Project Final Step: 4-bit ALU Functions

Group 1
Cristian Perez, Joel Sarmiento

July 29, 2025

1 Introduction

In this final step of our Verilog-based project, we designed and implemented a fully functional 4-bit Arithmetic Logic Unit (ALU) using basic logic and arithmetic building blocks developed in previous steps. The project began with the creation of individual logic gates (AND, OR, NOT, etc.) and arithmetic circuits (adder, subtractor, multiplier, divider, shifter), each of which was thoroughly tested using Verilog and GTKWave simulation.

Once verified, these components were integrated into a single ALU module capable of performing both logical and arithmetic operations based on control signals. A centralized control table defines which operation is executed, allowing flexible switching between functions. All simulations were conducted using Icarus Verilog and GTKWave to verify correctness and behavior.

This document includes the Verilog module code, testbenches, waveform screenshots, and control tables, culminating in a complete 4-bit ALU design. This project provides a foundational understanding of how ALUs operate in modern processors and how modular design in Verilog enables scalable digital systems.

2 Verilog Code, Testbenches, and Waveforms

2.1 AND Gate

Verilog Code

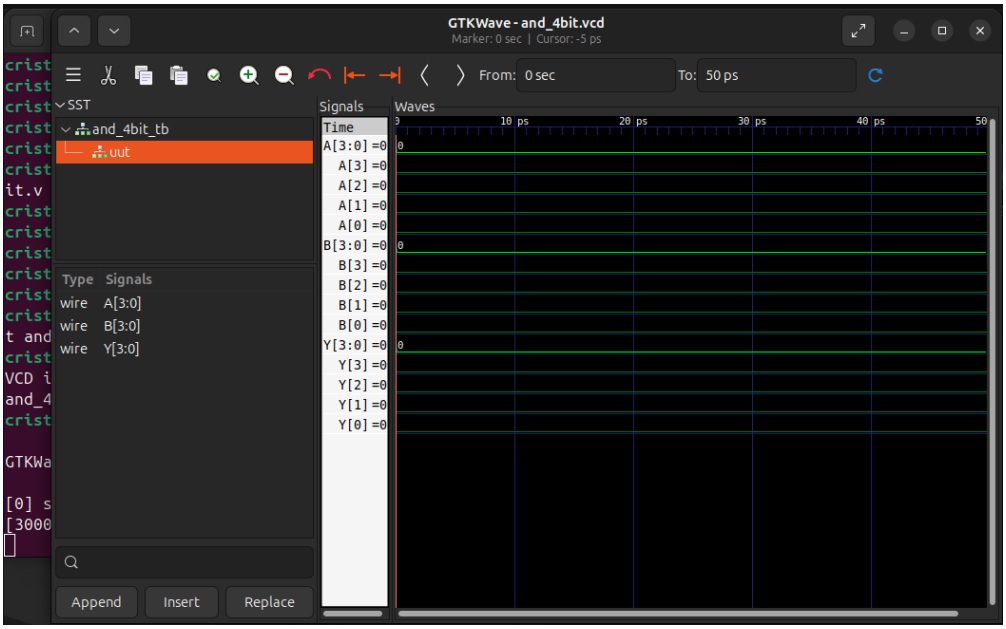
```
module and_4bit(input [3:0] A, B, output [3:0] Y);
    assign Y = A & B;
endmodule
```

Testbench

```
‘timescale 1ns/1ps
module and_4bit_tb;
    reg [3:0] A, B;
    wire [3:0] Y;
```

```
    and_4bit uut (.A(A), .B(B), .Y(Y));

    initial begin
        $dumpfile("and_4bit.vcd");
        $dumpvars(0, and_4bit_tb);

        A = 4'b0000; B = 4'b0000; #10;
        A = 4'b1010; B = 4'b1100; #10;
        A = 4'b1111; B = 4'b1111; #10;
        $finish;
    end
endmodule
```

Waveform



2.2 NAND Gate

Verilog Code

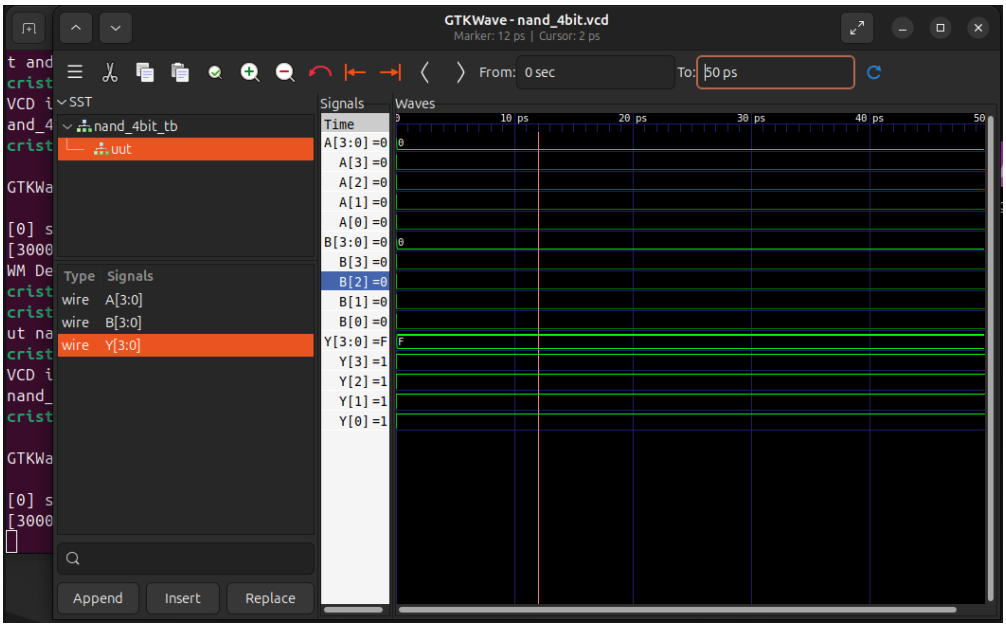
```
module nand_4bit(input [3:0] A, B, output [3:0] Y);
    assign Y = ~(A & B);
endmodule
```

