PROJECT STEP 3

SMARTY PANTS

Brian Lee, Chloe Gentry, Vinny Rose

CS.3339 Computer Architecture

Texas State University

December 3, 2024

Contents

1 Introduction

In this project, we explored a comprehensive Verilog-based implementation and testing process for logical, arithmetic, and shift operations. The conversation covered the design of modular Verilog components, including logic gates, arithmetic units, and shifters, as well as their integration into a testbench. Each module was designed to handle specific operations like addition, subtraction, and various shift types, ensuring clear functionality and reusability. The testbench applied rigorous testing using various input cases, including edge scenarios, and captured results in both display outputs and waveform files for simulation.

Next, we tested each circuit's functionality to ensure expected behavior, followed by generating simulation waveforms to visually confirm the ALU's performance. Finally, we compiled our work and observations into a comprehensive report, which summarizes the methods, results, and key insights gained from the project.

2 Logic circuit

This Verilog code defines a Logic Unit with both single-bit and 4-bit modules for performing logical operations such as AND, NAND, OR, NOR, XOR, XNOR, and NOT. Each operation is implemented as a separate module for modularity and ease of testing.

2.1 Logic Circuit code

```
1  `ifndef LOGIC_UNIT_V
2  `define LOGIC_UNIT_V
3
4  module nand_1bit (Y, A, B);
5      output Y;
6      input A, B;
7      assign Y = !(A && B);
8  endmodule
9
10 module nor_1bit (Y, A, B);
11     output Y;
12     input A, B;
13     assign Y = !(A | B);
14 endmodule
15
16 module not_1bit (Y, A);
17     output Y;
18     input A;
```

```
19      assign Y = !A;
20  endmodule
21
22
23  module and_4bit (Y, A, B);
24      output [3:0] Y; // wire type by default
25      input [3:0] A, B;
26      assign Y = A & B;
27  endmodule
28
29  module nand_4bit (Y, A, B);
30      output [3:0] Y; // wire type by default
31      input [3:0] A, B;
32
33      assign Y = ~(A & B);
34  endmodule
35
36  module nor_4bit (Y, A, B);
37      output [3:0] Y; // wire type by default
38      input [3:0] A, B;
39      assign Y = ~(A | B);
40  endmodule
41
42  module not_4bit (Y, A);
43      output [3:0] Y;  // 4-bit output
44      input  [3:0] A;  // 4-bit input
45      assign Y = ~A;
46  endmodule
47
48  module or_4bit (Y, A, B);
49      output [3:0] Y; // wire type by default
50      input [3:0] A, B;
51      assign Y = A | B;
52  endmodule
53
54  module xnor_4bit (Y, A, B);
55      output [3:0] Y;  // 4-bit output
56      input  [3:0] A;  // 4-bit input
57      input  [3:0] B;  // 4-bit input
58      assign Y = ~(A ^ B);
59  endmodule
60
61  module xor_4bit (Y, A, B);
```

```
    output [3:0] Y; // wire type by default
    input [3:0] A, B;
    assign Y = A ^ B;
endmodule

module logic_1bit (
    output reg Y,          // reg because assigned inside always block
    input A, B,           // Inputs
    input [2:0] control   // 3-bit control signal to select operation
);
    always @(*) begin
        case (control)
            3'b000: Y = A & B;        // AND
            3'b001: Y = !(A & B);     // NAND
            3'b010: Y = A | B;        // OR
            3'b011: Y = !(A | B);     // NOR
            3'b100: Y = A ^ B;        // XOR
            3'b101: Y = ~(A ^ B);     // XNOR
            3'b110: Y = !A;           // NOT
            default: Y = 0;           // Default to 0 if invalid control
    signal
        endcase
    end
endmodule

module logic_4bit (
    output reg [3:0] Y,      // reg for procedural assignments
    input [3:0] A, B,        // 4-bit inputs
    input [2:0] control      // 3-bit control signal to select operation
);
    always @(*) begin
        case (control)
            3'b000: Y = A & B;        // AND
            3'b001: Y = ~(A & B);     // NAND
            3'b010: Y = A | B;        // OR
            3'b011: Y = ~(A | B);     // NOR
            3'b100: Y = A ^ B;        // XOR
            3'b101: Y = ~(A ^ B);     // XNOR
            3'b110: Y = ~A;           // NOT (applies to both A)
            default: Y = 4'b0000;     // Default to 0000 if invalid control
     signal
        endcase
    end
```

```
endmodule

`endif
```

