Vietnamese-German University

OTH REGENSBURG

PROJECT REPORT: 8-BIT PARAMETERIZED ARITHMETIC LOGIC UNIT (ALU)

Course: Hardware Description Languages & Verification
Tools Used: Xilinx Vivado
Date: [21/01/2026]

1. Participant

1. Tong Ngoc Trung Kien - Student ID: 10223041
2. Nguyen Tran Ha Long - Student ID: 10223046

2. Introduction

The primary objective of this project is to design, implement, and verify an 8-bit Arithmetic Logic Unit (ALU) using VHDL. The ALU serves as the central processing core for digital systems, capable of performing arithmetic, logical, and bit-manipulation operations. The design targets FPGA implementation and emphasizes reusability through parameterization.

3. Design Specifications

The ALU is designed according to the following specifications:

- **Data width:**
  Two 8-bit input operands **A** and **B**, and one 8-bit output **ALU_Out**.

- **Control signal:**
  A 4-bit opcode **ALU_Sel** selects one of 16 different ALU operations.

- **Status flag:**
  A single-bit **Carryout** flag indicates carry or overflow conditions in arithmetic operations.

- **Processing block:**
  The internal combinational block **Y** performs the selected arithmetic or logic operation.

- **Architecture type:**
  Purely combinational logic (no clock, no memory elements).

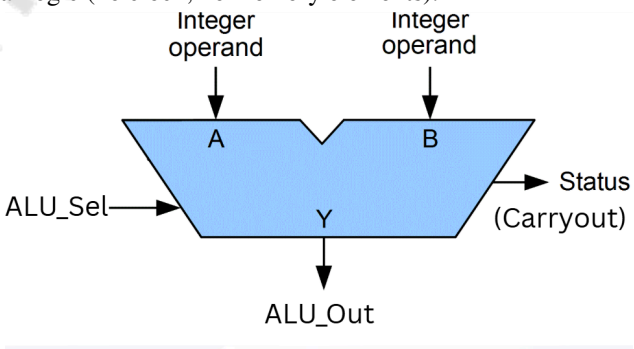


Figure 1: High-Level Block Diagram

4. Theoretical Background

4.1. ALU Architecture

The ALU entity defines an 8-bit combinational arithmetic logic unit. Two 8-bit operands A and B are provided as inputs, while the 4-bit ALU_Sel signal determines the selected operation.
The output ALU_Out returns the 8-bit result of the operation. The Carryout signal is used to indicate carry in addition and borrow in subtraction operations. The generic parameter N is introduced to control the shift and rotate amount, allowing the ALU to be easily reconfigured without
modifying the internal logic.

4.2. VHDL Standards: `Numeric_Std` vs. `Std_Logic_Arith`

To ensure industry-standard compliance, this design utilizes the **`ieee.numeric_std`** library. Unlike the proprietary **`std_logic_arith`**, **`numeric_std`** is an official IEEE standard that strictly distinguishes between **`signed`** and **`unsigned`** types. This prevents ambiguous behavior during synthesis and simulation, ensuring that arithmetic operations like addition and comparison behave exactly as intended across different EDA tools.

5. Design & Implementation

5.1. Entity Description

The Entity declares the interface of the 9 ALU. A key feature of this design is the usage of the Generic parameter **`N`.** This allows the shift/rotate amount to be configured at instantiation time, enhancing the module's reusability for different system requirements without modifying the source code.

Code Listing 1: ALU Entity Declaration

```
entity ALU is
    generic (
        N : natural := 1  -- Shift/Rotate amount (Default = 1)
    );
    port (
        A        : in  std_logic_vector(7 downto 0);
        B        : in  std_logic_vector(7 downto 0);
        ALU_Sel  : in  std_logic_vector(3 downto 0);
        ALU_Out  : out std_logic_vector(7 downto 0);
        Carryout : out std_logic
    );
end ALU;
```

This entity defines a reusable and configurable ALU interface. The separation between interface (entity) and functionality (architecture) improves modularity and supports clean hierarchical design.

