PROJECT STEP 2

SMARTY PANTS

Brian Lee, Chloe Gentry, Vinny Rose

CS.3339 Computer Architecture

Texas State University

October 28, 2024

Contents

1 Introduction

In this project, we implemented essential components of a 4-bit Arithmetic Logic Unit (ALU). First, we coded 4-bit binary logic functions, including AND, NAND, OR, NOR, XOR, XNOR, and NOT, as well as a 2x4-bit input/output shifter. We then implemented 4-bit arithmetic operations such as addition, subtraction, multiplication, and division, integrating carry-in and carry-out circuits for extended accuracy. We included an extra 4-bit output for multiplication and division remainders.

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 2x4 Shift Circuit

The 2x4 shift circuit takes a 4-bit input and shifts the bits left or right by up to two positions, depending on the control signal. Each shift moves the bits in the input to adjacent positions, and the vacant bits are filled with zeros. Shifting by one or two positions allows for additional flexibility in data manipulation, making the circuit versatile in operations where both single and double shifts are required.

2.1 Shift Circuit Verilog Code

```
1  `timescale 1ns / 1ps
2
3  module shift_tb;
4
5      // Inputs
6      reg [3:0] A;
7      reg [1:0] amt;
8      reg dir;
9
10     // Outputs
11     wire [3:0] Y;
12
13     // Instantiate the shifter module
14     shift_gate uut (
15         .A(A),
16         .amt(amt),
17         .dir(dir),
```

```
        .Y(Y)
    );

    initial begin
        // Open a file for waveform output
        $dumpfile("shift_tb.vcd");
        $dumpvars(0, shift_tb);

        // Test 1: No shift
        A = 4'b1010;  // Input: 10
        amt = 2'b00;  // Shift amount: 0
        dir = 1'b0;   // Direction: left
        #10;

        // Test 2: Left shift by 1
        amt = 2'b01;  // Shift amount: 1
        dir = 1'b0;   // Direction: left
        #10;

        // Test 3: Left shift by 2
        amt = 2'b10;  // Shift amount: 2
        dir = 1'b0;   // Direction: left
        #10;

        // Test 4: Right shift by 1
        amt = 2'b01;  // Shift amount: 1
        dir = 1'b1;   // Direction: right
        #10;

        // Test 5: Right shift by 2
        amt = 2'b10;  // Shift amount: 2
        dir = 1'b1;   // Direction: right
        #10;

        // Test 6: Edge case - All zeros
        A = 4'b0000;  // Input: 0
        amt = 2'b10;  // Shift amount: 2
        dir = 1'b0;   // Direction: left
        #10;

        // End simulation
        $finish;
    end
```

```
61
62 endmodule
```

To test the Shift circuit we have created 3 input registers, and 1 output. Input A is 4bits, being the input that we want to shift, next input, amt, is 2 bits, being the amount that we want to shift A by, and the last input, dir, is the direction we want to shift, being 0 for left and 1 for right. It will then output the resulting shifted bits into Y, and we will know if its working correctly if Y is a shifted version of A in the direction and the amount specified.

```
1  `timescale 1ns / 1ps
2
3  module shift_tb;
4
5      // Inputs
6      reg [3:0] A;
7      reg [1:0] amt;
8      reg dir;
9
10     // Outputs
11     wire [3:0] Y;
12
13     // Instantiate the shifter module
14     shift_gate uut (
15         .A(A),
16         .amt(amt),
17         .dir(dir),
18         .Y(Y)
19     );
20
21     initial begin
22         // Open a file for waveform output
23         $dumpfile("shift_tb.vcd");
24         $dumpvars(0, shift_tb);
25
26         // Test 1: No shift
27         A = 4'b1010;  // Input: 10
28         amt = 2'b00;  // Shift amount: 0
29         dir = 1'b0;   // Direction: left
30         #10;
31
32         // Test 2: Left shift by 1
33         amt = 2'b01;  // Shift amount: 1
34         dir = 1'b0;   // Direction: left
```

```
35          #10;
36
37          // Test 3: Left shift by 2
38          amt = 2'b10;  // Shift amount: 2
39          dir = 1'b0;   // Direction: left
40          #10;
41
42          // Test 4: Right shift by 1
43          amt = 2'b01;  // Shift amount: 1
44          dir = 1'b1;   // Direction: right
45          #10;
46
47          // Test 5: Right shift by 2
48          amt = 2'b10;  // Shift amount: 2
49          dir = 1'b1;   // Direction: right
50          #10;
51
52          // Test 6: Edge case - All zeros
53          A = 4'b0000;  // Input: 0
54          amt = 2'b10;  // Shift amount: 2
55          dir = 1'b0;   // Direction: left
56          #10;
57
58          // End simulation
59          $finish;
60      end
61
62  endmodule
```

2.2 Shift Circuit Waveforms

At 10 ns, we can see that A is 1010, amt is 01 (1), and dir is 0 (left), so the output Y is 0100.

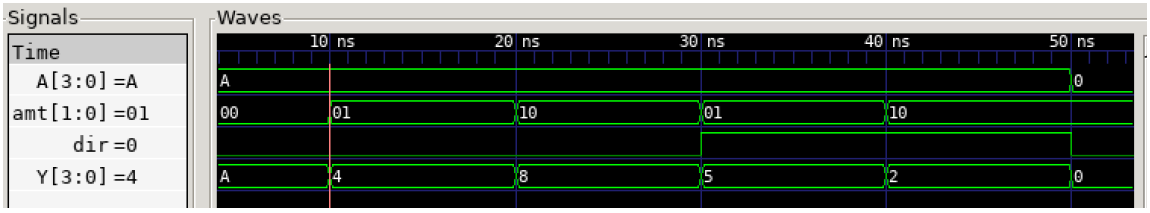

Figure 1: Shift Circuit with marker at 10ns

At 20 ns, we can see that A is 1010, amt is 10 (2), and dir is 0 (left), so the output Y is 1000.

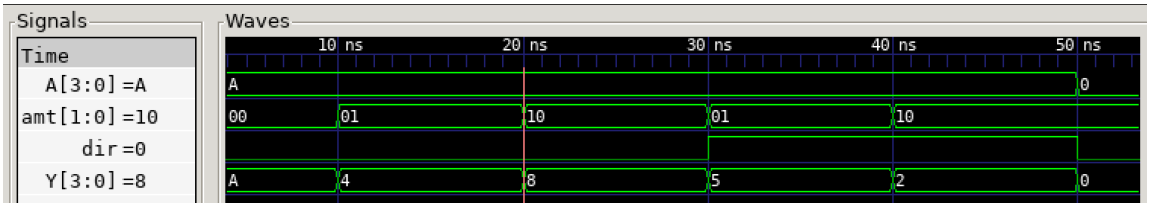

Figure 2: Shift Circuit with marker at 20ns

At 30 ns, we can see that A is 1010, amt is 01 (1), and dir is 1 (right), so the output Y is 0101.

