ALU PROJECT

BRIEF INTRODUCTION

This project focuses on designing and verifying a parameterized Arithmetic Logic Unit (ALU) in Verilog. The ALU performs a wide range of arithmetic and logical operations and supports configurable data width and command width. Though internally combinational, the design uses pipelined output registers (RES, OFLOW, COUT, G, L, E, ERR), introducing a one-cycle delay to support higher clock frequencies.

The ALU supports two modes—arithmetic and logical—controlled by the MODE signal. It handles operations like addition, subtraction (with/without carry), comparison, increment/decrement, multiplication, shifts, rotates, and basic logic functions. Input validation through INP_VALID allows selective operand usage.

Verification is a key part of the project, focusing on achieving high code coverage. This ensures all functional paths are tested, corner cases are identified, and the design is robust and reliable under all specified conditions.

OBJECTIVES

- To implement and design an Arithmetic Logic Unit(ALU) in Verilog HDL that can perforrm a wide range of arithmetic and logic functions.
- To facilitate parameterization for data width and command structure to enable reusability and scalability.
- To provide for proper handling of signed and unsigned operations, carry, overflow, and comparison flags.
- To confirm the proper functioning and correctness of the ALU using exhaustive simulation testbenches in a Verilog simulation environment.
- To exhibit the behavior of the ALU under various operating modes and input scenarios, such as edge cases and erroneous inputs.

ARCHITECTURE

The ALU's architecture is meticulously designed to process arithmetic and logical operations efficiently, structured around a clear separation of input, control, and output stages. It consists of the following key components:

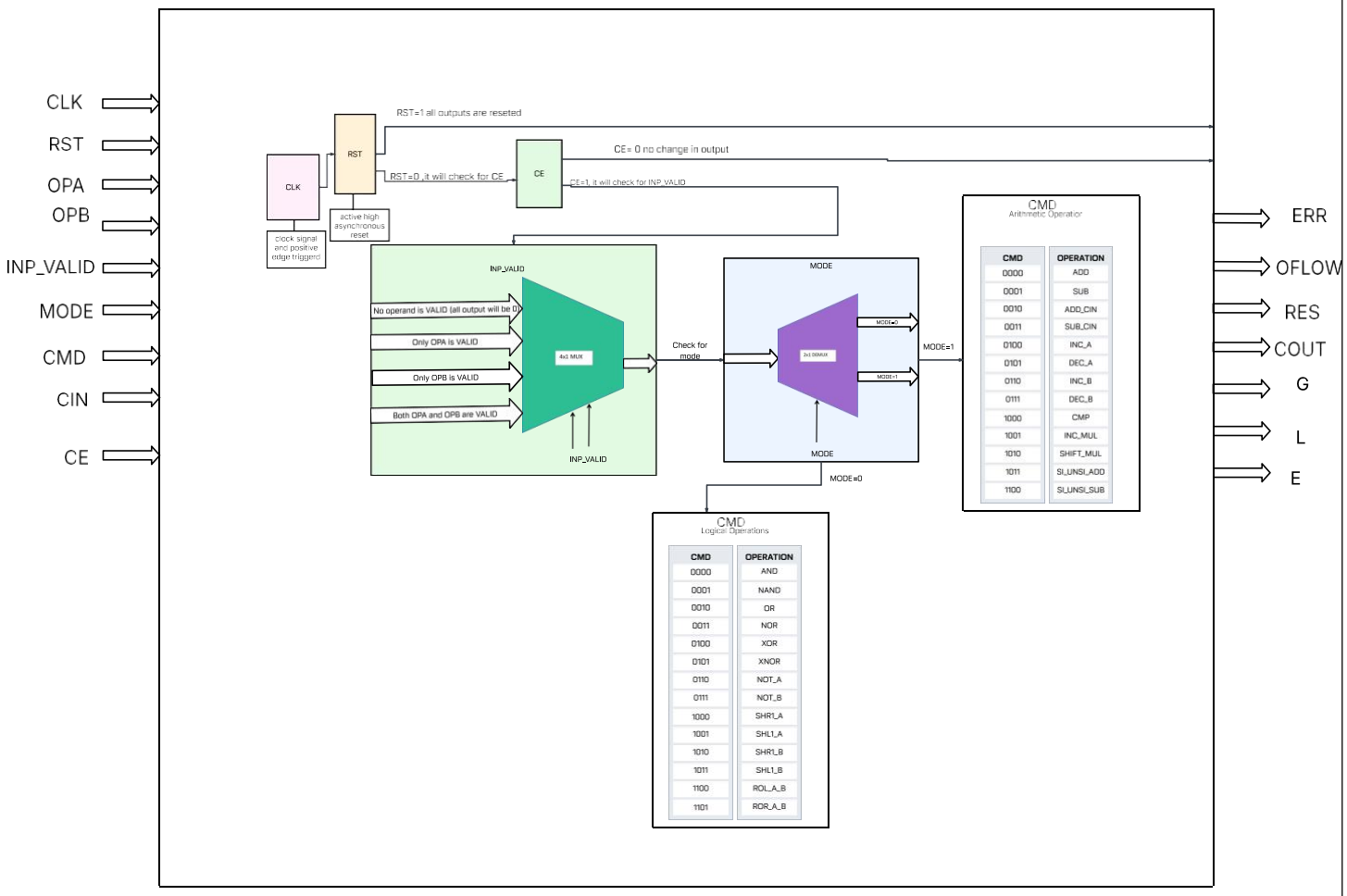
1. **Input Operands**: The ALU accepts two primary data inputs, OPA and OPB, which serve as the operands for the various arithmetic and logical computations.
2. **Clock and Reset Logic (**CLK**,** RST**):**
    1. CLK is the clock signal, indicating that the system is positive edge triggered.
    2. RST is an active-high, asynchronous reset signal. When RST is high, all outputs are reset.
    3. When RST is low (RST=0), the CE (Chip Enable) signal is checked.
3. **Clock Enable Logic (**CE**):**
    1. CE acts as an overall enable for the ALU.
    2. If CE is low (CE=0), there is no change in the output.
    3. If CE is high (CE=1), the INP_VALID signal is checked.
4. **Input Validation Logic (**INP_VALID**):** This block controls which operands are active for a given operation.
    1. **No operand is VALID (2'b00):** In this state, all outputs will be reset.
    2. **Only OPA is VALID (2'b01):** Operations will be performed solely on OPA.
    3. **Only OPB is VALID (2'b10):** Operations will be performed solely on OPB.
    4. **Both OPA and OPB are VALID (2'b11):** Operations requiring two operands will proceed.
    5. The INP_VALID signal selects one of these four paths, which then feeds into the main processing path.
5. **Mode Selection (**MODE**):** This crucial control signal determines the broad category of operation the ALU will perform:
    1. MODE = 1 **(Arithmetic Operations):** When MODE is high, the ALU operates in arithmetic mode. The CMD input is decoded based on the CMD table for arithmetic operations. This includes operations like ADD, SUB, ADD_CIN, SUB_CIN, INC_A, DEC_A, INC_B, DEC_B, CMP (for G, L, E flags), INC_MUL, SHIFT_MUL, SI_UNSI_ADD, and SI_UNSI_SUB.

2. MODE = 0 **(Logical Operations):** When MODE is low, the ALU performs logical operations. The CMD input is then decoded according to the CMD table for logical operations. These operations include AND, NAND, OR, NOR, XOR, XNOR, NOT_A, NOT_B, SHR1_A, SHL1_A, SHR1_B, SHL1_B, ROL_A_B, and ROR_A_B.
3. A multiplexer, controlled by MODE, directs the validated inputs to the appropriate arithmetic or logical computation unit.

6. **Control Logic for Opcode Decoding (**CMD**):** The CMD input, in conjunction with the MODE signal, precisely defines the operation to be executed. Dedicated decoding logic interprets the CMD code within the selected mode to activate the specific functional unit (e.g., adder, subtractor, shifter, comparator, multiplier, logical gate array).
7. **Arithmetic and Logical Execution Units:** These are the core computational blocks that perform the actual operations. The diagram implies separate units or shared resources configured by the CMD and MODE signals.
8. **Output Logic with Flag Generation:** After computation, the results are formatted and various status flags are generated.