2.2 Logic Waveforms

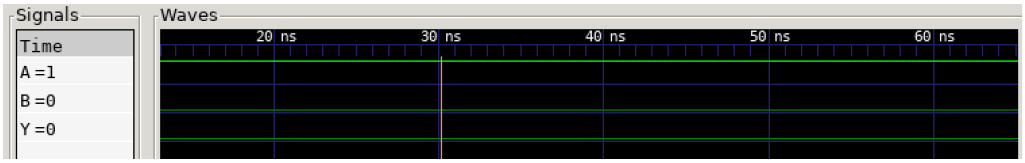

Figure 1: 1-bit Nor wave form

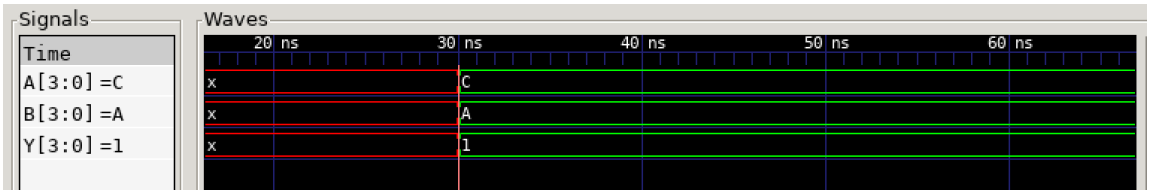

Figure 2: 4-bit Nor wave form

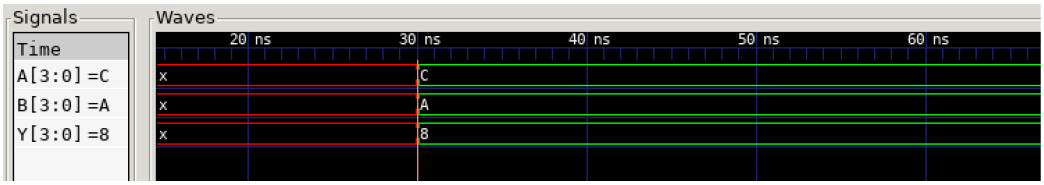

Figure 3: 4-bit And wave form

3 Alu circuit

This Verilog code defines an Arithmetic Logic Unit (ALU) with modular functionality for addition, subtraction, multiplication, and division. Each operation is implemented as a separate module, and a 4-bit opcode determines the operation to execute. The ALU produces outputs such as the arithmetic result, carry-out for addition, product (high and low bits) for multiplication, quotient, remainder, and validity for division. The design promotes modularity and flexibility, making it easy to test and extend.

