4 Bit Arithmetic Logical Unit

Subject: Digital Design using Verilog

Shashank Bagda
92100133020
6 TK1 - A

Overview

The project aims to design and implement a 4-bit Arithmetic Logical Unit (ALU) using Verilog. The ALU will be capable of performing basic arithmetic and logical operations such as addition, subtraction, multiplication, division, logical AND, logical OR, logical XOR, and logical NOT.

Description

The 4-bit ALU is composed of several functional units, including:

Arithmetic Unit:

Performs addition, subtraction, multiplication, and division operations on two 4-bit operands.

Logical Unit:

Performs logical operations such as AND, OR, XOR, and NOT on the operands.

The Verilog code defines modules for both the ALU and its testbench. The ALU module contains combinational logic for each operation, while the testbench module provides stimulus to test the functionality of the ALU.

Algorithm/Flowchart

The flowchart for the ALU operation is straightforward

1. Receive two 4-bit operands, A and B.
2. Perform the desired operation (addition, subtraction, multiplication, division, logical AND, logical OR, logical XOR, or logical NOT) on the operands.
3. Output the result of the operation.

The Flowchart of the Verilog Code

```
Start
Input A (4 bits)
Input B (4 bits)
v
| Addition
|   Sum = A + B
v
| Subtraction
|   Diff = A - B
v
| Multiplication
|   Product = A * B
v
| Division
|   Quotient = A / B
v
v  Logical Operations
+-| AND
  |   And = A & B
  v
+-| OR
  |   Or = A | B
  v
+-| XOR
  |   Xor = A ^ B
  v
v  Not Operations
```

```
+-| Not A
  |   not_a = ~A
  v
+-| Not B
  |   not_b = ~B
v
Output Sum (8 bits)
Output Diff (8 bits)
Output Product (8 bits)
Output Quotient (8 bits)
Output And (4 bits)
Output Or (4 bits)
Output Xor (4 bits)
Output Not A (4 bits)
Output Not B (4 bits)
End
```

This flowchart shows the basic structure of the code. The code first takes two 4-bit inputs (A and B). Then, it performs various operations on these inputs, including addition, subtraction, multiplication, division, logical AND, logical OR, logical XOR, and bitwise NOT. Finally, the code outputs the results of these operations.

ASM Chart of Addition

Let's create an ASM chart for adding two 4-bit values, even though the original Verilog code utilizes a dataflow approach:

Assumptions:

- The instruction set architecture (ISA) is a simple one with basic arithmetic instructions.
- Registers exist to store operands and the result.
- An instruction cycle consists of fetching, decoding, and executing instructions.

Registers:

- A: Holds the first 4-bit value
- B: Holds the second 4-bit value
- Sum: Stores the 8-bit result (may require handling carry)

Instructions:

- LOAD A, value: Loads a value into register A
- LOAD B, value: Loads a value into register B
- ADD A, B: Adds the contents of B to A, storing the result in A
- STORE Sum, value: Stores a value from a register into memory

Explanation:

1. Start: The process begins.
2. LOAD A, value_A: The first 4-bit value is loaded into register A.
3. LOAD B, value_B: The second 4-bit value is loaded into register B.
4. ADD A, B: The contents of B are added to the contents of A. The result is stored back in A. (Depending on the ISA, carry handling might be necessary.)
5. STORE Sum, A: The 8-bit result (potentially including the carry) from register A is stored in the memory location designated as "Sum."
6. End: The addition process is complete.

ASM Chart:

```
+-------------------+
|      Start       |
+-------------------+
         |
         v
+-------------------+
|  LOAD A, value_A  | (Load first operand)
+-------------------+
         |
         v
```

```
+------------------+
|  LOAD B, value_B  | (Load second operand)
+------------------+
         |
         v
+------------------+
|      ADD A, B     | (Add B to A)
+------------------+
         |
         v
+------------------+
|  STORE Sum, A     | (Store result in Sum)  // Might require carry handling
+------------------+
         |
         v
+------------------+
|      End        |
+------------------+
```

Considerations:

- This is a generic ASM chart and may need adjustments based on the specific ISA and carry handling mechanisms.
- If the addition results in a value exceeding 15 (overflow), additional instructions might be needed to handle it appropriately.

Alternative (Dataflow Approach):

While this ASM chart provides a step-by-step approach, the original Verilog code utilizes a dataflow approach, which means the addition happens concurrently when the code is executed. There's no explicit loop or sequence of instructions. The hardware performs the addition operation directly based on the logic gates within the circuit.

ASM Chart of Subtraction

Following the same approach as for addition, here's the ASM chart for subtracting two 4-bit values:

Assumptions:

- Same ISA and register setup as the addition chart.
- Instructions might differ for subtraction depending on the ISA.