    1. RES: The primary output, providing the result of the operation.
    2. OFLOW: Indicates an overflow condition, primarily relevant for signed arithmetic operations.
    3. COUT: Represents the carry-out from arithmetic operations.
    4. G**,** L**,** E: Comparison flags indicating Greater Than, Less Than, or Equal to, set by the CMP operation.
    5. ERR: An error flag, specifically used to indicate invalid shift amounts in rotate operations (ROL_A_B, ROR_A_B).

This structured approach, with clear control flow based on RST, CE, INP_VALID, and MODE signals, allows for a highly functional and controllable ALU.



CMD	OPERATION
0000	ADD
0001	SUB
0010	ADD_CIN
0011	SUB_CIN
0100	INC_A
0101	DEC_A
0110	INC_B
0111	DEC_B
1000	CMP
1001	INC_MUL
1010	SHIFT_MUL
1011	SI_UNSI_ADD
1100	SI_UNSI_SUB

CMD	OPERATION
0000	AND
0001	NAND
0010	OR
0011	NOR
0100	XOR
0101	XNOR
0110	NOT_A
0111	NOT_B
1000	SHR1_A
1001	SHL1_A
1010	SHR1_B
1011	SHL1_B
1100	ROL_A_B
1101	ROR_A_B

WORKING

Operands and opcode are processed combinationally to perform the selected operation and update status flags accordingly. The Arithmetic Logic Unit (ALU) uses control signals to determine the type of operation and processes inputs without waiting for clock edges during computation.

❖ Operation Mode Selection – MODE Signal

- MODE = 1 (Arithmetic Mode): Enables arithmetic operations like ADD, SUB, INC, DEC, MUL, and signed variants such as ADD_SIGN and SUB_SIGN.

- MODE = 0 (Logical Mode): Enables logical operations including AND, OR, XOR, NOT, NAND, NOR, XNOR, SHL, SHR, and ROTATE operations.

❖ **Operand Validity – INP_VALID Signal**

Operand usage is determined by the 2-bit INP_VALID signal:

- 2'b00: Both OPA and OPB are invalid → operation skipped; ERR is asserted.
- 2'b01: Only OPA is valid → single-operand operation performed on OPA.
- 2'b10: Only OPB is valid → single-operand operation performed on OPB.
- 2'b11: Both operands are valid → full binary operations enabled.

❖ **Combinational Operation and Result Handling**

All supported ALU operations are computed in a purely combinational manner. Results are first stored in an intermediate register and then transferred to the output register RES at the next rising clock edge, introducing a 1-cycle latency for output updates. This ensures stability and synchronization without glitches.