3.1 ALU circuit code

```
`ifndef ALU_V
`define ALU_V

module addition (
    input [3:0] A,          // 4-bit input A
    input [3:0] B,          // 4-bit input B
    input carry_in,         // 1-bit carry in
    output [3:0] Sum,       // 4-bit sum output
    output carry_out        // 1-bit carry out
);

    wire [4:0] full_sum;   // 5-bit sum to accommodate carry-out

    // Perform the addition with carry_in
    assign full_sum = A + B + carry_in;

    // Assign the lower 4 bits of the sum to the Sum output
    assign Sum = full_sum[3:0];

    // The 5th bit is the carry_out
    assign carry_out = full_sum[4];

endmodule

module subtraction (Y, A, B);
    output [3:0] Y;  // 4-bit output
    input  [3:0] A;  // 4-bit input A
    input  [3:0] B;  // 4-bit input B

    assign Y = A - B;  // Perform 4-bit subtraction

endmodule

module multiplication (
    input [3:0] A, B,
    output [3:0] product_low,   // Lower 4 bits of the product
    output [3:0] product_high   // Upper 4 bits of the product
);
    wire [7:0] full_product;    // 8-bit wire to hold the full product

    assign full_product = A * B;
    assign product_low = full_product[3:0];   // Lower 4 bits
```

```
    assign product_high = full_product[7:4];  // Upper 4 bits
endmodule

module division (
    input [3:0] dividend,
    input [3:0] divisor,
    output reg [3:0] quotient,
    output reg [3:0] remainder,
    output reg valid
);
    reg [3:0] temp_dividend;
    reg [3:0] temp_quotient;

    integer i;

    always @(*) begin
        quotient = 4'b0000;
        remainder = 4'b0000;
        valid = 1'b0;

        if (divisor == 0) begin
            valid = 1'b0;
            quotient = 4'b0000;
            remainder = dividend;
        end else begin
            temp_dividend = dividend;
            temp_quotient = 4'b0000;

            for (i = 3; i >= 0; i = i - 1) begin
                temp_quotient = {temp_quotient[2:0], temp_dividend[3]};
                temp_dividend = {temp_dividend[2:0], 1'b0};

                //if (temp_quotient >= divisor) begin
                //    temp_quotient = temp_quotient - divisor;
                //    dividend[i:i] = 1'b1;
                //    end
            end

            quotient = temp_quotient;
            remainder = dividend;
            valid = 1'b1;
        end
    end
```

```
endmodule

`ifndef ALU_V
`define ALU_V

module alu (
    input [3:0] A, B,       // 4-bit inputs
    input carry_in,         // Carry input for addition
    input [1:0] shift_amt,  // 2-bit shift amount (unused in this case)
    input shift_dir,        // Shift direction (unused in this case)
    input [3:0] opcode,     // 4-bit control signal
    output reg [3:0] Y,     // 4-bit result output (this is a reg)
    output reg carry_out,   // Carry-out for addition (this is a reg)
    output reg valid_div,   // Valid flag for division (this is a reg)
    output reg [3:0] remainder, // Remainder from division (this is a reg)
    output reg [3:0] product_low, // Multiplication low part (this is a reg
    )
    output reg [3:0] product_high // Multiplication high part (this is a
    reg)
);

    // Intermediate wires for all module outputs
    wire [3:0] arithmetic_out;
    wire [3:0] quotient_out, remainder_out;
    wire add_carry_out;

    // Instantiate Arithmetic Modules
    addition u_add (.A(A), .B(B), .carry_in(carry_in), .Sum(arithmetic_out)
    , .carry_out(add_carry_out));
    subtraction u_sub (.Y(arithmetic_out), .A(A), .B(B)); // Subtraction
    multiplication u_mul (.A(A), .B(B), .product_low(product_low), .
    product_high(product_high));  // Multiplication
    division u_div (.dividend(A), .divisor(B), .quotient(quotient_out), .
    remainder(remainder_out), .valid(valid_div)); // Division

    always @(*) begin
        // Default values (optional)
        product_low = 4'b0000;
        product_high = 4'b0000;
        valid_div = 1'b0;

        case (opcode[3:2])
            2'b00: begin    // Arithmetic operations (addition/subtraction)
```

```
124                 case (opcode[1:0])
125                     2'b00: begin
126                         Y = arithmetic_out;  // Addition
127                         carry_out = add_carry_out;
128                     end
129                     2'b01: Y = arithmetic_out;  // Subtraction
130                     default: Y = 4'b0000; // Default
131                 endcase
132             end
133 
134             2'b01: begin  // Multiplication
135                 Y = product_low;  // Lower 4 bits of the product
136                 product_high = product_high;  // Returning upper 4 bits if
    needed
137             end
138 
139             2'b10: begin  // Division
140                 Y = quotient_out;
141                 remainder = remainder_out;
142                 valid_div = 1'b1;
143             end
144 
145             default: Y = 4'b0000; // Default case for unsupported
    operations
146         endcase
147     end
148 endmodule
149 
150 `endif
151 `endif
```