Testbench

```
‘timescale 1ns/1ps
```

```
module nand_4bit_tb;
    reg [3:0] A, B;
    wire [3:0] Y;

    nand_4bit uut (.A(A), .B(B), .Y(Y));

    initial begin
        $dumpfile("nand_4bit.vcd");
        $dumpvars(0, nand_4bit_tb);

        A = 4'b0000; B = 4'b0000; #10;
        A = 4'b1010; B = 4'b1100; #10;
        A = 4'b1111; B = 4'b1111; #10;
        $finish;
    end
endmodule
```

Waveform



2.3 OR Gate

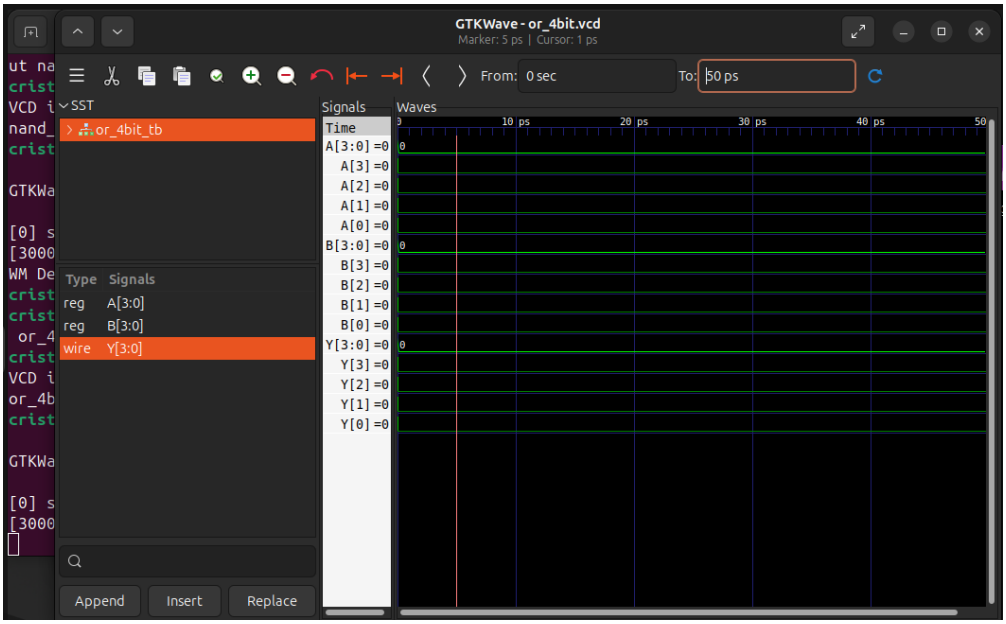
Verilog Code

```
module or_4bit(input [3:0] A, B, output [3:0] Y);
    assign Y = A | B;
endmodule
```

Testbench

```
‘timescale 1ns/1ps
module or_4bit_tb;
    reg [3:0] A, B;
    wire [3:0] Y;

    or_4bit uut (.A(A), .B(B), .Y(Y));

    initial begin
        $dumpfile("or_4bit.vcd");
        $dumpvars(0, or_4bit_tb);

        A = 4’b0000; B = 4’b0000; #10;
        A = 4’b1010; B = 4’b1100; #10;
        A = 4’b1111; B = 4’b1111; #10;
        $finish;
    end
endmodule
```

Waveform



2.4 NOR Gate

Verilog Code

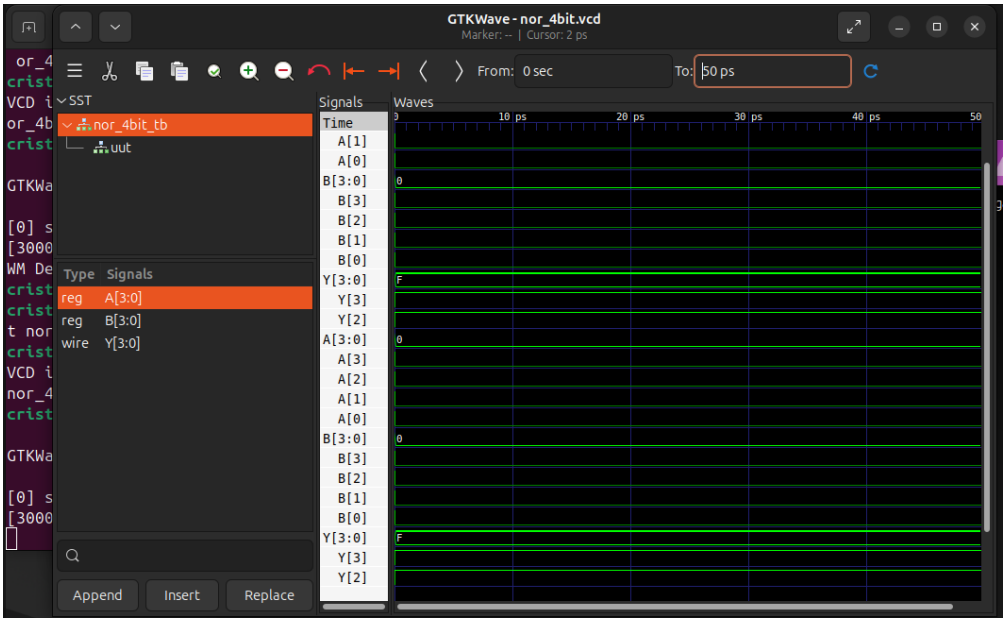
```
module nor_4bit(input [3:0] A, B, output [3:0] Y);
```

```
    assign Y = ~(A | B);
endmodule
```