Registers:

- A: Holds the first 4-bit value (minuend)
- B: Holds the second 4-bit value (subtrahend)
- Diff: Stores the 8-bit result (may require handling borrow)

Instructions:

- Same as addition chart, potentially with different subtraction instruction (e.g., SUB instead of ADD)

Explanation:

1. Start: The process begins.
2. LOAD A, value_A: The first 4-bit value (minuend) is loaded into register A.
3. LOAD B, value_B: The second 4-bit value (subtrahend) is loaded into register B.
4. SUB A, B: The contents of B are subtracted from the contents of A. The result is stored back in A. (Depending on the ISA, borrow handling might be necessary.)
5. STORE Diff, A: The 8-bit result (potentially including the borrow) from register A is stored in the memory location designated as "Diff."
6. End: The subtraction process is complete.

ASM Chart:

```
+-------------------+
|     Start       |
+-------------------+
         |
         v
```

```
+-------------------+
|  LOAD A, value_A  | (Load minuend)
+-------------------+
        |
        v
+-------------------+
|  LOAD B, value_B  | (Load subtrahend)
+-------------------+
        |
        v
+-------------------+
|     SUB A, B      | (Subtract B from A)
+-------------------+
        |
        v (Might require borrow handling)
+-------------------+
|  STORE Diff, A    | (Store result in Diff)
+-------------------+
        |
        v
+-------------------+
|      End          |
+-------------------+
```

Considerations:

- This is a generic ASM chart and may need adjustments based on the specific ISA and borrow handling mechanisms.
- If the subtraction results in a negative value (underflow), additional instructions might be needed to handle it appropriately.

Alternative (Dataflow Approach):

Similar to addition, the original Verilog code likely uses a dataflow approach for subtraction, meaning the hardware performs the operation directly based on the logic gates.

ASM Chart of Multiplication

While the provided Verilog code uses a dataflow approach for multiplication, here's an ASM chart for multiplying two 4-bit values using a basic booth multiplication algorithm:

Assumptions:

- The ISA has basic arithmetic and shift instructions.
- Registers exist to store operands and partial products.
- An instruction cycle consists of fetching, decoding, and executing instructions.

Registers:

- A: Holds the first 4-bit multiplier (multiplicand)
- B: Holds the second 4-bit multiplier (multiplier)
- Product: Stores the final 8-bit product (may require handling overflow)
- Temp: Temporary register for intermediate calculations

Instructions:

- LOAD A, value_A: Loads a value into register A
- LOAD B, value_B: Loads a value into register B
- SHR B, 1: Shifts the contents of B right by 1 bit
- ADD A, A: Shifts A left by 1 bit (multiplication by 2) if the least significant bit (LSB) of B is 1
- SUB A, A: Shifts A left by 1 bit (multiplication by 2) if the LSB of B is 0 and the second least significant bit (SLSB) is 1
- STORE Product, Temp: Stores the partial product in a temporary location
- LOOP: Repeat steps SHR B, 1, ADD/SUB A, A, and STORE Product until B becomes zero

ASM Chart:

```
+-------------------+
|      Start        |
+-------------------+
          |
          v
+-------------------+
|  LOAD A, value_A  | (Load multiplicand)
+-------------------+
          |
          v
+-------------------+
|  LOAD B, value_B  | (Load multiplier)
+-------------------+
          |
          v
+-------------------+
|  CLEAR Temp       | (Initialize temporary register to 0)
+-------------------+
          |
          v
+-------------------+
|  SHR B, 1         | (Shift B right by 1)
+-------------------+
          |
          v
Loop:
+-------------------+
|  Is B zero? (NZ)   | (Check if B is zero)
+------------------+
         |   Yes (Exit)
         |   v
         | No (Continue)
         |   v
+------------------+
|  Test LSB of B     | (Check LSB of B)
+------------------+
         |   1 (Add)
         |   v
         |   0 (Sub) (check SLSB)
         |   v
+------------------+
|  ADD A, A          | (Shift A left by 1) - Multiplication by 2
+------------------+
         |   v (if LSB is 1)
+------------------+
|  SUB A, A          | (Shift A left by 1) - Multiplication by 2 (if LSB is 0 and SLSB is 1)
+------------------+
         |   v
+------------------+
|  STORE Product,Temp| (Store partial product)
+------------------+
         |
         v
         |
         v
+------------------+
|  Jump to Loop      | (Repeat loop)
+--------------------+
         | (Exit)
         v
+--------------------+
|      End           |
+--------------------+
```

Explanation:

1. Start: The process begins.
2. LOAD A, value_A: The first 4-bit value (multiplicand) is loaded into register A.
3. LOAD B, value_B: The second 4-bit value (multiplier) is loaded into register B.
4. CLEAR Temp: The temporary register "Temp" is cleared to 0.
5. SHR B, 1: The contents of B are shifted right by 1 bit.
6. Loop: This label marks the beginning of the loop.
7. Is B zero? (NZ): A conditional jump checks if B is zero. If so, the loop exits. Otherwise, it continues.
8. Test LSB of B: The least significant bit (LSB) of B is checked.
9. ADD A, A: If the LSB of B is 1, A is shifted left by 1 bit (effectively multiplying by 2). This is achieved by adding A to itself.
10. SUB A, A: If the LSB of B is 0 and the second least significant bit (SLSB) is 1, A is shifted left by 1 bit. This is achieved by subtracting A from itself (effectively inverting and adding 1).
11. STORE Product, Temp: The partial

ASM Chart of Division

Division of unsigned integers using a hardware divider is a complex operation. Here's a simplified approach using an iterative restoring division algorithm, suitable for an ASM chart, but keep in mind it might not be the most efficient implementation:

Assumptions:

- The ISA has basic arithmetic, shift, and comparison instructions.
- Registers exist to hold operands, quotient, and remainder.
- An instruction cycle consists of fetching, decoding, and executing instructions.