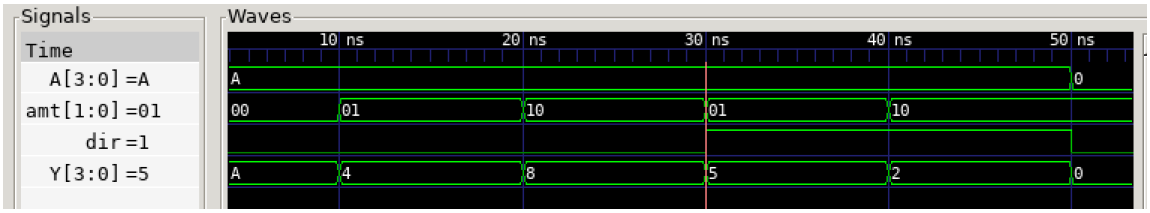

Figure 3: Shift Circuit with marker at 30ns

3 Not Circuit

The Not circuit takes one input A, with an output Y. This will output a 0 if A is a 1, and will output 1 if A is a 0.

3.1 Not Circuit Verilog Code

```
1  `ifndef NOT_GATE_V
2  `define NOT_GATE_V
3
4  module not_gate (Y, A);
5      output [3:0] Y;  // 4-bit output
6      input  [3:0] A;  // 4-bit input
7      assign Y = ~A;
8  endmodule
9
10 `endif
```

To test the Not circuit, we have created one register A, as well as a wire Y. This way we are able to take each input for A and test output for A=0 and A=1. We'll know if this is working correctly if Y returns 1 for the test where A is 0 and Y returns 0 when A is 1.

```
1  `timescale 1ns / 1ns
2  `include "not_gate.v"
3
4  module not_tb;
5
6  reg [3:0]A;
7  wire [3:0]Y;
8
9  not_gate uut(Y, A);
10
11 initial begin
12
13     $dumpfile("not_tb.vcd");//holds output waveform
14     $dumpvars(0, not_tb);
15
16         A = 4'b0000; #10;
17         A = 4'b1111; #10;
18         A = 4'b1010; #10;
19         A = 4'b0101; #10;
20
21     $display("Testing not");
22
23 end
24
25 endmodule
```

3.2 Not Circuit Waveform

At 0 ns, we can see that A is the 0h(0000), so the output Y is Fh(1111).

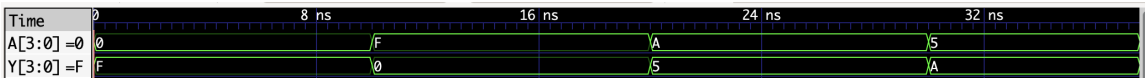

Figure 4: Not Circuit with marker at 0ns

At 20 ns, A becomes the Ah(1010), so Y becomes the 5h(0101).

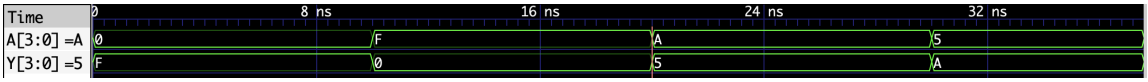

Figure 5: Not Circuit with marker at 20ns

At 40 ns, A becomes the 5h(0101), so Y becomes the hex value Ah(1010).

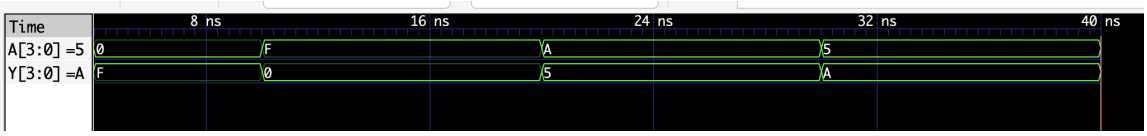

Figure 6: Not Circuit with marker at 40ns

4 And Circuit

The And circuit takes two 4-bit inputs A and B, with a 4-bit output Y. A and B will undergo the bitwise And operation, and the result will be output to Y.

4.1 And Circuit Verilog Code

```
1  `ifndef AND_GATE_V
2  `define AND_GATE_V
3
4  module and_gate (Y, A, B);
5      output [3:0] Y;
6      input [3:0] A, B;
7      assign Y = A & B;
8  endmodule
9
10 `endif
```

To test the And circuit, we have created two registers A and B, as well as a wire Y. We then create a for loop to produce all 16 possible values for A and B. We will know that our And circuit is behaving as expected if Y produces correct values for all combinations of A and B. For example, if A=0001 and B=0001, the expected result for Y is 0001.

```
1  `timescale 1ns / 1ns
2  `include "and_gate.v"
3
4  module and_tb;
5
6      reg [3:0] A, B;
7      wire [3:0] Y;
8
9      and_gate uut(Y, A, B);
10
```

```
    integer i, j;

    initial begin
        $dumpfile("and_tb.vcd");\
        $dumpvars(0, and_tb);

        // Loop through all 16 possible values of A and B
        for (i = 0; i < 16; i = i + 1) begin
            for (j = 0; j < 16; j = j + 1) begin
                A = i;
                B = j;
                #10;
                $display("A = %b, B = %b, Y = %b", A, B, Y);  // Display A, B, and Y
            end
        end

        $display("All combinations tested");
        $finish;
    end

endmodule
```

4.2 And Circuit Waveform

At 1ns, A and B are both 0000, so Y is also 0000

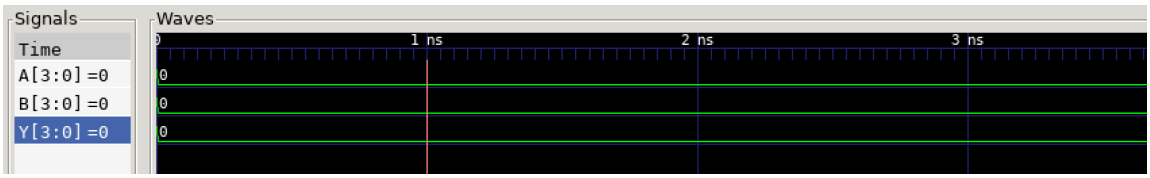

Figure 7: And Circuit with marker at 1ns

At 500ns, A=0011 and B=0010, so Y=0010

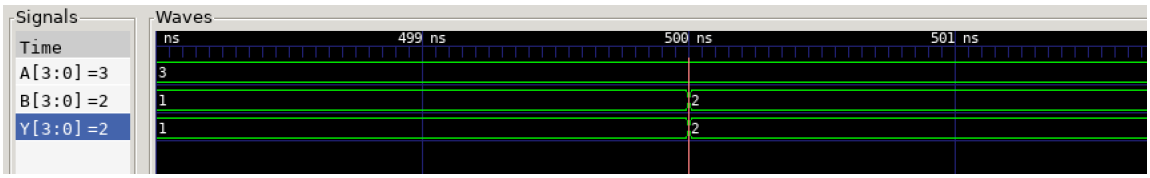

Figure 8: And Circuit with marker at 500ns

At 2550ns, A and B are both F (1111), so Y is also F (1111).