Testbench

```
‘timescale 1ns/1ps
module nor_4bit_tb;
    reg [3:0] A, B;
    wire [3:0] Y;

    nor_4bit uut (.A(A), .B(B), .Y(Y));

    initial begin
        $dumpfile("nor_4bit.vcd");
        $dumpvars(0, nor_4bit_tb);

        A = 4’b0000; B = 4’b0000; #10;
        A = 4’b1010; B = 4’b1100; #10;
        A = 4’b1111; B = 4’b1111; #10;
        $finish;
    end
endmodule
```

Waveform



2.5 XOR Gate

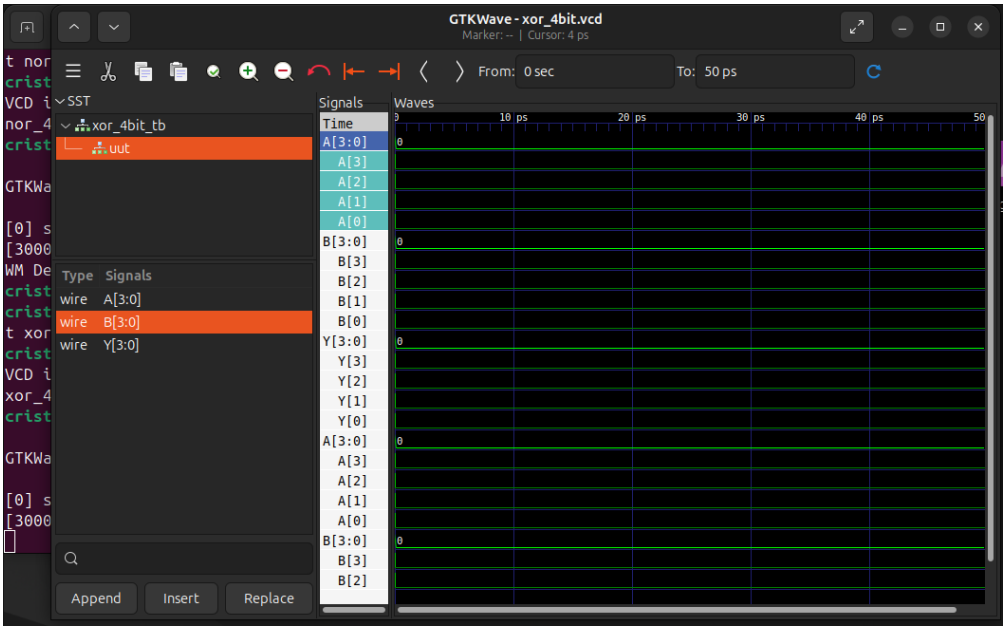
Verilog Code

```
module xor_4bit(input [3:0] A, B, output [3:0] Y);
    assign Y = A ^ B;
endmodule
```

Testbench

```
‘timescale 1ns/1ps
module xor_4bit_tb;
    reg [3:0] A, B;
    wire [3:0] Y;

    xor_4bit uut (.A(A), .B(B), .Y(Y));

    initial begin
        $dumpfile("xor_4bit.vcd");
        $dumpvars(0, xor_4bit_tb);

        A = 4’b0000; B = 4’b0000; #10;
        A = 4’b1010; B = 4’b1100; #10;
        A = 4’b1111; B = 4’b1111; #10;
        $finish;
    end
endmodule
```