3.2 ALU Waveforms

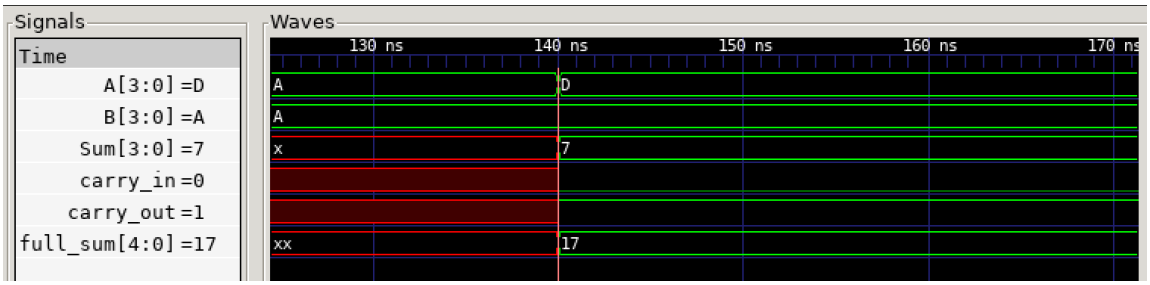

Figure 4: 4-bit Addition wave form

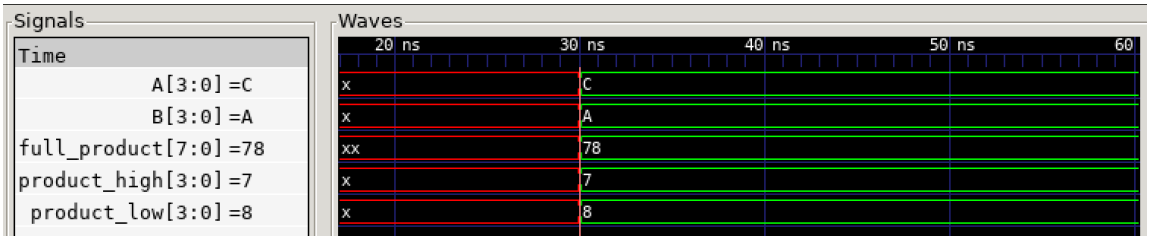

Figure 5: 4-bit Multiplication wave form

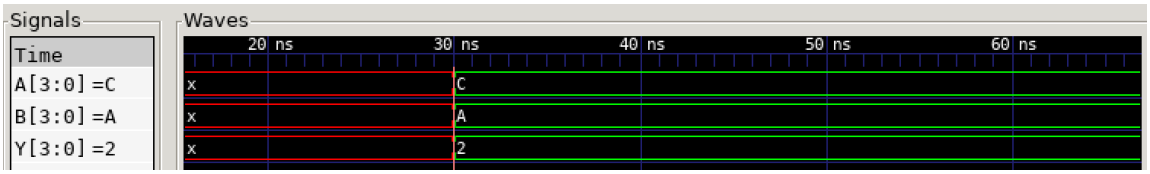

Figure 6: 4-bit Subtraction wave form

4 Shift circuit

The provided Verilog code defines two modules for bit shifting operations. A 1 bit shift and a 2 by 4 bit shift

4.1 Shifter circuit code

```
`ifndef SHIFTER_V
`define SHIFTER_V

module shift_1bit (
    output reg [3:0] Y,      // Change Y to reg
    input [3:0] A,           // 4-bit input
    input [1:0] control      // 2-bit control signal for shift direction
);
    always @(*) begin
        case (control)
            2'b00: Y = A;                // No shift
            2'b01: Y = A << 1;           // Left shift by 1
            2'b10: Y = A >> 1;           // Right shift by 1
            default: Y = 4'b0000;        // Default case
        endcase
    end
```

```
endmodule

module shift_2x4bit (
    input [3:0] A,
    input [1:0] amt,
    input dir,
    output reg [3:0] Y   // Keep Y as reg since it's assigned in an always
    block
);

    always @(*) begin
        case (amt)
            2'b00: Y = A; // No shift
            2'b01: begin
                if (dir == 1'b0) // Left shift by 1
                    Y = {A[2:0], 1'b0};
                else // Right shift by 1
                    Y = {1'b0, A[3:1]};
            end
            2'b10: begin
                if (dir == 1'b0) // Left shift by 2
                    Y = {A[1:0], 2'b00};
                else // Right shift by 2
                    Y = {2'b00, A[3:2]};
            end
            default: Y = 4'b0000; // Default case (should not happen)
        endcase
    end

endmodule

`endif
```