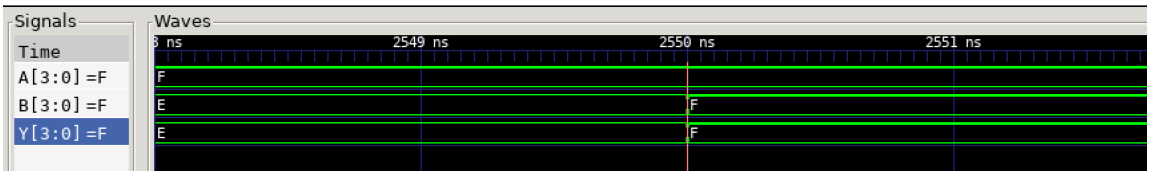

Figure 9: And Circuit with marker at 2550ns

5 Nand Circuit

The Nand circuit takes two 4-bit inputs A and B, with a 4-bit output Y. A and B will undergo the bitwise Nand operation, and the result will be output to Y. “‘

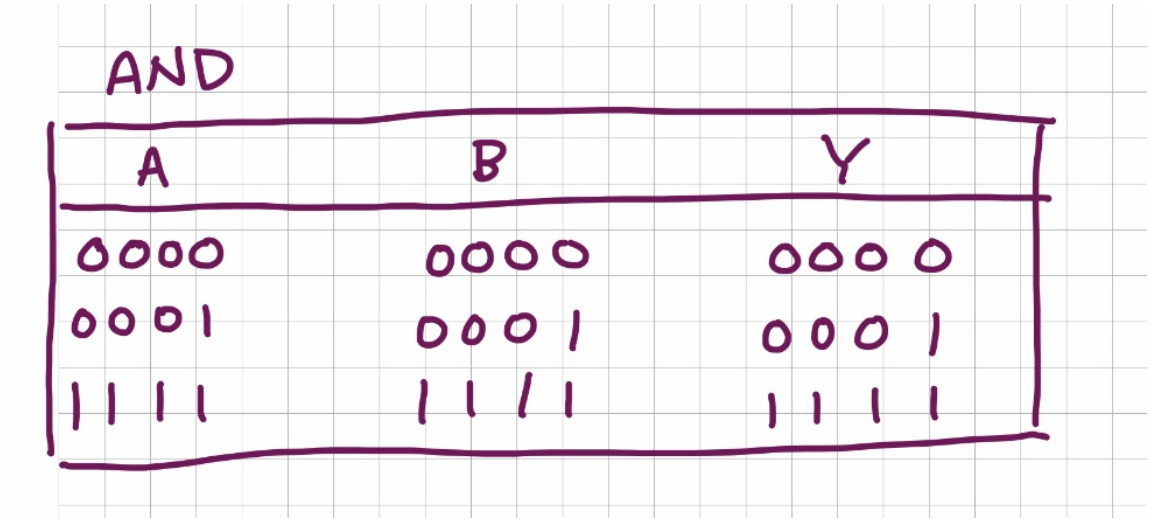

Figure 10: And truth table and gate

5.1 Nand Circuit Verilog Code

```
`ifndef NAND_GATE_V
`define NAND_GATE_V

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

    assign Y = ~(A & B);
endmodule

`endif
```

To test the Nand circuit, we have created two registers A and B, as well as a wire Y. We then create a for loop to produce all 16 possible values for A and B. We will know that our Nand circuit is behaving as expected if Y produces correct values for all combinations of A and B. For example, if A=0001 and B=0001, the expected result for Y is 1110.

```
`timescale 1ns / 1ns
`include "nand_gate.v"

module nand_tb;

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

nand_gate uut(Y, A, B);

integer i, j;

initial begin

    $dumpfile("nand_tb.vcd");
    $dumpvars(0, nand_tb);

    // Loop through all 16 possible values of A and B
    for (i = 0; i < 16; i = i + 1) begin
        for (j = 0; j < 16; j = j + 1) begin
            A = i;
            B = j;
            #10;
            $display("A = %b, B = %b, Y = %b", A, B, Y);  // Display A, B,
   and Y
        end
```

```
26      end
27
28      $display("All combinations tested");
29      $finish;
30  end
31
32  endmodule
```

5.2 Nand Circuit Waveform

At 1ns, A and B are both 0000, so Y is F (1111).

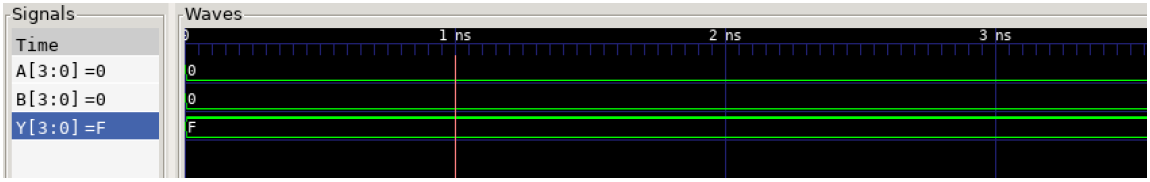

Figure 11: Nand Circuit with marker at 1ns

At 500ns, A=0011 and B=0010, so Y=D (1101)

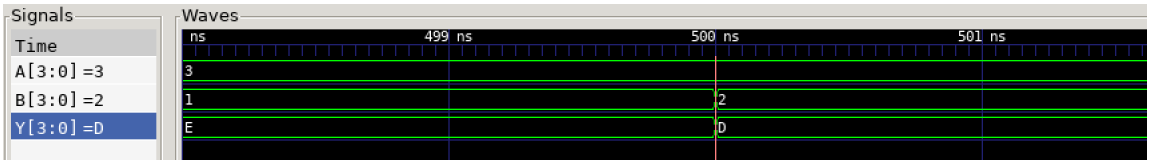

Figure 12: Nand Circuit with marker at 500ns

At 2500ns, A and B are both F (1111), so Y is 0000.

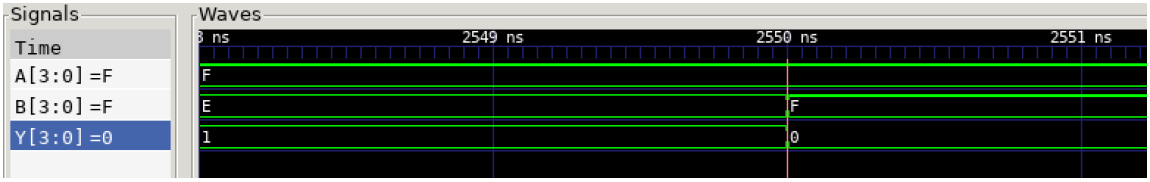

Figure 13: Nand Circuit with marker at 560ns

6 Or Circuit

The Or circuit takes two 4-bit inputs A and B, and has a 4-bit output Y. Each bit of Y is the result of a bitwise OR operation applied between corresponding bits in A and B.

6.1 Or Circuit Verilog Code

```
`ifndef OR_GATE_V
`define OR_GATE_V

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

`endif
```

To test the Or circuit, we have created two registers A and B, as well as a wire Y. We then create a for loop to produce all 16 possible values for A and B. We will know that our And circuit is behaving as expected if Y produces correct values for all combinations of A and B.

```
`timescale 1ns / 1ns
`include "or_gate.v"

module or_tb;

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

    or_gate uut(Y, A, B);

    integer i, j;

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

        // Loop through all 16 possible values of A and B
        for (i = 0; i < 16; i = i + 1) begin
            for (j = 0; j < 16; j = j + 1) begin
                A = i;
```

```
                B = j;
                #10;
                $display("A = %b, B = %b, Y = %b", A, B, Y);  // Display A,
    B, and Y
            end
        end

        $display("All combinations tested");
        $finish;
    end

endmodule
```

6.2 Or Circuit Waveform

At 160 ns, A is 1 (0001) and B is 0 (0000), so Y is 1 (0001).