5.2. Architecture Strategy

The internal logic is implemented using a Behavioral Architecture within a single VHDL process. The design strategy focuses on two key technical decisions:

5.2.1. Use of Variables

```
process(A, B, ALU_Sel)
  variable v_A, v_B   : unsigned(7 downto 0);
  variable v_sum9     : unsigned(8 downto 0);
  variable v_result   : unsigned(7 downto 0);
begin
```

Variables are used to store intermediate values because they update immediately within a process. This allows sequential-style arithmetic calculations while still modeling purely combinational hardware.

5.2.2. Carry and Overflow Handling

```
v_sum9 := ('0' & v_A) + ('0' & v_B);
Carryout <= v_sum9(8);
```

By extending both operands to 9 bits, the most significant bit naturally captures carry or borrow information without additional logic.

6. Supported Operation

6.1. Arithmetic Operations (0000 – 0011)

```
when "0000" =>  -- ADD
  v_sum9   := ('0' & v_A) + ('0' & v_B);
  v_result := v_sum9(7 downto 0);
  Carryout <= v_sum9(8);

when "0001" =>  -- SUB
  v_sum9   := ('0' & v_A) - ('0' & v_B);
  v_result := v_sum9(7 downto 0);
```

Arithmetic operations are implemented using unsigned arithmetic. For addition and subtraction, operands are extended to 9 bits so that the most significant bit can capture carry or borrow information. Multiplication results are truncated to 8 bits, while division includes a divide-by-zero protection mechanism.

6.2. Shift and Rotate Operations (0100 – 0111)

```
when "0100" =>  -- SLL
  v_result := v_A sll N;

when "0101" =>  -- SRL
  v_result := v_A srl N;

when "0110" =>  -- ROL
  v_result := v_A rol N;

when "0111" =>  -- ROR
  v_result := v_A ror N;
```

Shift and rotate operations use the generic parameter N to define the shift amount. Logical shifts insert zeros, while rotate operations preserve all bits by wrapping them around the vector boundaries.

6.3. Logic Operations (1000 – 1101)

```
when "1000" =>  -- AND
  v_result := v_A and v_B;

when "1001" =>  -- OR
  v_result := v_A or v_B;

when "1010" =>  -- XOR
  v_result := v_A xor v_B;

when "1011" =>  -- NAND
  v_result := not (v_A and v_B);

when "1100" =>  -- NOR
  v_result := not (v_A or v_B);

when "1101" =>  -- XNOR
  v_result := not (v_A xor v_B);
```

Logic operations are applied on a bit-by-bit basis using standard Boolean operators. These operations are purely combinational and do not affect the Carryout signal.

6.4. Comparison Operations (1110-1111)

```
when "1110" =>  -- GREATER THAN
  if v_A > v_B then
    v_result := x"01";
  else
    v_result := x"00";
  end if;

when "1111" =>  -- EQUAL
  if v_A = v_B then
    v_result := x"01";
  else
    v_result := x"00";
  end if;
```

Comparison operations return a Boolean result encoded as an 8-bit value. The value `0x01` represents a true condition, while `0x00` represents false. This encoding simplifies verification and integration with control logic.

7. Simulation and Verification

The ALU was verified using a self-checking VHDL testbench. The testbench applies predefined input vectors and validates the outputs using `assert` statements.
Verified scenarios include:

- Normal addition and overflow detection (e.g., `255 + 1` resulting in carryout is pulled to 1).

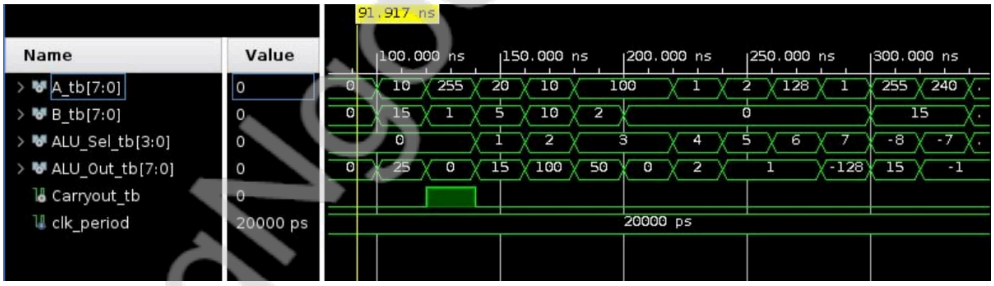


Figure 2: Overall simulation waveform of the ALU showing input operands, control signal and corresponding outputs.