4.2 Shift Waveform

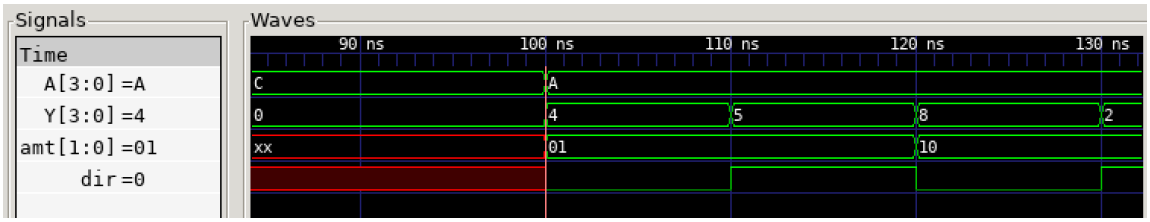

Figure 7: 4-bit Shift wave form

5 Testbench circuit

This Verilog testbench evaluates logical, arithmetic, and shift operations across multiple modules. It tests single-bit and 4-bit logical operations, including AND, OR, NOT, XOR, and their variations, as well as addition, subtraction, multiplication, and division with detailed result validation. Shift operations are tested for both single-bit and multi-bit shifts in left and right directions.

5.1 Testbench circuit code

```
`timescale 1ns / 1ps
`include "alu.v"
`include "logic_unit.v"
`include "shifter.v"

module testbench;

    // Inputs for logical operations
    reg A, B;
    reg [3:0] A4, B4;
    reg [1:0] amt;          // For shift amount
    reg dir;                // For shift direction (0 = left, 1 = right)
    reg [1:0] control;      // For controlling shift type (left/right)
    reg carry_in;           // For arithmetic operations
    reg [3:0] dividend, divisor;

    // Outputs for logical operations
    wire nand1_out, nor1_out, not1_out;
    wire [3:0] and4_out, nand4_out, nor4_out, not4_out, or4_out, xor4_out,
   xnor4_out;

    // Outputs for arithmetic operations
    wire [3:0] Sum, SubResult, MulLow, MulHigh, DivQuotient, DivRemainder;
    wire carry_out, valid;

    // Outputs for shift operations
    wire [3:0] shift1out, shift4out;

    // Instantiate logical operations modules
    nand_1bit uut_nand1 (.Y(nand1_out), .A(A), .B(B));
    nor_1bit uut_nor1 (.Y(nor1_out), .A(A), .B(B));
    not_1bit uut_not1 (.Y(not1_out), .A(A));