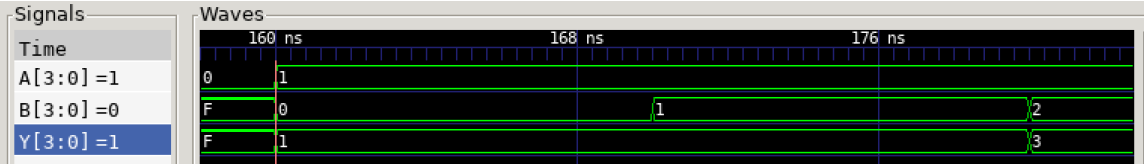

Figure 14: Nor Circuit with marker at 160ns

At 1650 ns, A is A (1010) and B is 5 (0101), so Y is now F (1111).

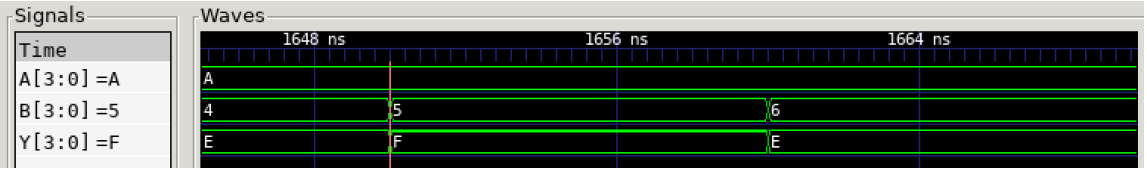

Figure 15: Nor Circuit with marker at 1650ns

At 2400 ns, A is F (1111) and B is 0 (0000), so Y is now F (1111).

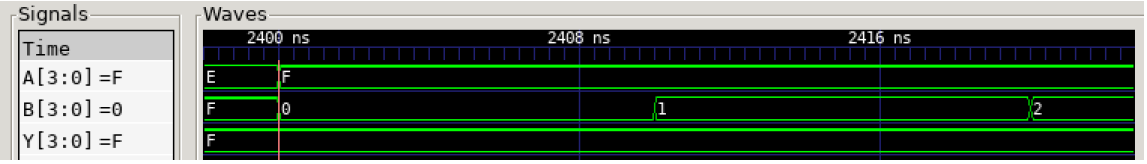

Figure 16: Nor Circuit with marker at 2400ns

7 Nor Circuit

The Or circuit takes two 4-bit inputs A and B, and has a 4-bit output Y. Each bit of Y is the result of the opposite of a bitwise OR operation applied between corresponding bits in A and B.

7.1 Nor Circuit Verilog Code

```
1  `ifndef NOR_GATE_V
2  `define NOR_GATE_V
3
4  module nor_gate (Y, A, B);
5      output [3:0] Y;
6      input [3:0] A, B;
7      assign Y = ~(A | B);
8  endmodule
9
10 `endif
```

To test the Nor circuit, we have created two registers A and B, as well as a wire Y. We then create a for loop to produce all 16 possible values for A and B. We will know that our And circuit is behaving as expected if Y produces correct values for all combinations of A and B.

```
1  `timescale 1ns / 1ns
2  `include "nor_gate.v"
3
4  module nor_tb;
5
6      reg [3:0] A, B;
7      wire [3:0] Y;
8
9      nor_gate uut(Y, A, B);
10
```

```
    integer i, j;

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

        // Loop through all 16 possible values of A and B
        for (i = 0; i < 16; i = i + 1) begin
            for (j = 0; j < 16; j = j + 1) begin
                A = i;
                B = j;
                #10;
                $display("A = %b, B = %b, Y = %b", A, B, Y);  // Display A, B, and Y
            end
        end

        $display("All combinations tested");
        $finish;
    end

endmodule
```

7.2 Nor Circuit Waveform

At 0ns, A and B are both 0000, so Y is F (1111).

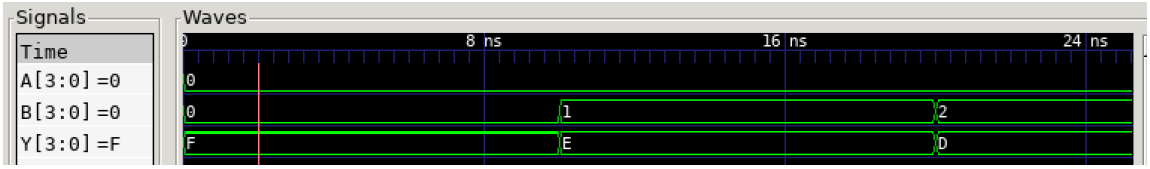

Figure 17: Nor Circuit with marker at 0ns

At 2400ns, A is F (1111) and B is 0 (0000), so Y is 0 (0000).

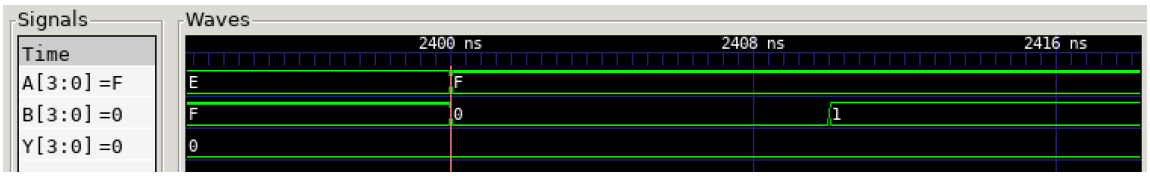

Figure 18: Nor Circuit with marker at 2400ns

At 1600ns, A is A (1010) and B is 0 (0000), so Y is 5 (0101).

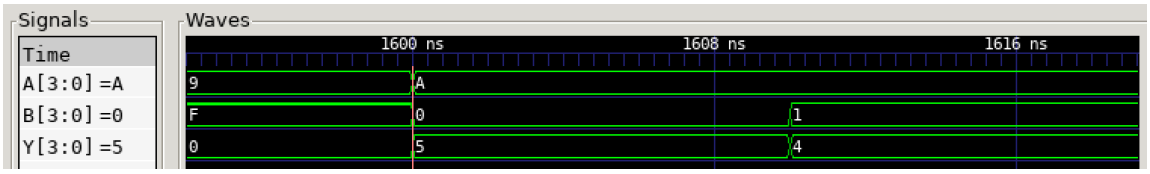

Figure 19: Nor Circuit with marker at 1600ns

8 Xor Circuit

The Xor circuit takes two 4-bit inputs A and B, with a 4-bit output Y. A and B will undergo the bitwise Xor operation, and the result will be output to Y.

```
1  `ifndef XOR_GATE_V
2  `define XOR_GATE_V
3
4  module xor_gate (Y, A, B);
5      output [3:0] Y;
6      input [3:0] A, B;
7      assign Y = A ^ B;
8  endmodule
9
10 `endif
```

To test the Xor circuit, we have created two registers A and B, as well as a wire Y. We then create a for loop to produce all 16 possible values for A and B. We will know that our Xor circuit is behaving as expected if Y produces correct values for all combinations of A and B. For example, if A=1010 and B=1101, the expected result for Y is 0111.

```
1  `timescale 1ns / 1ns
2  `include "xor_gate.v"
3
```

```
module xor_tb;

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

    xor_gate uut(Y, A, B);

    integer i, j;

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

        // Loop through all 16 possible values of A and B
        for (i = 0; i < 16; i = i + 1) begin
            for (j = 0; j < 16; j = j + 1) begin
                A = i;
                B = j;
                #10;
                $display("A = %b, B = %b, Y = %b", A, B, Y);  // Display A, B, and Y
            end
        end

        $display("All combinations tested");
        $finish;
    end

endmodule
```

8.1 Xor Circuit Waveform

At 1ns, A and B are both 0000, so Y is also 0000.