Waveform



2.6 XNOR Gate

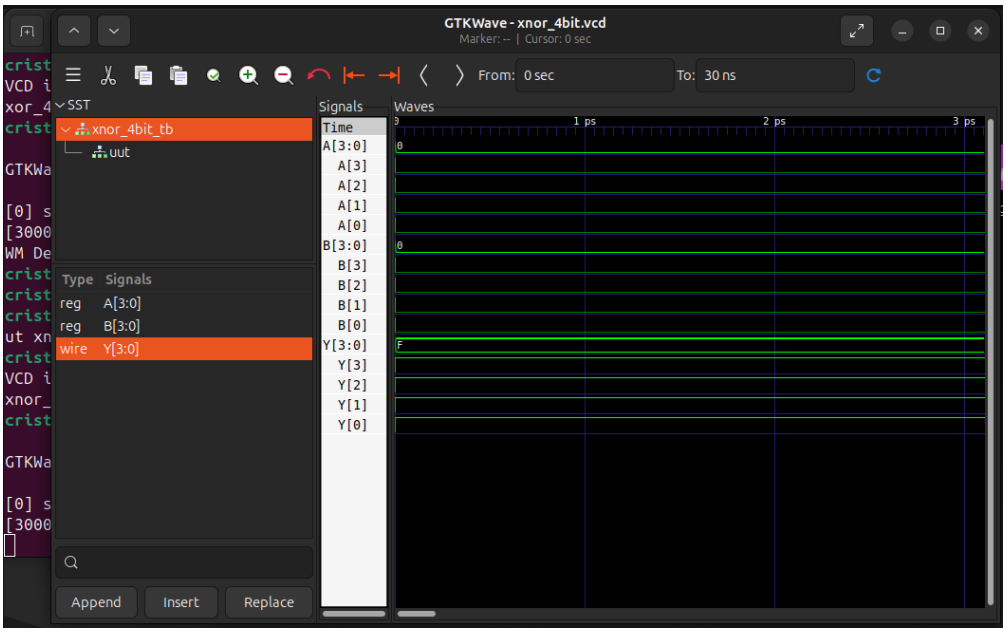
Verilog Code

```
module xnor_4bit(input [3:0] A, B, output [3:0] Y);
    assign Y = ~(A ^ B);
endmodule
```

Testbench

```
‘timescale 1ns/1ps
module xnor_4bit_tb;
    reg [3:0] A, B;
    wire [3:0] Y;

    xnor_4bit uut (.A(A), .B(B), .Y(Y));

    initial begin
        $dumpfile("xnor_4bit.vcd");
        $dumpvars(0, xnor_4bit_tb);

        A = 4’b0000; B = 4’b0000; #10;
        A = 4’b1010; B = 4’b1100; #10;
        A = 4’b1111; B = 4’b1111; #10;
        $finish;
    end
endmodule
```

Waveform



2.7 NOT Gate

Verilog Code

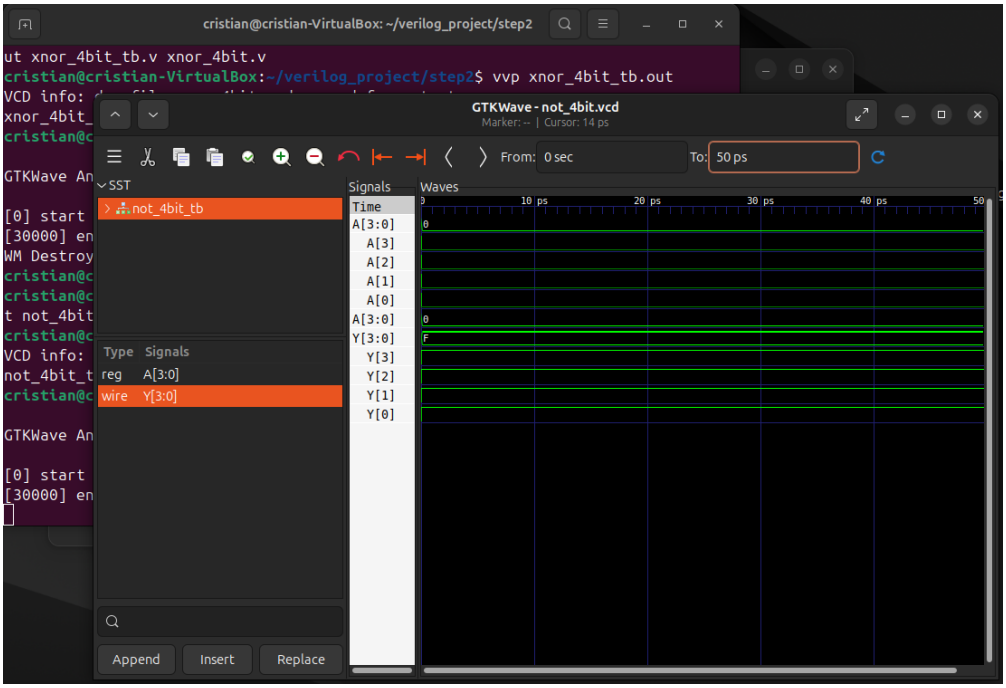
```
module not_4bit(input [3:0] A, output [3:0] Y);
    assign Y = ~A;
endmodule
```

Testbench

```
‘timescale 1ns/1ps
module not_4bit_tb;
    reg [3:0] A;
    wire [3:0] Y;

    not_4bit uut (.A(A), .Y(Y));

    initial begin
        $dumpfile("not_4bit.vcd");
        $dumpvars(0, not_4bit_tb);

        A = 4’b0000; #10;
        A = 4’b1010; #10;
        A = 4’b1111; #10;
        $finish;
```

```
    end
endmodule
```