Registers:

- A: Holds the dividend (4 bits)
- B: Holds the divisor (constant 4-bit value, non-zero) // Treat division by zero as an error
- Q: Stores the final quotient (4 bits)
- R: Stores the final remainder (4 bits)
- Temp: Temporary register for intermediate calculations

Instructions:

- LOAD A, value_A: Loads a value into register A
- LOAD B, value_B: Loads a value into register B (divisor)
- SHR A, 1: Shifts the dividend right by 1 bit
- ADD A, A: Shifts A left by 1 bit (effectively multiplying by 2)
- SUB A, B: Subtracts the divisor from the shifted dividend
- SGE Q, Temp: Checks if the result of subtraction (in Temp) is signed greater than or equal to zero (Temp >= 0)
- ADD A, B: Adds the divisor back if the subtraction result was non-negative (represents borrowing a bit back)
- SHR Q, 1: Shifts the quotient right by 1 bit (tentative quotient bit)
- LOOP: Repeat steps SHR A, 1, ADD/SUB A, B, SGE, ADD, SHR until all bits of A are processed

ASM Chart:

```
+-------------------+
|      Start        |
+-------------------+
         |
         v
+-------------------+
|  LOAD A, value_A  | (Load dividend)
+-------------------+
         |
         v
+-------------------+
|  LOAD B, value_B  | (Load divisor - constant, non-zero)
+-------------------+
         |
         v
+-------------------+
|  CLEAR Q, R       | (Initialize quotient and remainder to 0)
+-------------------+
         |
         v
+-------------------+
|  SHR A, 1         | (Shift dividend right by 1)
+-------------------+
         |
         v
Loop:
+-------------------+
|  Is it A zero? (Z)    | (Check if all bits of A are processed)
```

```
+-------------------+
        |   Yes (Exit)
        |   v
        | No (Continue)
        |   v
+-------------------+
|  ADD A, A         | (Shift A left by 1)
+-------------------+
        |
        v
+-------------------+
|  SUB A, B         | (Subtract divisor)
+-------------------+
        |
        v
+-------------------+
|  SGE Q, Temp      | (Check if the result is non-negative)
+-------------------+
        |   Yes (>= 0)
        |   v
        |   No (< 0)
        |   v
+-------------------+
|  ADD A, B         | (Add divisor back)
+-------------------+
        |   v (if >= 0)
+-------------------+
|  SHR Q, 1         | (Shift tentative quotient right by 1)
+-------------------+
        |
        v
+------------------+
|  Jump to Loop      | (Repeat loop)
+------------------+
        | (Exit)
        v
+------------------+
|  STORE Q, quotient | (Store final quotient)
+------------------+
        |
        v
+------------------+
|  STORE R, remainder| (Store final remainder)
+------------------+
        |
        v
+------------------+
|      End         |
+------------------+
```

Explanation:

1. Start: The process begins.
2. LOAD A, value_A: The dividend (4 bits) is loaded into register A.
3. LOAD B, value_B: The divisor (constant 4-bit non-zero value) is loaded into register B.
4. CLEAR Q, R: The quotient (Q) and remainder (R) registers are cleared to 0.
5. SHR A, 1: The dividend is shifted right by 1 bit, effectively dividing it by 2.
6. Loop: This label marks the beginning of the loop that iterates for each bit of the dividend.
7. Is A Zero? (Z): A conditional jump checks if all bits

ASM Chart of Logical AND

The original Verilog code utilizes a dataflow approach for logical operations, but here's an ASM chart for performing a bitwise AND operation on two 4-bit values:

Assumptions:

- The ISA has basic bitwise AND instruction.
- Registers exist to store operands and the result.
- An instruction cycle consists of fetching, decoding, and executing instructions.

Registers:

- A: Holds the first 4-bit value
- B: Holds the second 4-bit value
- And: Stores the 4-bit result of the AND operation

Instructions:

- LOAD A, value_A: Loads a value into register A
- LOAD B, value_B: Loads a value into register B
- AND A, B: Performs a bitwise AND operation on A and B, storing the result in A

ASM Chart:

```
+------------------+
|     Start       |
+------------------+
         |
         v
+------------------+
|  LOAD A, value_A  | (Load first operand)
+------------------+
         |
         v
+------------------+
|  LOAD B, value_B  | (Load second operand)
```

```
+-------------------+
         |
         v
+-------------------+
|       AND A, B      | (Perform bitwise AND)
+-------------------+
         |
         v
+-------------------+
|  STORE And, A     | (Store result in And)
+-------------------+
         |
         v
+-------------------+
|       End        |
+-------------------+
```

Explanation:

1. Start: The process begins.
2. LOAD A, value_A: The first 4-bit value is loaded into register A.
3. LOAD B, value_B: The second 4-bit value is loaded into register B.
4. AND A, B: The contents of A and B are bitwise ANDed together. The result is stored back in register A.
5. STORE And, A: The 4-bit AND result from register A is stored in the memory location designated as "And."
6. End: The AND operation is complete.