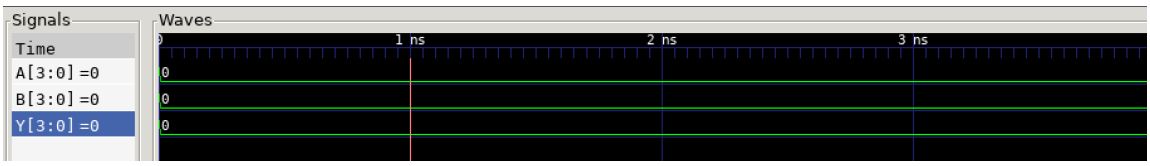

Figure 20: Xor Circuit with marker at 1ns

At 500ns, A=0011 and B=0010, so Y=0001.

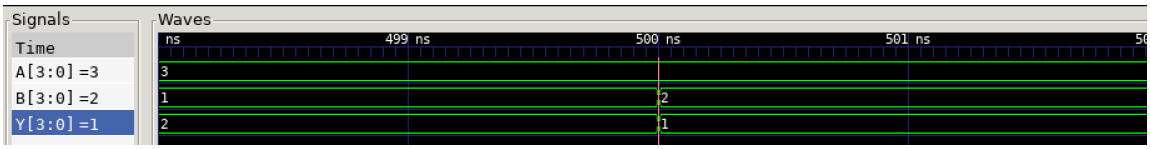

Figure 21: Xor Circuit with marker at 500ns

At 2500ns, A and B are both F, so Y=0000.

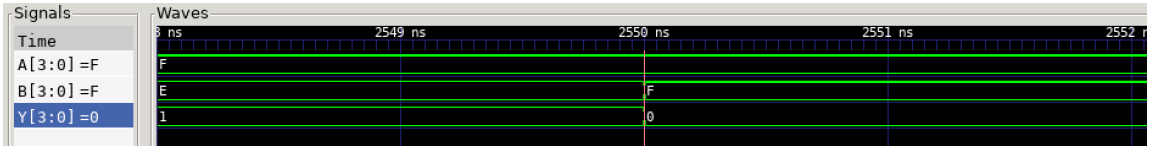

Figure 22: Xor Circuit with marker at 2500ns

9 XNor Circuit

The Xnor circuit takes two 4-bit inputs A and B, with a 4-bit output Y. A and B will undergo the bitwise Xnor operation, and the result will be output to Y.

```
1  `ifndef xnor_GATE_V
2  `define xnor_GATE_V
3
4  module xnor_gate (Y, A, B);
5      output [3:0] Y;  // 4-bit output
6      input  [3:0] A;  // 4-bit input
7      input  [3:0] B;  // 4-bit input
8      assign Y = ~(A ^ B);
9  endmodule
10
11 `endif
```

To test the Xnor circuit, we have created two registers A and B, as well as a wire Y. We give the circuit some test cases to test its validity. We will know that our Xnor circuit is behaving as expected if Y produces correct values for all combinations of A and B. For example, if A = 1010 and B = 1101, the expected result for Y is 1000.

```
1  `timescale 1ns / 1ns
2  `include "xnor_gate.v"
3
```

```
module xnor_tb;

reg [3:0]A;
reg [3:0]B;
wire [3:0]Y;

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

integer i;  // Loop variable
initial begin

    $dumpfile("xnor_tb.vcd");//holds output waveform
    $dumpvars(0, xnor_tb);
        // Apply test cases
        A = 4'b0000; B = 4'b0000; #10;
        A = 4'b0000; B = 4'b1111; #10;
        A = 4'b1010; B = 4'b0101; #10;
        A = 4'b1100; B = 4'b1010; #10;
        A = 4'b1111; B = 4'b1111; #10;
        A = 4'b0110; B = 4'b0110; #10;
        A = 4'b1001; B = 4'b1001; #10;
        A = 4'b1101; B = 4'b0111; #10;


    $display("Testing xnor");

end

endmodule
```

9.1 Xnor Circuit Waveform

At 1ns, A and B are both 0h (0000), so Y is Fh(0000).

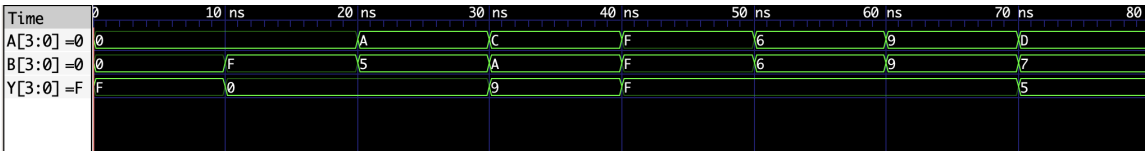

Figure 23: Xnor Circuit with marker at 1ns

At 20ns, A = Ah (1010) and B = 5h(0101), so Y = 0h (0000).

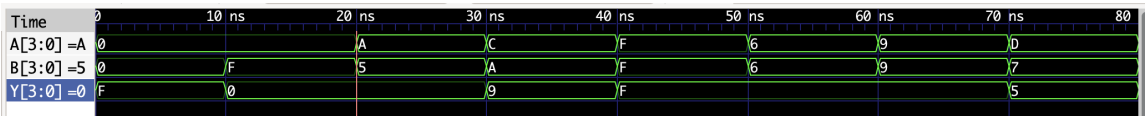

Figure 24: Xor Circuit with marker at 500ns

At 40ns, A and B are both Fh(1111), so Y = Fh(1111).

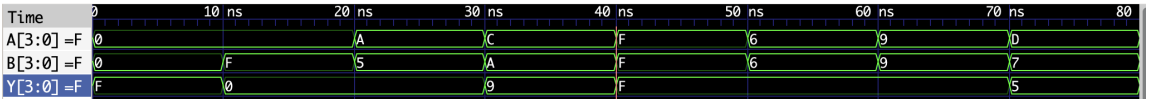

Figure 25: Xor Circuit with marker at 2500ns

10 Addition Circuit

The Addition circuit takes two 4-bit inputs A and B along with a 1-bit carry in to extend the output range from 15 to 31. The circuit's outputs include a 4-bit output called Sum, along with a 1-bit carry out to accommodate the extended range. The has an intermediary 5-bit variable, full sum, which is used to extract carry out and partial sum. First the full sum is found by adding A, B, and carry in. Then, Sum is assigned to the 4 lower bits of full sum. Then, carry out is assigned to the 5th bit of full sum.

```
1 `ifndef ADDITION_V
2 `define ADDITION_V
3
4 module addition (
5     input [3:0] A,          // 4-bit input A
6     input [3:0] B,          // 4-bit input B
7     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

`endif
```