Waveform



2.8 Arithmetic Shifter

Verilog Code

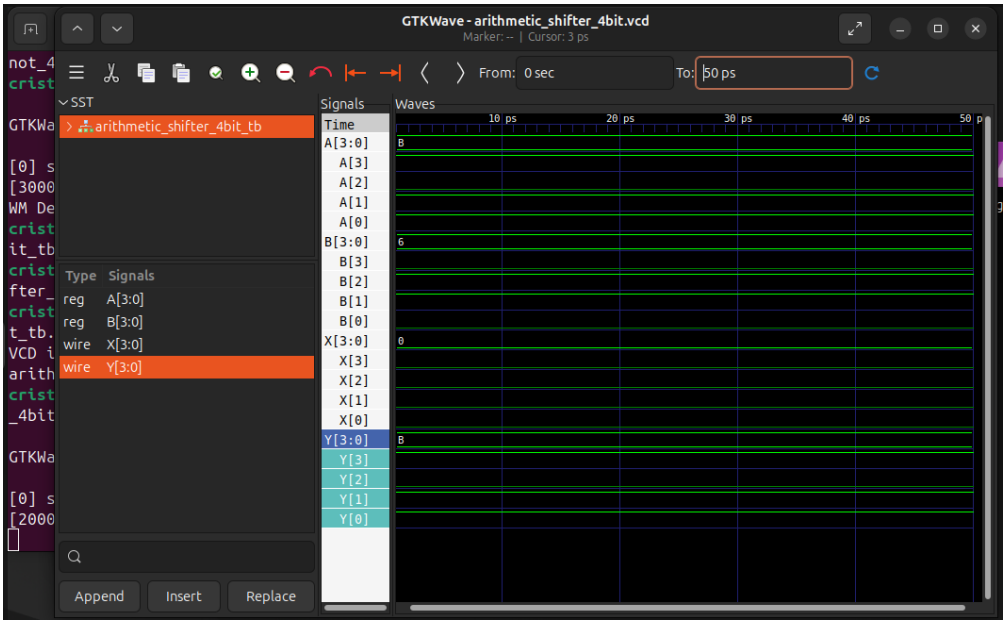
```
module arithmetic_shifter_4bit(
    input [3:0] A,
    input [3:0] B,
    output reg [3:0] X,
    output reg [3:0] Y
);
    always @(*) begin
        if (B[3] == 0)
            {Y, X} = {A, B[0]} << B[2:1];
        else
            {X, Y} = {B[0], A} >> B[2:1];
    end
endmodule
```

Testbench

```
‘timescale 1ns/1ps
```

```
module arithmetic_shifter_4bit_tb;
    reg [3:0] A, B;
    wire [3:0] X, Y;

    arithmetic_shifter_4bit uut (.A(A), .B(B), .X(X), .Y(Y));

    initial begin
        $dumpfile("arithmetic_shifter_4bit.vcd");
        $dumpvars(0, arithmetic_shifter_4bit_tb);

        A = 4'b1011; B = 4'b0110; #10;
        A = 4'b1101; B = 4'b1101; #10;
        $finish;
    end
endmodule
```

Waveform



2.9 Adder

Verilog Code

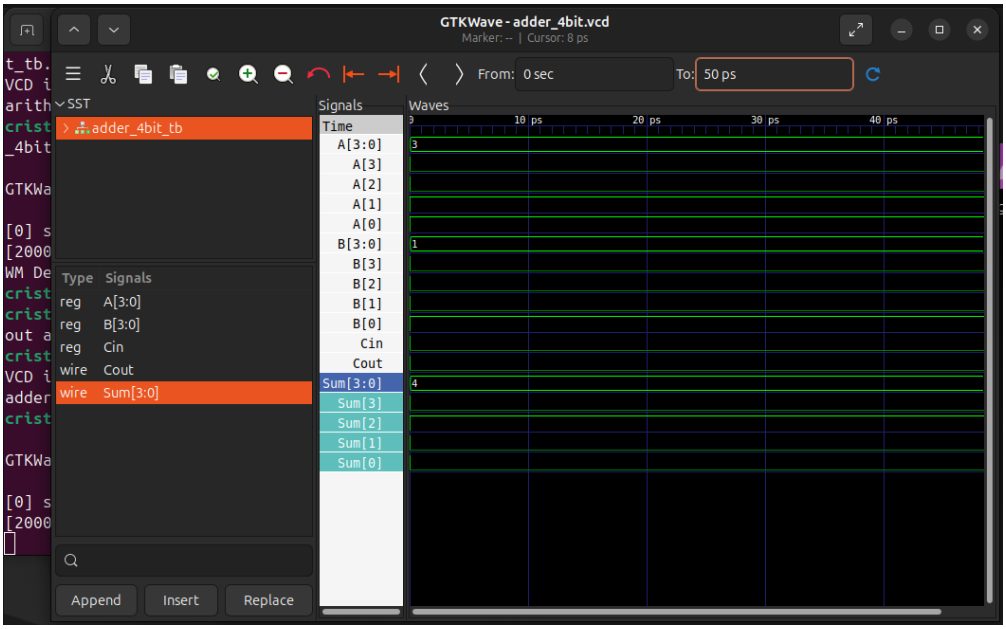
```
module adder_4bit(
    input [3:0] A, B,
    input Cin,
    output [3:0] Sum,
    output Cout
```

```
);
    assign {Cout, Sum} = A + B + Cin;
endmodule
```

Testbench

```
‘timescale 1ns/1ps
module adder_4bit_tb;
    reg [3:0] A, B;
    reg Cin;
    wire [3:0] Sum;
    wire Cout;

    adder_4bit uut (.A(A), .B(B), .Cin(Cin), .Sum(Sum), .Cout(Cout));

    initial begin
        $dumpfile("adder_4bit.vcd");
        $dumpvars(0, adder_4bit_tb);

        A = 4’b0011; B = 4’b0001; Cin = 0; #10;
        A = 4’b1111; B = 4’b0001; Cin = 1; #10;
        $finish;
    end
endmodule
```