❖ **Status Flag Updates**

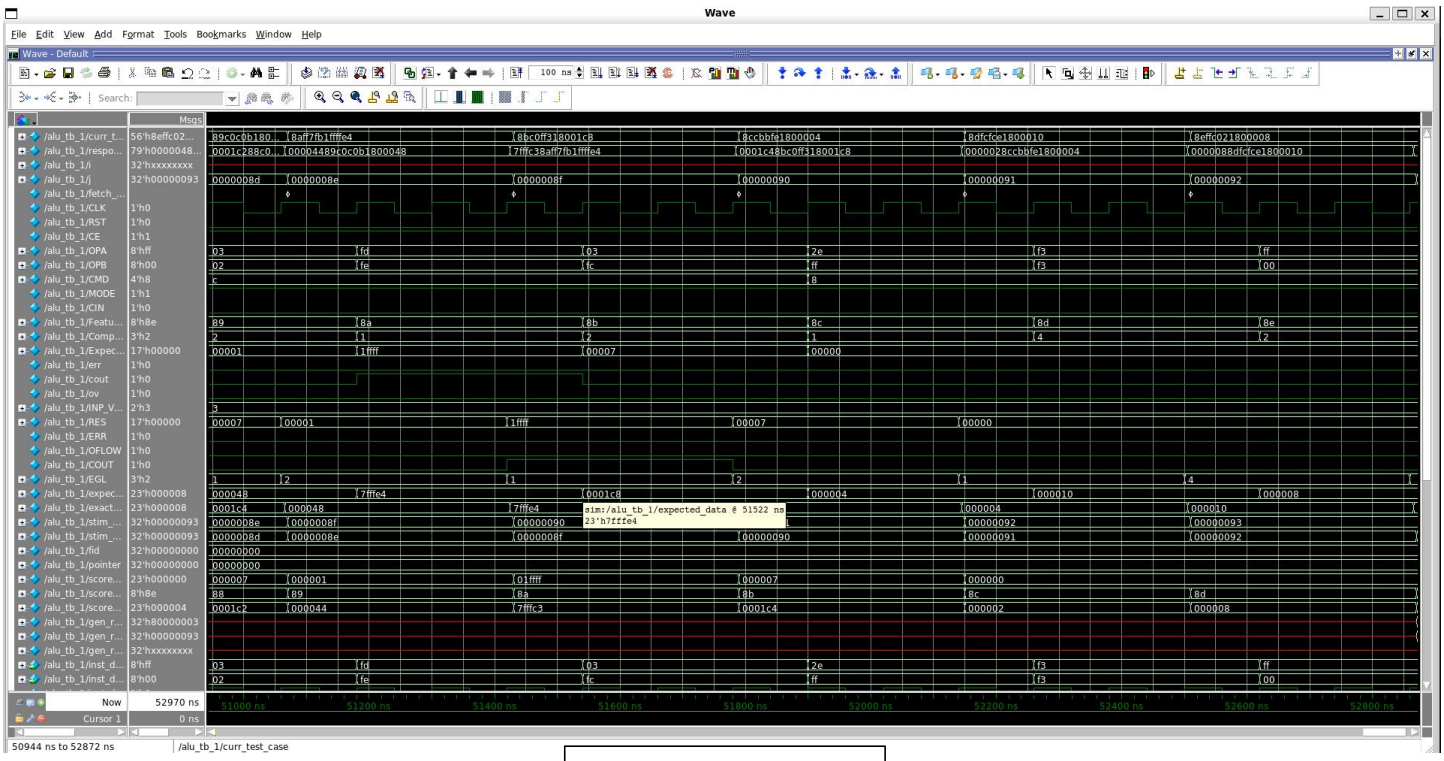
- COUT: Set during unsigned operations with carry-out.
- OFLOW: Set during signed arithmetic overflow.
- G, E, L: Comparison flags indicating Greater, Equal, or Less.
- ERR: Indicates invalid operand combinations or unsupported operation codes.

RESULT

The simulation confirms accurate operation outputs and correct flag behavior for all defined instructions, as observed in the waveform. The Verilog-based ALU executes all arithmetic and logical operations correctly according to the provided opcode. It processes inputs accurately based on the MODE and INP_VALID control signals and generates valid outputs including the result (RES) and status flags (COUT, OFLOW, G, E, L, and ERR).

Multiplication operations correctly demonstrate one cycle delay, implemented using internal pipeline registers. Waveform analysis verifies that the ALU behaves as expected across all supported operations, including edge cases such as overflows, comparisons, and invalid input conditions.

This validates the ALU's functional correctness and the effective integration of pipelining for delay-sensitive computations.



Questa Sim Waveform

CONCLUSION

The Verilog-based ALU meets all functional specifications, delivering consistent and accurate performance across a broad range of arithmetic and logical operations. Its modular architecture ensures a clear separation of functionalities, simplifying debugging, extension, and reuse. Internal registers enable pipelining for proper timing in multi-cycle operations like multiplication, while maintaining single-cycle execution for combinational instructions.

Simulation results validate the ALU's correctness by confirming accurate outputs and proper status flag updates under diverse conditions. Moreover, the design is fully synthesizable and optimized for efficient hardware implementation, making it well-suited for integration into more complex digital systems.