To test the Addition circuit, we have created two registers A and B, a register carry in, a wire Sum, and a wire carry out. We then perform 5 tests with A, B, and carry in at various values. We will know that the Addition circuit is working as expected if carry out plus Sum produces correct results for every combination of A, B, and carry in that we have tested. For example, if A is 1000, B is 1000, and carry in is 1 , then the expected result: Sum is 0001, carry out is 1 for a decimal value of 17.

```
`timescale 1ns / 1ns
`include "addition.v"  // Include the addition module

module addition_tb;

    reg [3:0] A, B;       // 4-bit inputs
    reg carry_in;         // 1-bit carry in
    wire [3:0] Sum;       // 4-bit sum output
    wire carry_out;       // 1-bit carry out

    // Instantiate the addition module
    addition uut (
        .A(A),
        .B(B),
        .carry_in(carry_in),
        .Sum(Sum),
        .carry_out(carry_out)
```

```
18      );
19
20      // Test cases
21      initial begin
22          // Initialize dump file for waveform
23          $dumpfile("addition_tb.vcd");
24          $dumpvars(0, addition_tb);
25
26          // Test 1: A = 4'b0001, B = 4'b0010, carry_in = 0
27          A = 4'b0001;
28          B = 4'b0010;
29          carry_in = 1'b0;
30          #10;
31          $display("Test 1: A = %b, B = %b, carry_in = %b => Sum = %b,
    carry_out = %b", A, B, carry_in, Sum, carry_out);
32
33          // Test 2: A = 4'b1111, B = 4'b0001, carry_in = 0 (expect carry_out
     = 1)
34          A = 4'b1111;
35          B = 4'b0001;
36          carry_in = 1'b0;
37          #10;
38          $display("Test 2: A = %b, B = %b, carry_in = %b => Sum = %b,
    carry_out = %b", A, B, carry_in, Sum, carry_out);
39
40          // Test 3: A = 4'b1010, B = 4'b0101, carry_in = 0
41          A = 4'b1010;
42          B = 4'b0101;
43          carry_in = 1'b0;
44          #10;
45          $display("Test 3: A = %b, B = %b, carry_in = %b => Sum = %b,
    carry_out = %b", A, B, carry_in, Sum, carry_out);
46
47          // Test 4: A = 4'b1000, B = 4'b1000, carry_in = 1 (expect carry_out
     = 1)
48          A = 4'b1000;
49          B = 4'b1000;
50          carry_in = 1'b1;
51          #10;
52          $display("Test 4: A = %b, B = %b, carry_in = %b => Sum = %b,
    carry_out = %b", A, B, carry_in, Sum, carry_out);
53
```

```
54          // Test 5: A = 4'b0000, B = 4'b0000, carry_in = 1 (expect carry_out
        = 0)
55          A = 4'b0000;
56          B = 4'b0000;
57          carry_in = 1'b1;
58          #10;
59          $display("Test 5: A = %b, B = %b, carry_in = %b => Sum = %b,
       carry_out = %b", A, B, carry_in, Sum, carry_out);
60
61          // Finish simulation
62          $finish;
63      end
64
65  endmodule
```

10.1 Addition Circuit Waveform

At 2ns, A is 1 and B is 2, so Sum is 3 with carry in and carry out at 0.

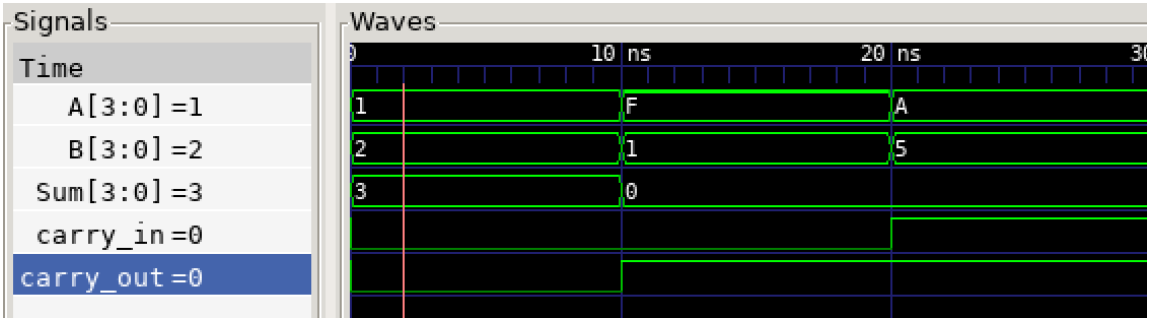

Figure 26: Addition Circuit with marker at 2ns

At 20ns, A is A(1010) and B is 5(0101), so Sum is F(1111) with carry in and carry out at 0.

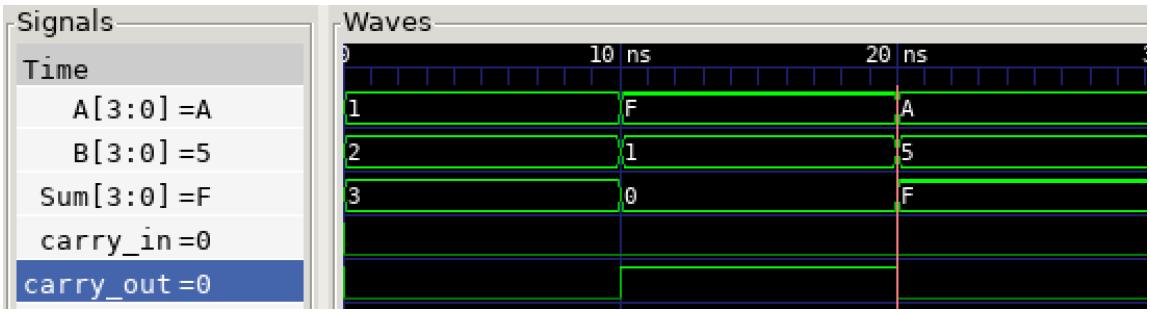

Figure 27: Addition Circuit with marker at 20ns

At 30ns, A and B are both 8(1000), so Sum is 1 with carry in and carry out at 1, resulting in a decimal value of 17.

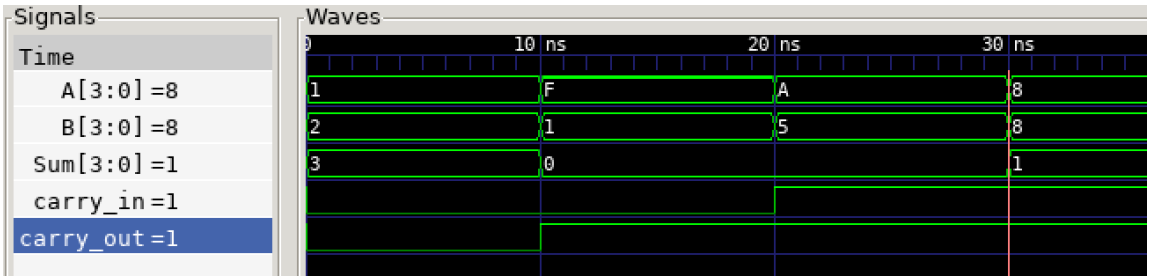

Figure 28: Addition Circuit with marker at 30ns

11 Multiplication Circuit

The Multiplication circuit takes two 4-bit inputs A and B, and produces two 4-bit outputs, product low and product high, representing the high and low 4 bits of the product of A and B. The circuit has an intermediary 8-bit variable, full product, which is used to extract the low and high 4 bits of the product. First, full product is assigned to the product of A and B. Then, product low is assigned the first 4 bits of full product. Then, product high is assigned the last 4 bits of full product.

```
1 `ifndef MULTIPLICATION_V
2 `define MULTIPLICATION_V
3
4 module multiplier_4bit (
5     input [3:0] A, B,
6     output [3:0] product_low,   // Lower 4 bits of the product
7     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

`endif
```