Alternative (Dataflow Approach):

As mentioned before, the original Verilog code likely uses a dataflow approach for the AND operation. This means the hardware performs the bitwise AND directly on the corresponding bits of A and B based on the logic gates within the circuit. There's no explicit loop or sequence of instructions for this operation in the code itself.

ASM Chart of Logical OR

Here's the ASM chart for performing a bitwise OR operation on two 4-bit values:

Assumptions:

- The ISA has a basic bitwise OR instruction.
- Registers exist to store operands and the result.
- An instruction cycle consists of fetching, decoding, and executing instructions.

Registers:

- A: Holds the first 4-bit value
- B: Holds the second 4-bit value
- Or: Stores the 4-bit result of the OR operation

Instructions:

- LOAD A, value_A: Loads a value into register A
- LOAD B, value_B: Loads a value into register B
- OR A, B: Performs a bitwise OR operation on A and B, storing the result in A

ASM Chart:

```
+-------------------+
|      Start       |
+-------------------+
         |
         v
+-------------------+
|  LOAD A, value_A  | (Load first operand)
+-------------------+
         |
         v
+-------------------+
|  LOAD B, value_B  | (Load second operand)
+-------------------+
          |
          v
+-------------------+
|      OR A, B      | (Perform bitwise OR)
+-------------------+
          |
          v
+-------------------+
|  STORE Or, A      | (Store result in Or)
+-------------------+
          |
          v
+-------------------+
|      End          |
+-------------------+
```

Explanation:

1. Start: The process begins.
2. LOAD A, value_A: The first 4-bit value is loaded into register A.
3. LOAD B, value_B: The second 4-bit value is loaded into register B.
4. OR A, B: The contents of A and B are bitwise ORed together. The result is stored back in register A.
5. STORE Or, A: The 4-bit OR result from register A is stored in the memory location designated as "Or."
6. End: The OR operation is complete.

Alternative (Dataflow Approach):

Similar to the AND operation, the original Verilog code likely uses a dataflow approach for the OR operation. In this case, the hardware performs the bitwise OR directly on the corresponding bits of A and B based on the logic gates within the circuit. There's no explicit loop or sequence of instructions for this operation in the Verilog code itself.

ASM Chart of Logical XOR

Following the same structure as the previous charts, here's the ASM chart for performing a bitwise XOR operation on two 4-bit values:

Assumptions:

- The ISA has a basic bitwise XOR instruction.
- Registers exist to store operands and the result.
- An instruction cycle consists of fetching, decoding, and executing instructions.

Registers:

- A: Holds the first 4-bit value
- B: Holds the second 4-bit value
- Xor: Stores the 4-bit result of the XOR operation

Instructions:

- LOAD A, value_A: Loads a value into register A
- LOAD B, value_B: Loads a value into register B
- XOR A, B: Performs a bitwise XOR operation on A and B, storing the result in A

ASM Chart:

```
+-------------------+
|      Start        |
+-------------------+
         |
         v
+-------------------+
|  LOAD A, value_A  | (Load first operand)
+-------------------+
         |
         v
+-------------------+
|  LOAD B, value_B  | (Load second operand)
+-------------------+
         |
         v
+-------------------+
|       XOR A, B      | (Perform bitwise XOR)
+-------------------+
         |
         v
+-------------------+
|  STORE Xor, A      | (Store result in Xor)
+-------------------+
         |
         v
+-------------------+
|       End        |
+-------------------+
```

Explanation:

1. Start: The process begins.
2. LOAD A, value_A: The first 4-bit value is loaded into register A.
3. LOAD B, value_B: The second 4-bit value is loaded into register B.
4. XOR A, B: The contents of A and B are bitwise XORed together. The result is stored back in register A.
5. STORE Xor, A: The 4-bit XOR result from register A is stored in the memory location designated as "Xor."
6. End: The XOR operation is complete.

Alternative (Dataflow Approach):

As with the AND and OR operations, the original Verilog code likely uses a dataflow approach for the XOR operation. The hardware performs the bitwise XOR directly on the corresponding bits of A and B based on the logic gates within the circuit. There's no explicit loop or sequence of instructions for this operation in the Verilog code itself.

ASM Chart of Logical NOT

The provided Verilog code uses a dataflow approach for bitwise NOT, so an ASM chart wouldn't be entirely suitable. However, here's an explanation and a possible approach using an ASM chart if we assume a more traditional instruction-based execution:

Explanation (Dataflow Approach):

In the original Verilog code, the not_a and not_b outputs are likely generated directly using logic gates that perform bitwise inversion on the corresponding bits of A and B. This happens concurrently whenever the code is executed. There's no explicit loop or sequence of instructions for this operation.