SYNTHESIZABLE CODE

```
module design_5 #(parameter width = 8, parameter cwidth =4)(

input [width-1:0] OPA, OPB,

input CLK, RST, CE, MODE, CIN,

input [1:0] INP_VALID,

input [cwidth-1:0] CMD,

output reg [2*width:0] RES,
```

```
output reg OFLOW,
output reg COUT,
output reg G,
output reg L,
output reg E,
output reg ERR
);

reg [2*width:0] temp;

//using temporary registers to store to cause a delay of 1 clock cycle
reg [2*width:0] next_RES;
reg next_COUT, next_OFLOW, next_G, next_L, next_E, next_ERR;

//MODE =1
//VALID_INP=2'b11
localparam CMD_ADD = 4'b0000;
localparam CMD_SUB = 4'b0001;
localparam CMD_ADD_CIN = 4'b0010;
localparam CMD_SUB_CIN = 4'b0011;
localparam CMD_EGL = 4'b1000;
localparam CMD_INC_MUL = 4'b1001;
localparam CMD_SHIFT_MUL = 4'b1010;
localparam CMD_SI_UNSI_ADD = 4'b1011;
localparam CMD_SI_UNSI_SUB = 4'b1100;

//VALID_INP=2'b01 (A is Valid)
localparam CMD_INC_A = 4'b0100;
```

```
localparam CMD_DEC_A = 4'b0101;


//VALID_INP =2'b10 (B is valid)
localparam CMD_INC_B = 4'b0110;
localparam CMD_DEC_B = 4'b0111;


///MODE =0
//VALID_INP=2'b11
localparam CMD_AND = 4'b0000;
localparam CMD_NAND = 4'b0001;
localparam CMD_OR = 4'b0010;
localparam CMD_NOR = 4'b0011;
localparam CMD_XOR = 4'b0100;
localparam CMD_XNOR = 4'b0101;
localparam CMD_ROL_A_B = 4'b1100;
localparam CMD_ROR_A_B = 4'b1101;


//VALID_INP =2'b01 (A is valid)
localparam CMD_NOT_A = 4'b0110;
localparam CMD_SHR1_A = 4'b1000;
localparam CMD_SHLI_A = 4'b1001;


//VALID_INP =2'b10 (B is valid)
localparam CMD_NOT_B = 4'b0111;
localparam CMD_SHR1_B = 4'b1010;
localparam CMD_SHL1_B = 4'b1011;


wire [2:0] amount;
```

```
assign amount = OPB[2:0];


always@(posedge CLK or posedge RST)
begin
        if(RST)begin
                next_RES <= 'b0;
                next_COUT <= 1'b0;
                next_OFLOW <= 1'b0;
                next_G <= 1'b0;
                next_E <= 1'b0;
                next_L <= 1'b0;
                next_ERR <= 1'b0;
        end
        else if(CE)
        begin
                if(INP_VALID == 2'b00)
                begin
                         next_RES <= 'b0;
                        next_COUT <= 1'b0;
                        next_OFLOW <= 1'b0;
                        next_G <= 1'b0;
                        next_E <= 1'b0;
                        next_L <= 1'b0;
                        next_ERR <= 1'b0;
                end
                else if(INP_VALID ==2'b11)
                begin
                        next_RES <= 'b0;
```

```
next_COUT <= 1'b0;
next_OFLOW <= 1'b0;
next_G <= 1'b0;
next_E <= 1'b0;
next_L <= 1'b0;
next_ERR <= 1'b0;
if(MODE)begin
case(CMD)
        CMD_ADD:
        begin
        temp = OPA + OPB;
        next_RES <= temp;
        next_COUT <= temp[width];
        end
        CMD_SUB:
        begin
        next_OFLOW <= (OPA<OPB)?1'b1:1'b0;
        next_RES <= OPA-OPB;
        end
        CMD_ADD_CIN:
        begin
        temp = OPA + OPB + CIN;
        next_RES <= temp;
        next_COUT <= temp[width];
        end
        CMD_SUB_CIN:
        begin
        next_OFLOW <= (OPA<OPB)?1'b1:1'b0;
```

```
next_RES <= OPA-OPB-CIN;
end
CMD_EGL:
begin
next_RES <= 'b0;
if(OPA==OPB)
begin
next_E <= 1'b1;
next_G <= 1'b0;
next_L <= 1'b0;
end
else if(OPA>OPB)
begin
next_E <= 1'b0;
next_G <= 1'b1;
next_L <= 1'b0;
end
else
begin
next_E <= 1'b0;
next_G <= 1'b0;
next_L <= 1'b1;
end
end
CMD_INC_MUL:
begin
temp = (OPA+1) * (OPB+1);
next_RES <= temp;
```

```
next_COUT <= temp[2*width];

end

CMD_SHIFT_MUL:

begin

temp =(OPA<<1)*OPB;

next_RES <= temp;

next_COUT <= temp[2*width];

end

CMD_SI_UNSI_ADD:

begin

temp = $signed(OPA) + $signed(OPB);

next_RES <= temp;

next_COUT <= temp[width];

next_OFLOW <= ((OPA[width-1] == OPB[width-1]) &&
                (temp[width-1] != OPA[width-1])) ? 1'b1 : 1'b0;

if($signed(OPA)==$signed(OPB))

begin

next_E <= 1'b1;

next_G <= 1'b0;

next_L <= 1'b0;

end

else if($signed(OPA)>$signed(OPB))

begin

next_E <= 1'b0;

next_G <= 1'b1;

next_L <= 1'b0;

end

else
```

```
begin
next_E <= 1'b0;
next_G <= 1'b0;
next_L <= 1'b1;
end
end
CMD_SI_UNSI_SUB:
begin
temp = $signed(OPA) - $signed(OPB);
next_RES<= temp;
next_COUT <= temp[width];
next_OFLOW <= ((OPA[width-1] != OPB[width-1]) &&
                (temp[width-1] != OPA[width-1])) ? 1'b1 : 1'b0;
if($signed(OPA) == $signed(OPB))
begin
next_E <= 1'b1;
next_G <= 1'b0;
next_L <= 1'b0;
end
else if($signed(OPA) > $signed(OPB))
begin
next_E <= 1'b0;
next_G <= 1'b1;
next_L <= 1'b0;
end
else
begin
next_E <= 1'b0;
```

```
                next_G <= 1'b0;
                next_L <= 1'b1;
                end
                end
                default:
                begin
                next_RES <= 'b0;
                next_COUT <= 1'b0;
                next_OFLOW <= 1'b0;
                next_G <= 1'b0;
                next_E <= 1'b0;
                next_L <= 1'b0;
                next_ERR <= 1'b0;
                end
                endcase
                end
        else begin
                case(CMD)
CMD_AND:next_RES <= {{width*2-(width-1){1'b0}},OPA&OPB};
CMD_NAND:next_RES <= {{width*2-(width-1){1'b0}},~(OPA&OPB)};
CMD_OR:next_RES <= {{width*2-(width-1){1'b0}},OPA|OPB};
CMD_NOR:next_RES <= {{width*2-(width-1){1'b0}},~(OPA|OPB)};
CMD_XOR:next_RES <= {{width*2-(width-1){1'b0}},OPA^OPB};
CMD_XNOR:next_RES <= {{width*2-(width-1){1'b0}},~(OPA^OPB)};
CMD_ROL_A_B:
begin
if (amount == 0)
        next_RES <= {{width*2-(width-1){1'b0}}, OPA};
```

```
                else
                        next_RES <= {{width*2-(width){1'b0}}, (OPA << amount) |
                                    (OPA >> (width - amount))};
                        next_ERR <= (width > 3 && |OPB[width-1:3]) ? 1'b1 : 1'b0;
                end
                CMD_ROR_A_B:
                begin
                if (amount == 0)
                        next_RES <= {{width*2-(width-1){1'b0}}, OPA};
                else
                        next_RES <= {{width*2-(width-1){1'b0}}, (OPA >> amount) | (OPA
                                    << (width - amount))};
                        next_ERR <= (width > 3 && |OPB[width-1:3]) ? 1'b1 : 1'b0;
                end
                default:
                begin
                next_RES <='b0;
                next_COUT <= 1'b0;
                next_OFLOW <= 1'b0;
                next_G <= 1'b0;
                next_E <= 1'b0;
                next_L <= 1'b0;
                next_ERR <= 1'b0;
                end
                endcase
        end
end
else if(INP_VALID == 2'b01)
begin
```

```
next_RES <=  'b0;
next_COUT <= 1'b0;
next_OFLOW <= 1'b0;
next_G <= 1'b0;
next_E <= 1'b0;
next_L <= 1'b0;
next_ERR <= 1'b0;
if(MODE)begin
        case(CMD)
                CMD_INC_A:next_RES <= OPA+1;
                CMD_DEC_A:next_RES <= OPA-1;
                default:
                begin
                next_RES <= 'b0;
                next_COUT <= 1'b0;
                next_OFLOW <= 1'b0;
                next_G <= 1'b0;