To test the Multiplication circuit, we have created two registers A and B,and two wires product low and product high. We then perform 5 tests with A and B at various values. We will know that the Multiplication circuit is working as expected if product low, product high produces correct results for every combination of A times B that we have tested. For example, if A is 1111 and B is 1111, then the expected result: product high is 1110, product low is 0001 for a decimal value of 225.

```
`timescale 1ns / 1ns
`include "multiplication.v"

module multiplication_tb;

    reg [3:0] A, B;
    wire [3:0] product_low;
    wire [3:0] product_high;

    multiplier_4bit uut (
        .A(A),
        .B(B),
        .product_low(product_low),
        .product_high(product_high)
    );

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

        A = 4'b0010; B = 4'b0011; #10;
        $display("A = %b, B = %b, product_low = %b, product_high = %b", A,
    B, product_low, product_high);

        A = 4'b0101; B = 4'b0101; #10;
```

```
25          $display("A = %b, B = %b, product_low = %b, product_high = %b", A,
    B, product_low, product_high);
26
27          A = 4'b0110; B = 4'b0011; #10;
28          $display("A = %b, B = %b, product_low = %b, product_high = %b", A,
    B, product_low, product_high);
29
30          A = 4'b1001; B = 4'b0100; #10;
31          $display("A = %b, B = %b, product_low = %b, product_high = %b", A,
    B, product_low, product_high);
32
33          A = 4'b1111; B = 4'b1111; #10;
34          $display("A = %b, B = %b, product_low = %b, product_high = %b", A,
    B, product_low, product_high);
35
36          $display("Selected test cases completed");
37          $finish;
38      end
39
40  endmodule
```

11.1 Multiplication Circuit Waveform

At 2ns, A is 2 and B is 3, so product high is 0 and product low is 6.

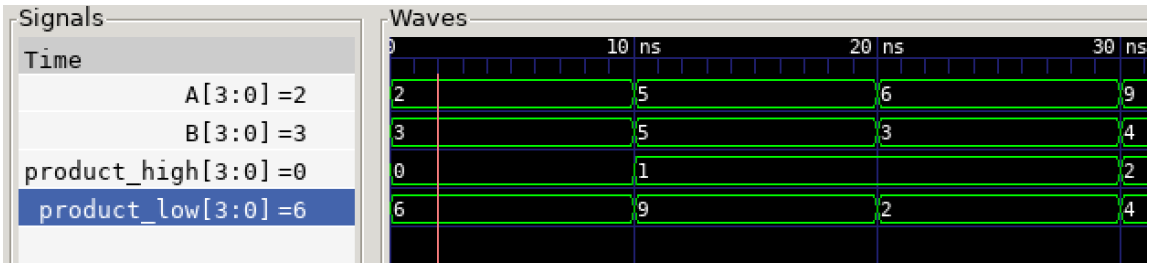

Figure 29: Multiplication Circuit with marker at 2ns

At 20ns, A is 6 and B is 3, so product upper is 0001 and product lower is 0010 for a decimal value of 18.

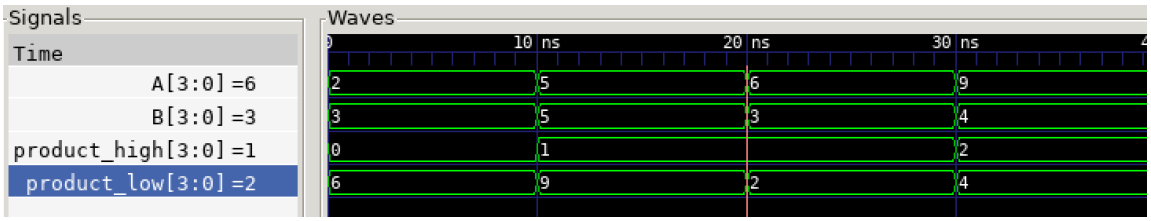

Figure 30: Multiplication Circuit with marker at 20ns

At 40ns, A and B are both F, so product upper is E and product lower is 0001 for a decimal value of 256.

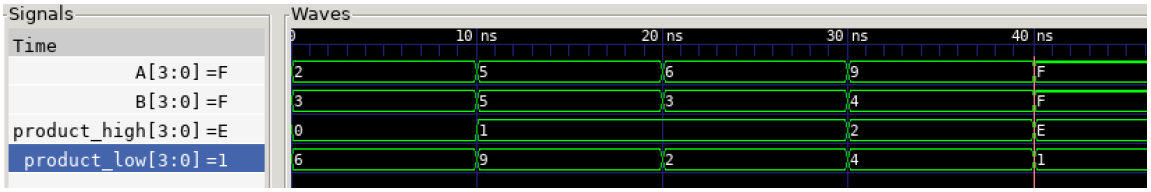

Figure 31: Multiplication Circuit with marker at 40ns

12 Subtraction Circuit

12.1 Subtraction Circuit Verilog Code

The Multiplication circuit takes two 4-bit inputs A and B, and produces one 4-bit output, Y with the difference of the two

```
1  `ifndef SUBTRACTION_V
2  `define SUBTRACTION_V
3
4  module subtraction (Y, A, B);
5      output [3:0] Y;  // 4-bit output
6      input  [3:0] A;  // 4-bit input A
7      input  [3:0] B;  // 4-bit input B
8
9      assign Y = A - B;  // Perform 4-bit subtraction
10
11 endmodule
12
13 `endif
```

To test the Subtraction circuit, we have created two registers A and B and a wire Y for output. We then perform 5 tests with A and B at various values. We will know that the Subtraction circuit is working as expected if it produces correct results for every combination of A times B that we have tested. For example, if A is 1111 and B is 1111, then the expected result: Y is 0000

```
1  `timescale 1ns / 1ns
2  `include "subtraction.v"
3
4  module subtraction_tb;
5
6      reg [3:0] A;      // 4-bit input A
7      reg [3:0] B;      // 4-bit input B
8      wire [3:0] Y;     // 4-bit output Y (result of A - B)
9
10     subtraction uut (Y, A, B);
11
12     initial begin
13         $dumpfile("subtraction_tb.vcd");  // Holds output waveform
14         $dumpvars(0, subtraction_tb);
15
16         // Test cases for 4-bit subtraction
17         A = 4'b0001; B = 4'b0001; #10;  // Expected Y: 0000
18         A = 4'b1010; B = 4'b0011; #10;  // Expected Y: 0111
19         A = 4'b1100; B = 4'b0101; #10;  // Expected Y: 1001
20         A = 4'b1111; B = 4'b1110; #10;  // Expected Y: 0001
21         A = 4'b0101; B = 4'b1001; #10;  // Expected Y: Overflow
22
23         $display("Testing subtraction");
24     end
25
26 endmodule
```

12.2 Subtraction Circuit Waveform

At 0ns, A is 1h (0001) and B is 1h (0001), so Y = 0h (0000)