Waveform



2.10 Subtractor

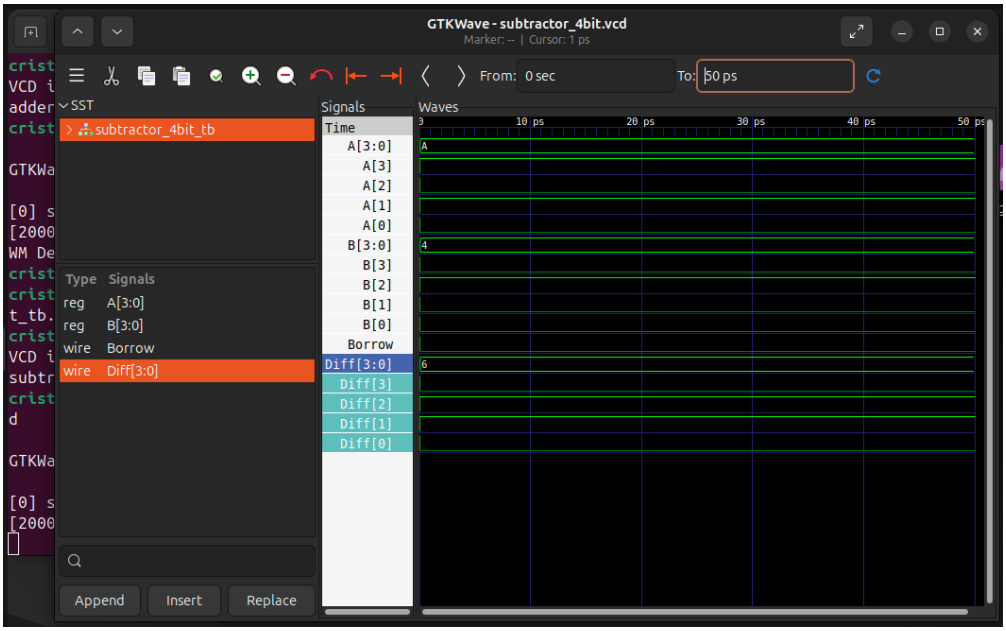
Verilog Code

```
module subtractor_4bit(
    input [3:0] A, B,
    output [3:0] Diff,
    output Borrow
);
    assign {Borrow, Diff} = {1'b0, A} - B;
endmodule
```

Testbench

```
`timescale 1ns/1ps
module subtractor_4bit_tb;
    reg [3:0] A, B;
    wire [3:0] Diff;
    wire Borrow;

    subtractor_4bit uut (.A(A), .B(B), .Diff(Diff), .Borrow(Borrow));

    initial begin
        $dumpfile("subtractor_4bit.vcd");
        $dumpvars(0, subtractor_4bit_tb);

        A = 4'b1010; B = 4'b0100; #10;
        A = 4'b0011; B = 4'b0100; #10;
        $finish;
    end
endmodule
```

Waveform



2.11 Multiplier

Verilog Code

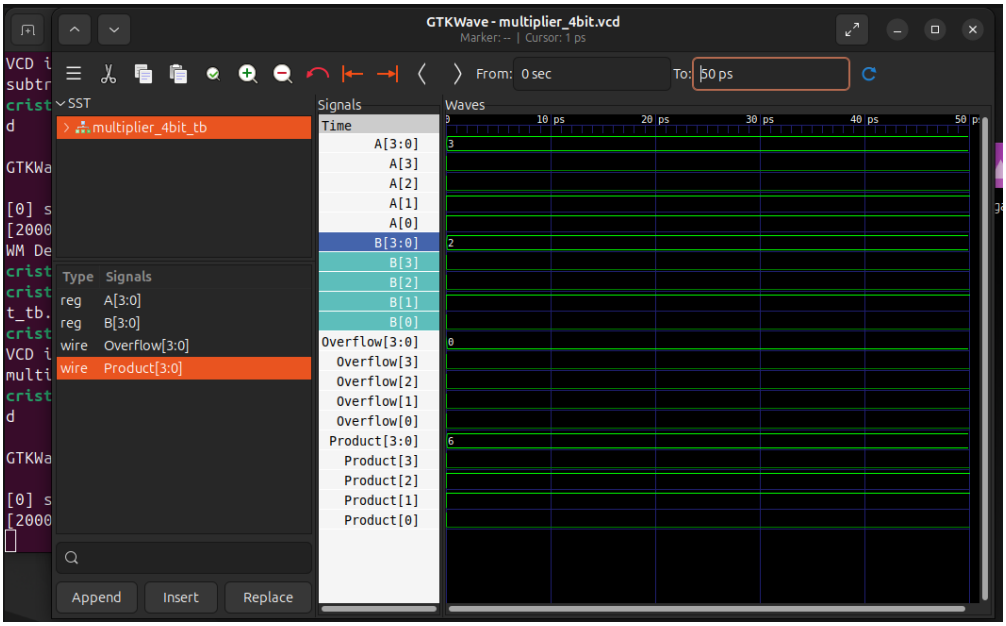
```
module multiplier_4bit(
    input [3:0] A, B,
    output [3:0] Product,
    output [3:0] Overflow
);
    wire [7:0] full_product;
    assign full_product = A * B;
    assign Product = full_product[3:0];
    assign Overflow = full_product[7:4];
endmodule
```