                next_E <= 1'b0;
                next_L <= 1'b0;
                next_ERR <= 1'b0;
                end
        endcase
end
else begin
        case(CMD)
                CMD_NOT_A:next_RES <= {{width*2-(width-1){1'b0}},~OPA};
                CMD_SHR1_A:next_RES <= {{width*2-(width1){1'b0}},OPA>>1};
                CMD_SHLI_A:next_RES <= {{width*2-(width-1){1'b0}},OPA<<1};
```

```
                default:
                begin
                next_RES <= 'b0;
                next_COUT <= 1'b0;
                next_OFLOW <= 1'b0;
                next_G <= 1'b0;
                next_E <= 1'b0;
                next_L <= 1'b0;
                next_ERR <= 1'b0;
                end
            endcase
        end
        end
else if(INP_VALID == 2'b10)
begin
        next_RES <= 'b0;
        next_COUT <= 1'b0;
        next_OFLOW <= 1'b0;
        next_G <= 1'b0;
        next_E <= 1'b0;
        next_L <= 1'b0;
        next_ERR <= 1'b0;
        if(MODE) begin
            case(CMD)
                CMD_INC_B: next_RES <= OPB + 1;
                CMD_DEC_B: next_RES <= OPB - 1;
                default:
                begin
```

```
                next_RES <= 'b0;
                next_COUT <= 1'b0;
                next_OFLOW <= 1'b0;
                next_G <= 1'b0;
                next_E <= 1'b0;
                next_L <= 1'b0;
                next_ERR <= 1'b0;
                end
        endcase
        end
    else begin
        case(CMD)
                CMD_NOT_B:next_RES <= {{width*2-(width-1){1'b0}},~OPB};
                CMD_SHR1_B:next_RES <= {{width*2-(width-1){1'b0}},OPB>>1};
                CMD_SHL1_B:next_RES <= {{width*2-(width-1){1'b0}},OPB<<1};
                default:
                begin
                next_RES <= 'b0;
                next_COUT <= 1'b0;
                next_OFLOW <= 1'b0;
                next_G <= 1'b0;
                next_E <= 1'b0;
                next_L <= 1'b0;
                next_ERR <= 1'b0;
                end
        endcase
        end
    end
```

```
        end
end
always @(posedge CLK or posedge RST) begin
        if(RST) begin
                RES <= 'b0;
                COUT <= 1'b0;
                OFLOW <= 1'b0;
                G <= 1'b0;
                E <= 1'b0;
                L <= 1'b0;
                ERR <= 1'b0;
        end else if (CE) begin
                RES <= next_RES;
                COUT <= next_COUT;
                OFLOW <= next_OFLOW;
                G <= next_G;
                L <= next_L;
                E <= next_E;
                ERR <= next_ERR;
        end
        end
Endmodule
```

FUTURE IMPROVEMENT

The following enhancements are planned to upgrade the current ALU design:

Pipelining: Multi-stage pipelining will be integrated to increase throughput and allow multiple instructions to be processed simultaneously.

Expanded Functionality: The ALU will be extended to support additional operations such as division, modulus, and barrel shift for greater versatility.

Scalable Data Width: The design will be modified to handle wider data widths like 16-bit and 32-bit, allowing more precise and complex computations.

Formal Verification Methods: Formal verification techniques will be introduced to rigorously validate the ALU's correctness for all possible input conditions.

Efficient Multiplication: The existing delay-based multiplication approach will be replaced with a pipelined multiplier unit for better performance in multi-cycle operations.