```

```
33      and_4bit uut_and4 (.Y(and4_out), .A(A4), .B(B4));
34      nand_4bit uut_nand4 (.Y(nand4_out), .A(A4), .B(B4));
35      nor_4bit uut_nor4 (.Y(nor4_out), .A(A4), .B(B4));
36      not_4bit uut_not4 (.Y(not4_out), .A(A4));
37      or_4bit uut_or4 (.Y(or4_out), .A(A4), .B(B4));
38      xor_4bit uut_xor4 (.Y(xor4_out), .A(A4), .B(B4));
39      xnor_4bit uut_xnor4 (.Y(xnor4_out), .A(A4), .B(B4));
40
41      // Instantiate shift modules
42      shift_1bit uut_shift1 (.Y(shift1out), .A(A4));
43      shift_2x4bit uut_shift4 (.Y(shift4out), .A(A4), .amt(amt), .dir(dir));
44
45      // Instantiate arithmetic operations modules
46      addition uut_addition (.A(A4), .B(B4), .carry_in(carry_in), .Sum(Sum),
    .carry_out(carry_out));
47      subtraction uut_subtraction (.Y(SubResult), .A(A4), .B(B4));
48      multiplication uut_multiplication (.A(A4), .B(B4), .product_low(MulLow)
    , .product_high(MulHigh));
49      division uut_division (.dividend(dividend), .divisor(divisor), .
    quotient(DivQuotient), .remainder(DivRemainder), .valid(valid));
50
51      // Create waveform dump
52      initial begin
53          $dumpfile("waveform.vcd");    // Name of the VCD file for the
    waveform
54          $dumpvars(0, testbench);      // Dump all variables in the testbench
     module
55
56          // Test Logical Operations
57          A = 1; B = 0;
58          #10; $display("nand_1bit: %b AND %b = %b", A, B, nand1_out);
59          #10; $display("nor_1bit: %b OR %b = %b", A, B, nor1_out);
60          #10; $display("not_1bit: NOT %b = %b", A, not1_out);
61
62          A4 = 4'b1100; B4 = 4'b1010;
63          #10; $display("and_4bit: %b AND %b = %b", A4, B4, and4_out);
64          #10; $display("nand_4bit: %b NAND %b = %b", A4, B4, nand4_out);
65          #10; $display("nor_4bit: %b NOR %b = %b", A4, B4, nor4_out);
66          #10; $display("not_4bit: NOT %b = %b", A4, not4_out);
67          #10; $display("or_4bit: %b OR %b = %b", A4, B4, or4_out);
68          #10; $display("xor_4bit: %b XOR %b = %b", A4, B4, xor4_out);
69          #10; $display("xnor_4bit: %b XNOR %b = %b", A4, B4, xnor4_out);
70
```

```
71          // Test Shift Operations
72          amt = 2'b01; dir = 1'b0; A4 = 4'b1010;   // Shift left by 1
73          #10; $display("shift_1bit (left): %b -> %b", A4, shift1out);
74
75          amt = 2'b01; dir = 1'b1; A4 = 4'b1010;   // Shift right by 1
76          #10; $display("shift_1bit (right): %b -> %b", A4, shift1out);
77
78          amt = 2'b10; dir = 1'b0; A4 = 4'b1010;   // Shift left by 2
79          #10; $display("shift_2x4bit (left): %b -> %b", A4, shift4out);
80
81          amt = 2'b10; dir = 1'b1; A4 = 4'b1010;   // Shift right by 2
82          #10; $display("shift_2x4bit (right): %b -> %b", A4, shift4out);
83
84          // Test Arithmetic Operations
85          A4 = 4'b1101; B4 = 4'b1010; carry_in = 1'b0;
86          #10; $display("addition: %b + %b = %b, carry_out = %b", A4, B4, Sum
    , carry_out);
87
88          #10; $display("subtraction: %b - %b = %b", A4, B4, SubResult);
89
90          #10; $display("multiplication: %b * %b = %b %b", A4, B4, MulHigh,
    MulLow);
91
92          dividend = 4'b1010; divisor = 4'b0011; // 10 / 3
93          #10; $display("division: %b / %b = quotient: %b, remainder: %b",
    dividend, divisor, DivQuotient, DivRemainder);
94
95          $finish;
96      end
97
98  endmodule
```

6 Opcode Module

Step3 > opcodes.md > # Opcode Table for ALU Operations

```
# Opcode Table for ALU Operations

| Opcode (4 bits) | Operation Group      | Specific Operation       | Description                                  |
|-----------------|----------------------|--------------------------|----------------------------------------------|
| 0000            | Arithmetic           | Addition                 | `Y = A + B + carry_in`                       |
| 0001            | Arithmetic           | Subtraction              | `Y = A - B`                                  |
| 0100            | Multiplication       | Product Low              | `Y = product_low` (Lower 4 bits of product)  |
| 1000            | Division             | Quotient                 | `Y = quotient_out`                           |
| 1001            | Division             | Remainder                | `remainder = remainder_out`                  |
| Default         | Undefined Operation  | No Operation             | `Y = 4'b0000`
```

Figure 8: Opcode Screenshot

7 Conclusion

This project introduced the design and testing of a Arithmetic Logic Unit (ALU), logic unit, and shifting unit. We coded essential logic functions like AND, OR, and NOT, along with shifting operations, which are crucial for handling binary data. We also developed basic arithmetic operations—addition, subtraction, multiplication, and division—which included handling carries and remainders to ensure accurate results.

Testing the circuits and generating waveforms confirmed that our ALU worked correctly across different inputs. This process highlighted the importance of both accuracy in coding and thorough testing. Overall, the project has been valuable for understanding digital circuits and the structure of an ALU, preparing us for more advanced digital logic design.