Testbench

```
‘timescale 1ns/1ps
module multiplier_4bit_tb;
    reg [3:0] A, B;
    wire [3:0] Product, Overflow;

    multiplier_4bit uut (.A(A), .B(B), .Product(Product), .Overflow(Overflow));

    initial begin
        $dumpfile("multiplier_4bit.vcd");
        $dumpvars(0, multiplier_4bit_tb);
```

```
        A = 4'b0011; B = 4'b0010; #10;
        A = 4'b1111; B = 4'b1111; #10;
        $finish;
    end
endmodule
```

Waveform



2.12 Divider

Verilog Code

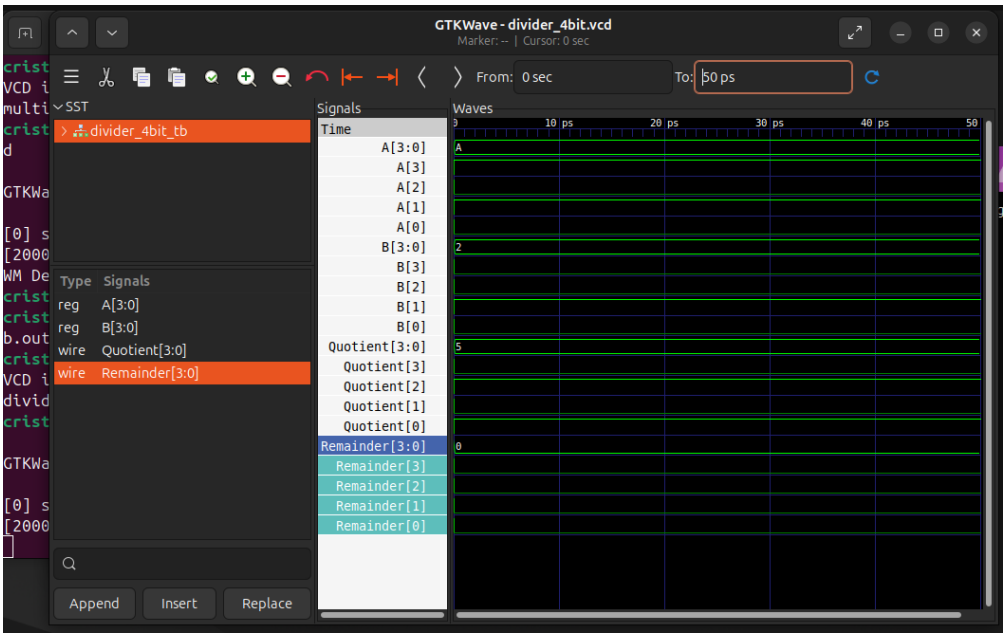
```
module divider_4bit(
    input [3:0] A, B,
    output [3:0] Quotient,
    output [3:0] Remainder
);
    assign Quotient = (B != 0) ? (A / B) : 4'b0000;
    assign Remainder = (B != 0) ? (A % B) : 4'b0000;
endmodule
```

Testbench

```
`timescale 1ns/1ps
module divider_4bit_tb;
    reg [3:0] A, B;
    wire [3:0] Quotient, Remainder;
```

```
    divider_4bit uut (.A(A), .B(B), .Quotient(Quotient), .Remainder(Remainder));

    initial begin
        $dumpfile("divider_4bit.vcd");
        $dumpvars(0, divider_4bit_tb);

        A = 4'b1010; B = 4'b0010; #10;
        A = 4'b1111; B = 4'b0100; #10;
        $finish;
    end
endmodule
```

Waveform



3 Operation Control Tables

Logic Operations Control Table

Operation	Control Signal	Function	Example (A=1010, B=1100)
AND	000	A AND B	1000
NAND	001	NOT (A AND B)	0111
OR	010	A OR B	1110
NOR	011	NOT (A OR B)	0001
XOR	100	A XOR B	0110
XNOR	101	NOT (A XOR B)	1001
NOT	110	NOT A	0101

Table 1: Logic Operations Control Table

Arithmetic Operations Control Table

Operation	Control Signal	Function	Example (A=1010, B=0011)
Addition	000	A + B	1101 (Carry=0)
Subtraction	001	A - B	0111 (Borrow=0)
Multiplication	010	A × B	Product=11110 (Overflow=1)
Division	011	A / B	Quotient=0011, Remainder=0001
Arithmetic Shift	100	Shift based on B	Output depends on B control bits

