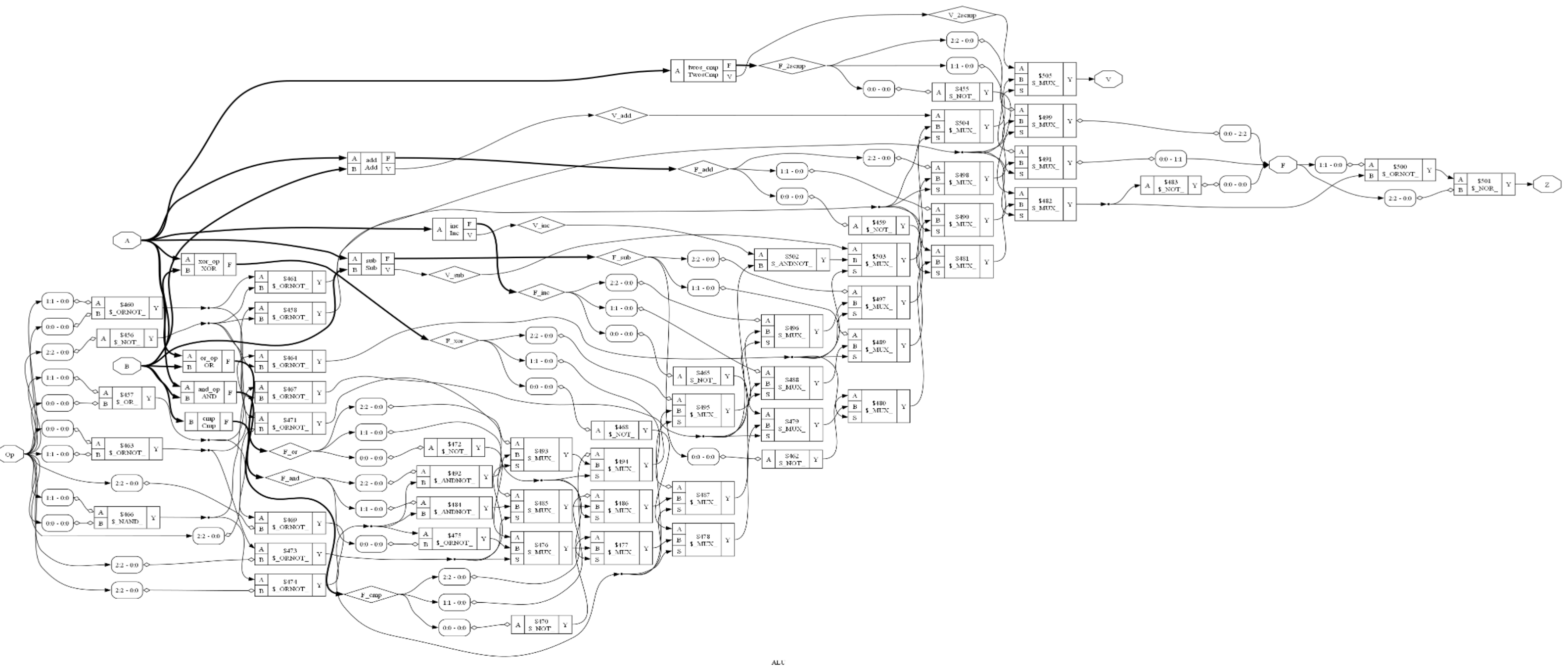


Fig (13).

CONCLUSION AND RECOMMENDATIONS

This project aimed to create a functional ALU capable of performing eight fundamental operations on 3-bit inputs, while correctly handling overflow and zero conditions through designated status flags.

Key Accomplishments:

1. **Functionality:** The ALU was designed to perform eight operations: 2's complement, addition, subtraction, increment, XOR, NOT, OR, and AND. Each operation was thoroughly tested to ensure accurate results.
2. **Overflow Handling:** One of the primary challenges was managing overflow. By employing a 4-bit temporary variable (temp) for intermediate calculations, we were able to detect overflow conditions effectively and set the overflow flag (V) when necessary.
3. **Zero Flag:** The zero flag (Z) was correctly implemented to indicate when the output (F) is zero. This feature is crucial for many computational processes that depend on the result being zero.
4. **Verilog Implementation:** The ALU was coded in Verilog, demonstrating a practical application of hardware description languages (HDLs) for digital design. The final Verilog code was structured for clarity and maintainability.
5. **Simulation and Verification:** Using ModelSim, we created a comprehensive testbench to simulate the ALU under various conditions. The waveforms generated during simulation verified the correctness of the ALU's operations and the proper setting of the status flags.
6. **Schematic Generation:** By generating schematics, we visually confirmed the logical structure of the ALU, ensuring that the design aligns with the intended functionality.

Challenges and Solutions:

- **Overflow Detection:** Initial approaches to handling overflow were inadequate due to the limitations of 3-bit representation. The final approach using a 4-bit temporary variable (temp) resolved this issue effectively.
- **Signal Representation:** Ensuring that the ALU's output remained a 3-bit signal while accurately detecting overflow required careful handling of intermediate results.

Future Work:

- **Optimization:** Future iterations could explore optimizing the ALU for speed and resource utilization, potentially extending its capabilities.
- **Scalability:** Extending the ALU design to support larger bit-widths would make it applicable to more complex computational tasks.
- **Integration:** Integrating this ALU into a larger CPU design could demonstrate its practical utility in a complete computing system.