Possible ASM Chart Approach (Assuming Instruction-Based Execution):

If we consider an ISA that has a bitwise NOT instruction, here's a possible ASM chart:

Assumptions:

- The ISA has a bitwise NOT instruction (NOT).
- Registers exist to store the operand and the result.
- An instruction cycle consists of fetching, decoding, and executing instructions.

Registers:

- A: Holds the first 4-bit value
- B: Holds the second 4-bit value
- NotA: Stores the 4-bit result of NOT operation on A
- NotB: Stores the 4-bit result of NOT operation on B

Instructions:

- LOAD A, value_A: Loads a value into register A
- LOAD B, value_B: Loads a value into register B
- NOT A, A: Performs a bitwise NOT operation on A, storing the result in A (modifies A itself)
- STORE NotA, A: Stores the inverted value of A in NotA (since A was modified)
- NOT B, B: Performs a bitwise NOT operation on B, storing the result in B (modifies B itself)
- STORE NotB, B: Stores the inverted value of B in NotB (since B was modified)

ASM Chart:

```
+-------------------+
|      Start        |
+-------------------+
         |
         v
+-------------------+
|  LOAD A, value_A  | (Load first operand)
+-------------------+
         |
         v
+-------------------+
|  LOAD B, value_B  | (Load second operand)
+-------------------+
         |
         v
+-------------------+
|      NOT A, A     | (Invert A, store in A)
+-------------------+
         |
         v
+-------------------+
|  STORE NotA, A    | (Store inverted A in NotA)
+-------------------+
         |
         v
+-------------------+
|      NOT B, B     | (Invert B, store in B)
+-------------------+
        |
        v
+-------------------+
|  STORE NotB, B     | (Store inverted B in NotB)
+-------------------+
        |
        v
+-------------------+
|       End        |
+-------------------+
```

Important Note:

This approach modifies the original operands (A and B) themselves during the NOT operation. In some cases, it might be preferable to use a separate temporary register to hold the inverted value and then store it in the designated output register (NotA or NotB). This would avoid modifying the original operands. The choice depends on the specific ISA and desired behavior.

The original Verilog code's dataflow approach is generally more efficient for bitwise operations like NOT, as it performs the inversion directly on the hardware level without needing separate instructions and temporary registers.

Python Code - Main.py

```
import tkinter as tk
from PIL import Image, ImageTk
import subprocess

def write_to_file():
    # Get the input strings from the entry fields
    input1 = entry1.get()
    input2 = entry2.get()

    # Validate input strings
    if not is_valid_binary(input1) or not is_valid_binary(input2):
        output_text.delete(1.0, tk.END)
        output_text.insert(tk.END, "Invalid input. Please enter 4-bit binary numbers only.\n")
        return

    # Write the input strings to a file
    with open("a.txt", "w") as file:
        file.write(input1)
    with open("b.txt", "w") as file:
        file.write(input2)

    # Run the Verilog simulation and capture output
    result = subprocess.run("python code_generator.py && iverilog verilog.v && vvp a.out",
                            shell=True, capture_output=True, text=True)

    # Display the output in the text widget
    output_text.delete(1.0, tk.END)
    output_text.insert(tk.END, result.stdout)


def is_valid_binary(input_str):
    # Check if input string is a valid 4-bit binary number
    return len(input_str) == 4 and all(char in "01" for char in input_str)


# Create the main application window
root = tk.Tk()
root.title("4 Bit ALU")
root.geometry("615x460")

# Load background image
background_image = Image.open("background.jpg")
# Resize the image to fit the window size
background_image = background_image.resize((615, 460), Image.BILINEAR)
background_photo = ImageTk.PhotoImage(background_image)

# Create a label for the background image
background_label = tk.Label(root, image=background_photo)
background_label.place(x=0, y=0, relwidth=1, relheight=1)

# Create labels and entry fields for input strings
label1 = tk.Label(root, text="   Input Value A   ", font=("Helvetica", 12))
label1.grid(row=0, column=0, padx=5, pady=5, sticky="w")
entry1 = tk.Entry(root, font=("Helvetica", 12))
entry1.grid(row=0, column=1, padx=5, pady=5)
```

```
label2 = tk.Label(root, text="   Input Value B   ", font=("Helvetica", 12))
label2.grid(row=1, column=0, padx=5, pady=5, sticky="w")
entry2 = tk.Entry(root, font=("Helvetica", 12))
entry2.grid(row=1, column=1, padx=5, pady=5)

# Create a button to trigger writing to file
write_button = tk.Button(root, text="Submit", command=write_to_file, font=("Helvetica",
12))
write_button.grid(row=2, column=0, columnspan=2, padx=5, pady=10)

# Add a separator
separator = tk.Frame(height=2, bd=1, relief="sunken")
separator.grid(row=3, column=0, columnspan=2, sticky="we", padx=5, pady=5)

# Add a label for the output section
output_label = tk.Label(root, text="Simulation Output:", font=("Helvetica", 14, "bold"))
output_label.grid(row=4, column=0, columnspan=2, padx=5, pady=5)

# Add a text widget to display simulation output
output_text = tk.Text(root, width=60, height=15, font=("Courier", 12))
output_text.grid(row=5, column=0, columnspan=2, padx=5, pady=5)

# Run the application
root.mainloop()
```