Table 2: Arithmetic Operations Control Table

4 Final ALU Integration

Verilog Code

```
module alu_4bit (
    input [3:0] A, B,
    input [2:0] control,
    input Cin,
    output reg [3:0] Result,
    output reg Cout,
    output reg Overflow,
    output reg [3:0] Remainder
);
    wire [3:0] and_out, nand_out, or_out, nor_out, xor_out, xnor_out, not_out;
    wire [3:0] sum, diff, prod, shift_x, shift_y, quot, rem;
    wire sum_cout, diff_borrow;
    wire [3:0] mult_ovf;

    and_4bit u_and (.A(A), .B(B), .Y(and_out));
    nand_4bit u_nand (.A(A), .B(B), .Y(nand_out));
    or_4bit u_or (.A(A), .B(B), .Y(or_out));
    nor_4bit u_nor (.A(A), .B(B), .Y(nor_out));
    xor_4bit u_xor (.A(A), .B(B), .Y(xor_out));
    xnor_4bit u_xnor (.A(A), .B(B), .Y(xnor_out));
    not_4bit u_not (.A(A), .Y(not_out));

    adder_4bit u_add (.A(A), .B(B), .Cin(Cin), .Sum(sum), .Cout(sum_cout));
    subtractor_4bit u_sub (.A(A), .B(B), .Diff(diff), .Borrow(diff_borrow));
    multiplier_4bit u_mult (.A(A), .B(B), .Product(prod), .Overflow(mult_ovf));
    divider_4bit u_div (.A(A), .B(B), .Quotient(quot), .Remainder(rem));
    arithmetic_shifter_4bit u_shift (.A(A), .B(B), .X(shift_x), .Y(shift_y));

    always @(*) begin
        case (control)
            3'b000: begin Result = sum; Cout = sum_cout; Overflow = 0; Remainder = 0; en
            3'b001: begin Result = diff; Cout = 0; Overflow = diff_borrow; Remainder = 0
            3'b010: begin Result = prod; Cout = 0; Overflow = |mult_ovf; Remainder = 0;
            3'b011: begin Result = quot; Cout = 0; Overflow = 0; Remainder = rem; end
            3'b100: begin Result = shift_x; Cout = 0; Overflow = 0; Remainder = shift_y;
            3'b101: begin Result = and_out; Cout = 0; Overflow = 0; Remainder = 0; end
            3'b110: begin Result = or_out; Cout = 0; Overflow = 0; Remainder = 0; end
            3'b111: begin Result = not_out; Cout = 0; Overflow = 0; Remainder = 0; end
            default: begin Result = 4'b0000; Cout = 0; Overflow = 0; Remainder = 0; end
        endcase
    end
```

```
endmodule
```

Testbench

```
‘timescale 1ns/1ps
module alu_4bit_tb;
    reg [3:0] A, B;
    reg [2:0] control;
    reg Cin;
    wire [3:0] Result;
    wire Cout, Overflow;
    wire [3:0] Remainder;

    alu_4bit uut (.A(A), .B(B), .control(control), .Cin(Cin), .Result(Result), .Cout(Cou

    initial begin
        $dumpfile("alu_4bit.vcd");
        $dumpvars(0, alu_4bit_tb);

        A = 4’b1010; B = 4’b0011; Cin = 0;

        control = 3’b000; #10; // ADD
        control = 3’b001; #10; // SUB
        control = 3’b010; #10; // MUL
        control = 3’b011; #10; // DIV
        control = 3’b100; #10; // SHF
        control = 3’b101; #10; // AND
        control = 3’b110; #10; // OR
        control = 3’b111; #10; // NOT

        $finish;
    end
endmodule
```

Waveform

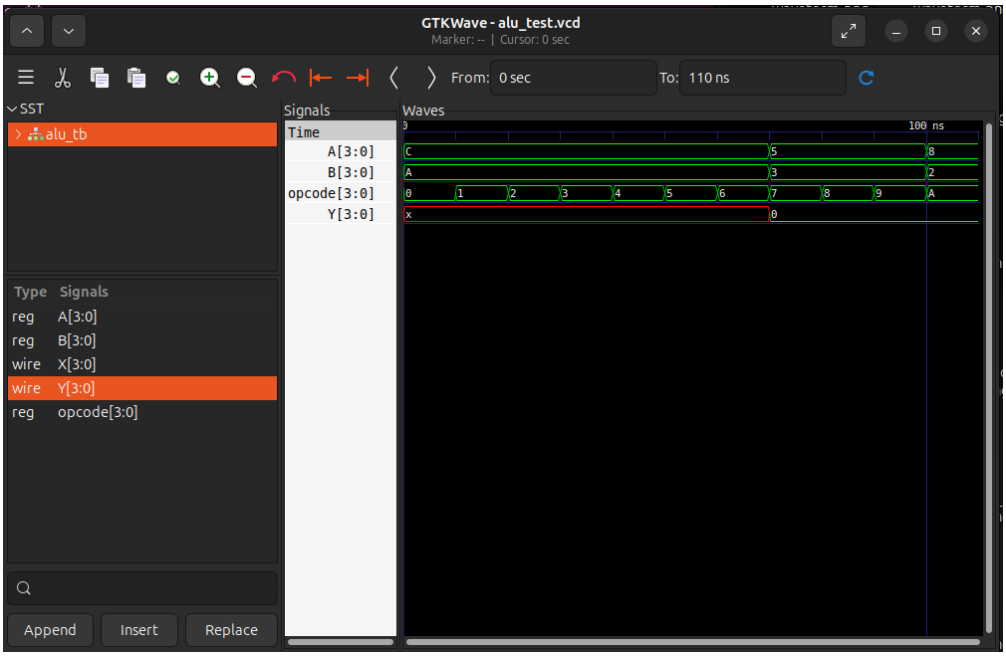


Figure 1: ALU Simulation Waveform

5 Conclusion

Through this project, we successfully developed and integrated all major components of a 4-bit Arithmetic Logic Unit using Verilog. Each logic and arithmetic module was designed, verified with testbenches, and simulated using GTKWave to confirm expected behavior.

The final ALU was constructed by combining the previously tested modules and implementing a control circuit to manage operation selection. The waveform output from the final ALU confirms its ability to correctly process multiple functions — including addition, subtraction, multiplication, division, logical operations, and arithmetic shifts — all based on control signal inputs.

This final integration step marks the completion of our modular ALU design. We now have a working digital system that demonstrates the principles of hardware logic design, binary arithmetic, and simulation-based verification. This project solidifies the importance of modular design, proper testing, and simulation in the development of digital systems.