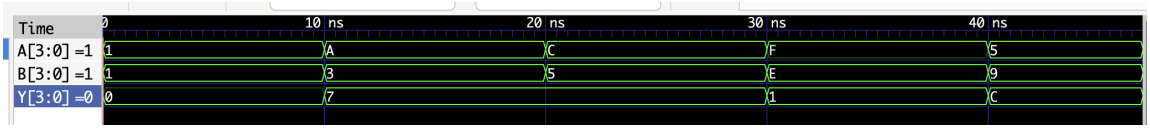

Figure 32: Subtraction Circuit with marker at 0ns

At 0ns, A is Ch (1100) and B is 5h (0101), so Y = 7h (0111)

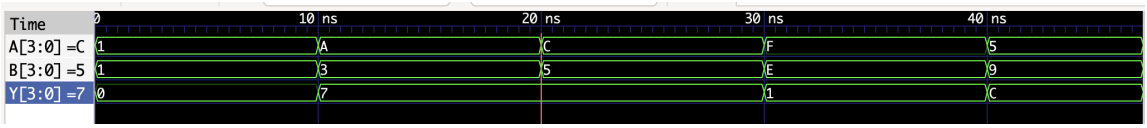

Figure 33: Subtraction Circuit with marker at 20ns

At 0ns, A is 5h (0101) and B is 9h (1001), so Y = Ch (1100)

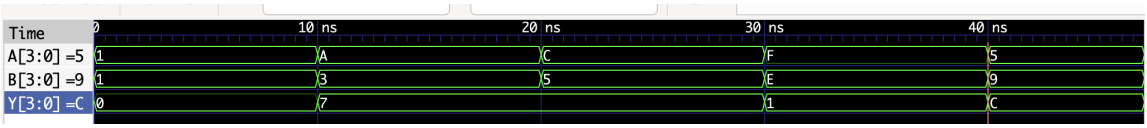

Figure 34: Multiplication Circuit with marker at 40ns

13 Division Circuit

The Division circuit takes two 4-bit inputs A and B, and produces two 4-bit outputs, quotient (Q) and remainder (R). The circuit first calculates how many times B can go into A, becoming Q, then it takes Q amount of B's away from A and the resulting number is R.

```
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
```

To test the Division circuit, we have created two registers A and B,and two wires quotient (Q) and remainder (R). We then perform 5 tests with A and B at various values. We will know that the Division circuit is working as expected if quotient and remainder produces correct results for every combination of A times B that we have tested.

```
`timescale 1ns / 1ps

module division_tb;

    reg [3:0] dividend;
    reg [3:0] divisor;

    wire [3:0] quotient;
    wire [3:0] remainder;
    wire valid;

    division uut (
        .dividend(dividend),
        .divisor(divisor),
```

```
        .quotient(quotient),
        .remainder(remainder),
        .valid(valid)
    );

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

        $monitor("Time: %0t | Dividend: %b | Divisor: %b | Quotient: %b | Remainder: %b | Valid: %b",
                 $time, dividend, divisor, quotient, remainder, valid);

        // Test 1: 0 / 1
        dividend = 4'b0000; // 0
        divisor = 4'b0001;  // 1
        #10; // Expected: Q = 0, R = 0, Valid = 1

        // Test 2: 1 / 1
        dividend = 4'b0001; // 1
        divisor = 4'b0001;  // 1
        #10; // Expected: Q = 1, R = 0, Valid = 1

        // Test 3: 4 / 2
        dividend = 4'b0100; // 4
        divisor = 4'b0010;  // 2
        #10; // Expected: Q = 2, R = 0, Valid = 1

        // Test 4: 7 / 4
        dividend = 4'b0111; // 7
        divisor = 4'b0100;  // 4
        #10; // Expected: Q = 1, R = 3, Valid = 1

        // Test 5: 10 / 3
        dividend = 4'b1010; // 10
        divisor = 4'b0011;  // 3
        #10; // Expected: Q = 3, R = 1, Valid = 1

        // Test 6: Division by zero
        dividend = 4'b1001; // 9
        divisor = 4'b0000;  // 0
        #10; // Expected: Q = 0, R = 9, Valid = 0

```

```
57          $finish;
58      end
59
60  endmodule
```

13.1 Multiplication Circuit Waveform

At 0ns, A is 0 and B is 1, so the quotient is 0 and the remainder is also 0.

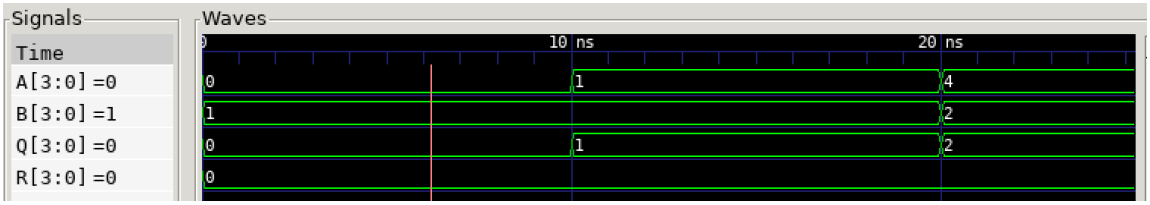

Figure 35: Division Circuit with marker at 0ns

At 20ns, A is 4 and B is 2, so the quotient is 2 and the remainder is 0.

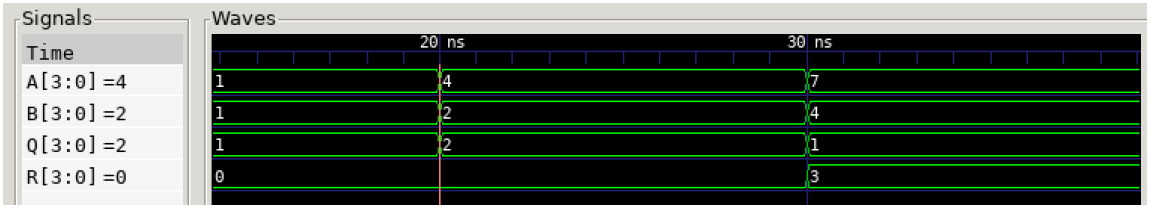

Figure 36: Division Circuit with marker at 20ns

At 30ns, A is 7 and B is 4, so the quotient is 1 and the remainder is 3.

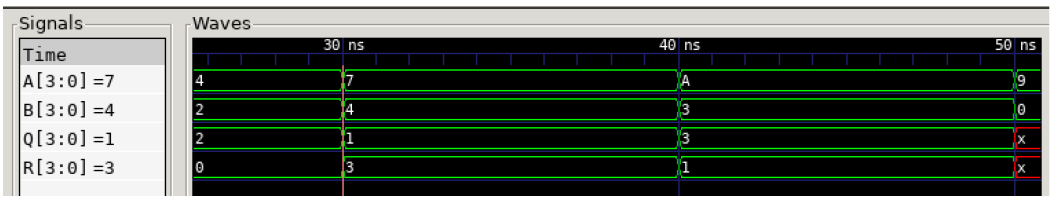

Figure 37: Division Circuit with marker at 30ns

14 Conclusion

This project introduced the design and testing of a basic 4-bit Arithmetic Logic Unit (ALU). 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.