Python Code - Code_Generator.py

```
# Python code to generate Verilog testbench with values read from files

# Read values from a.txt and b.txt
with open("a.txt", "r") as file:
    a_val = file.readline().strip()  # Read the first line and remove leading/trailing whitespace
with open("b.txt", "r") as file:
    b_val = file.readline().strip()  # Read the first line and remove leading/trailing whitespace

# print("value of A : " + str(a_val))
# print("value of B : " + str(b_val))

# Generate Verilog testbench code with the read values
verilog_code = f"""
module four_bit_operations(
    input [3:0] A,        // 4-bit input A
    input [3:0] B,        // 4-bit input B
    output [7:0] Sum,     // 4-bit output Sum for addition
    output [7:0] Diff,    // 4-bit output Diff for subtraction
    output [7:0] Product, // 4-bit output Product for multiplication
    output [7:0] Quotient,// 4-bit output Quotient for division
    output [3:0] And,     // 4-bit output And for logical AND
    output [3:0] Or,      // 4-bit output Or for logical OR
    output [3:0] Xor,     // 4-bit output Xor for logical XOR
    output [3:0] not_a,     // 4-bit output Not for A
    output [3:0] not_b      // 4-bit output Not for B
);
```

```
// Dataflow modeling for addition
assign Sum = A + B;

// Dataflow modeling for subtraction
assign Diff = A - B;

// Dataflow modeling for multiplication
assign Product = A * B;

// Dataflow modeling for division
assign Quotient = A / B;

// Dataflow modeling for logical AND
assign And = A & B;

// Dataflow modeling for logical OR
assign Or = A | B;

// Dataflow modeling for logical XOR
assign Xor = A ^ B;

// Dataflow modeling for logical NOT
assign not_a = ~A;
assign not_b = ~B;

endmodule
```

```
module testbench;

  // Define signals
  reg [3:0] A, B;          // 4-bit input signals
  wire [7:0] Sum, Diff, Product, Quotient; // 8-bit output signals
  wire [3:0] And, Or, Xor, not_a, not_b; // 4-bit output signals

  // Instantiate the module under test
  four_bit_operations UUT(
    .A(A),
    .B(B),
    .Sum(Sum),
    .Diff(Diff),
    .Product(Product),
    .Quotient(Quotient),
    .And(And),
    .Or(Or),
    .Xor(Xor),
    .not_a(not_a),
    .not_b(not_b)
  );

  // Stimulus generation
  initial begin
  $display("");
    // Assign values to signals
    A = 4'b{a_val};
    B = 4'b{b_val};
```

```
// Wait for some time for simulation to run
#10;

// Display output

$display("Value of A = %b", A);
$display("Value of B = %b", B);
$display("");

$display("Sum = %b", Sum);
$display("");

$display("Difference = %b", Diff);
$display("");

$display("Product = %b", Product);
$display("");

$display("Quotient = %b", Quotient);
$display("");

$display("AND = %b", And);
$display("");

$display("OR = %b", Or);
$display("");

$display("NOT of A = %b", not_a);
$display("NOT of B = %b", not_b);
```

```
    $display("");

    $display("XOR = %b", Xor);
    $display("");

    // End simulation
    $finish;
  end

endmodule
"""

# Write the generated Verilog code to a file
with open("verilog.v", "w") as file:
    file.write(verilog_code)
```

Verilog Code

```
module four_bit_operations(
    input [3:0] A,        // 4-bit input A
    input [3:0] B,        // 4-bit input B
    output [7:0] Sum,     // 4-bit output Sum for addition
    output [7:0] Diff,    // 4-bit output Diff for subtraction
    output [7:0] Product, // 4-bit output Product for multiplication
    output [7:0] Quotient,// 4-bit output Quotient for division
    output [3:0] And,     // 4-bit output And for logical AND
    output [3:0] Or,      // 4-bit output Or for logical OR
    output [3:0] Xor,     // 4-bit output Xor for logical XOR
    output [3:0] not_a,     // 4-bit output Not for A
    output [3:0] not_b      // 4-bit output Not for B
);

// Dataflow modeling for addition
assign Sum = A + B;

// Dataflow modeling for subtraction
assign Diff = A - B;

// Dataflow modeling for multiplication
assign Product = A * B;

// Dataflow modeling for division
assign Quotient = A / B;
```

```
// Dataflow modeling for logical AND
assign And = A & B;

// Dataflow modeling for logical OR
assign Or = A | B;

// Dataflow modeling for logical XOR
assign Xor = A ^ B;

// Dataflow modeling for logical NOT
assign not_a = ~A;
assign not_b = ~B;

endmodule
```