- Subtraction and borrow behavior.
- Multiplication and division, including division-by-zero protection.
- Shift and rotate operations using generic `N = 1.`
- Logical operations and comparison results.

Simulation waveforms confirm that all outputs match the expected values, and no assertion failures occurred during testing.

7.1. Testbench Stimulus and Timing

```
A_tb <= x"0A";
B_tb <= x"0F";
```

```
ALU_Sel_tb <= "0000";
wait for clk_period;
```

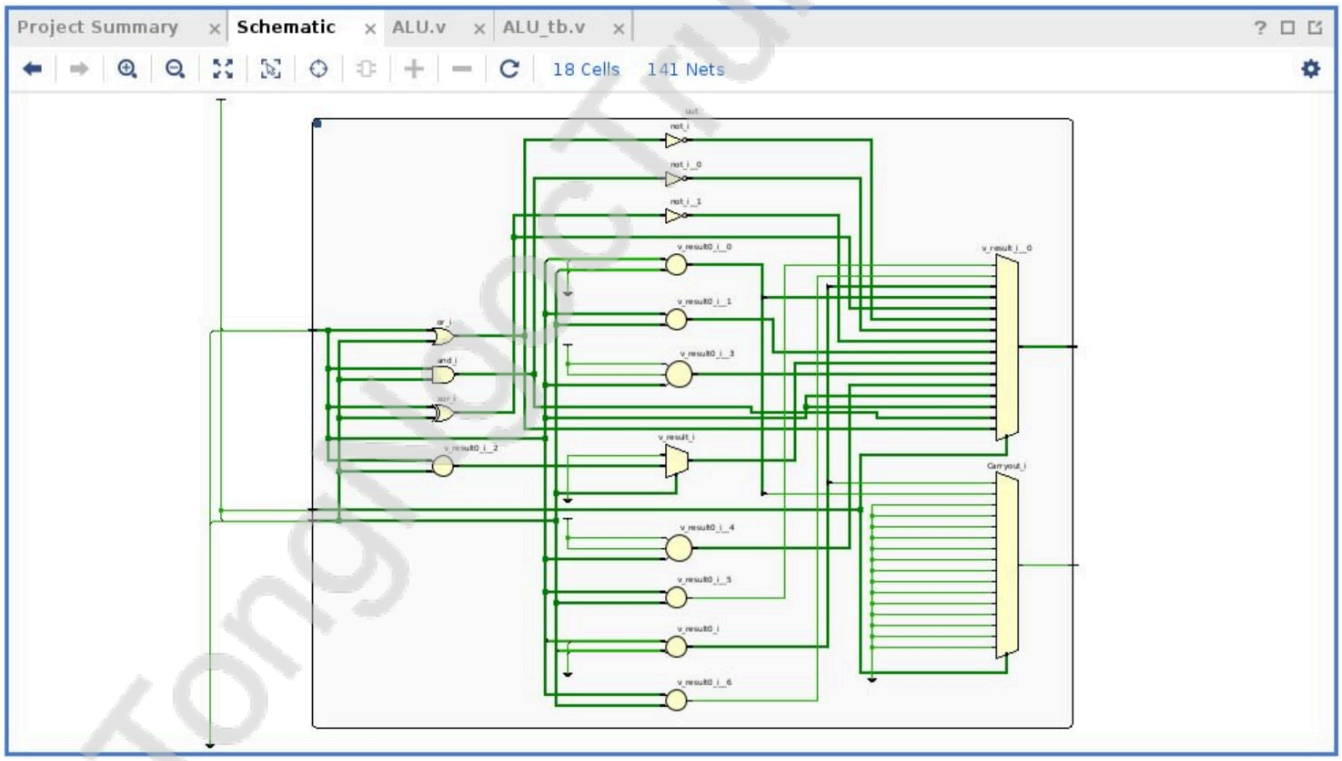
Each test case applies a new set of inputs followed by a fixed delay. Although the ALU is combinational, the wait statement ensures signal stability and improves waveform readability.