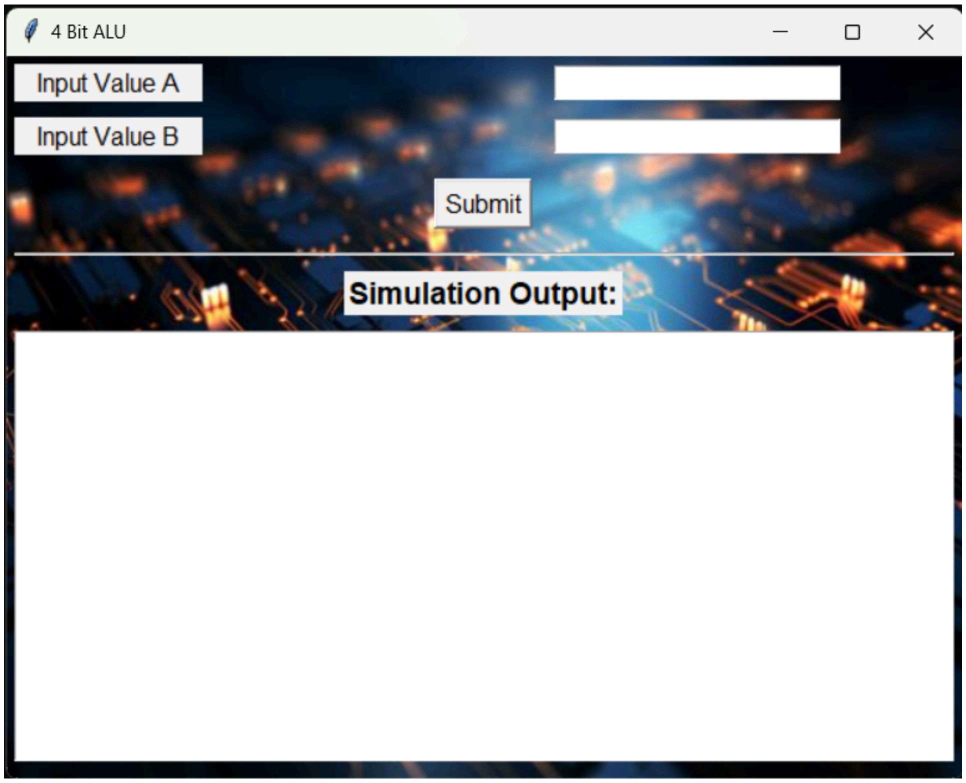
Testbench

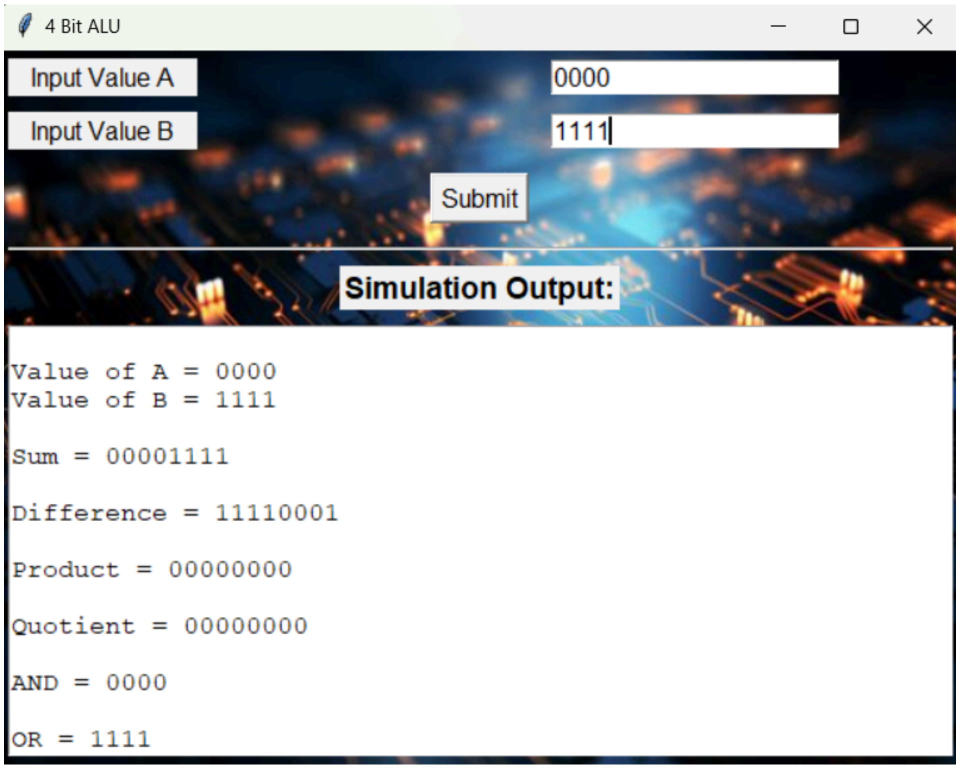
```
module testbench;

 // Define signals
 reg [3:0] A, B;        // 4-bit input signals
 wire [7:0] Sum, Diff, Product, Quotient; // 8-bit output signals
 wire [3:0] And, Or, Xor, not_a, not_b; // 4-bit output signals

 // Instantiate the module under test
 four_bit_operations UUT(
  .A(A),
  .B(B),
  .Sum(Sum),
  .Diff(Diff),
  .Product(Product),
  .Quotient(Quotient),
  .And(And),
  .Or(Or),
  .Xor(Xor),
  .not_a(not_a),
  .not_b(not_b)
 );

 // Stimulus generation
 initial begin
 $display("");
  // Assign values to signals
```

```
A = 4'b1111;
B = 4'b0000;

// Wait for some time for simulation to run
#10;

// Display output

$display("Value of A = %b", A);
$display("Value of B = %b", B);
$display("");

$display("Sum = %b", Sum);
$display("");

$display("Difference = %b", Diff);
$display("");

$display("Product = %b", Product);
$display("");

$display("Quotient = %b", Quotient);
$display("");

$display("AND = %b", And);
$display("");

$display("OR = %b", Or);
$display("");
```

```
    $display("NOT of A = %b", not_a);
    $display("NOT of B = %b", not_b);
    $display("");

    $display("XOR = %b", Xor);
    $display("");

    // End simulation
    $finish;
  end

endmodule
```

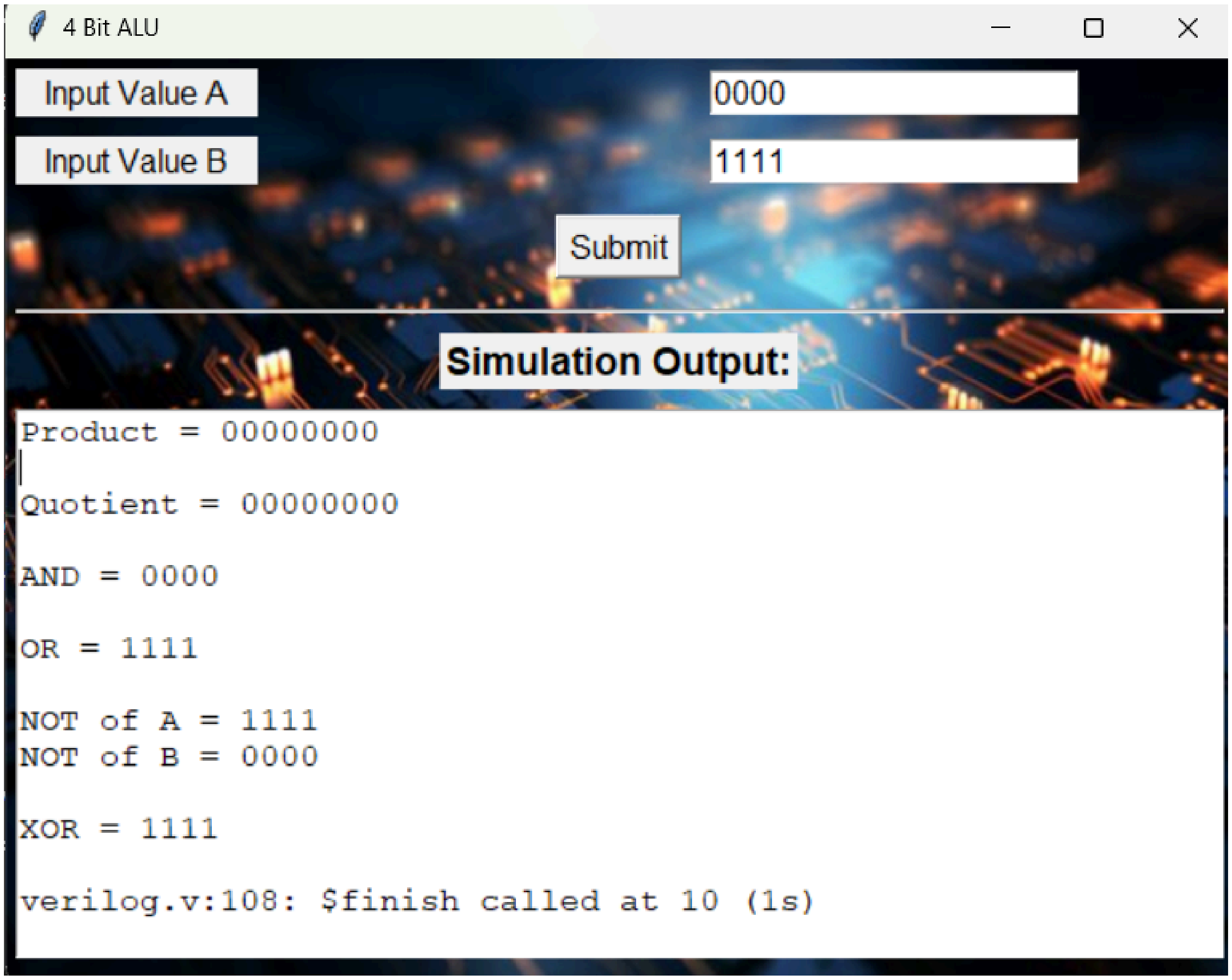
Output





0000
1111

```
Value of A = 0000
Value of B = 1111

Sum = 00001111

Difference = 11110001

Product = 00000000

Quotient = 00000000

AND = 0000

OR = 1111
```



```
Product = 00000000

Quotient = 00000000

AND = 0000

OR = 1111

NOT of A = 1111
NOT of B = 0000

XOR = 1111

verilog.v:108: $finish called at 10 (1s)
```

End of Documentation