7.2. Use of `assert` Statements

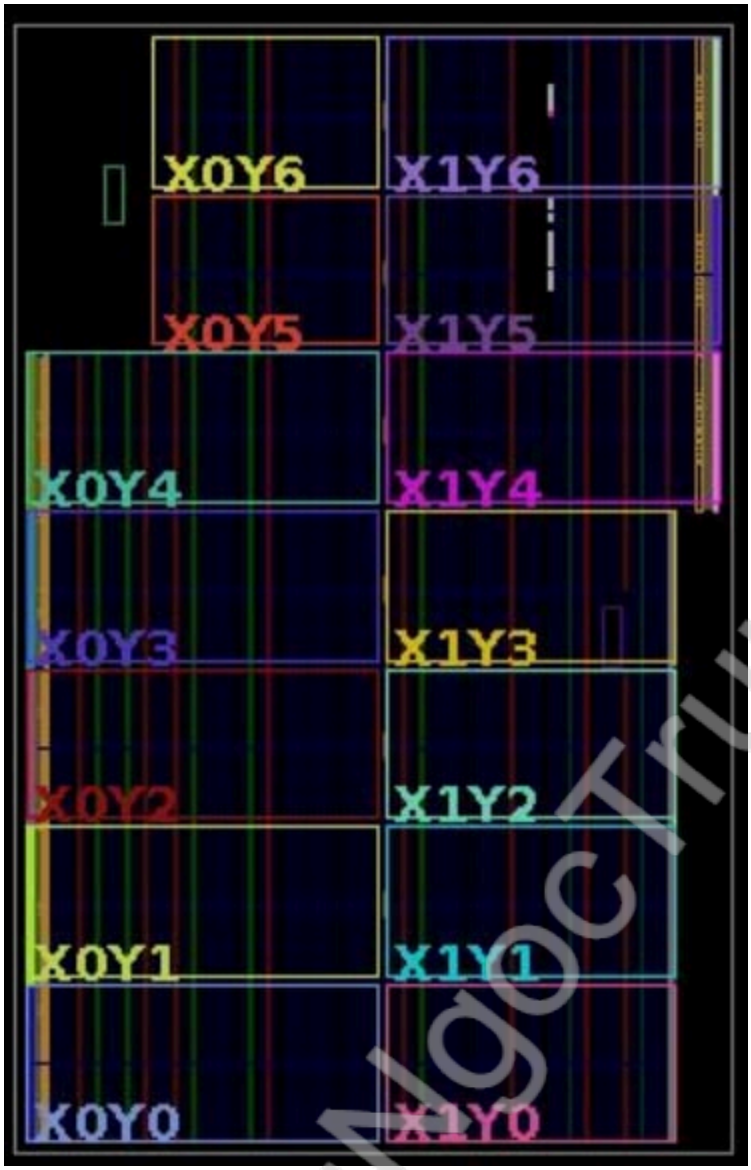
```
assert (ALU_Out_tb = x"19")
report "[FAIL] ADD: 10 + 15 != 25"
severity error;
```

Assert statements automatically verify the correctness of each operation. Any mismatch immediately stops the simulation and reports the failing case, making the testbench fully self-checking.

8. Synthesis



The ALU design was successfully synthesized using Vivado.
The RTL schematic confirms that the design is purely combinational and matches the intended ALU architecture.
The synthesized design occupies a small number of FPGA resources, indicating an efficient implementation.
No timing or synthesis errors were reported.



9. Conclusion

This project successfully designed, implemented, and verified an 8-bit parameterized ALU using VHDL. The ALU supports a wide range of arithmetic, logical, shift/rotate, and comparison operations and complies with IEEE standards.

Simulation results confirm correct functionality, and the design is suitable as a foundational datapath component for larger digital systems.

10. References

1. IEEE Standard VHDL Language Reference Manual.
2. Aschauer, F. **Hardware Description Languages & Verification - Lecture Slides.**