PROJECT FINAL STEP

GROUP TOMATO

Ulises Espinoza-Gonzalez and Jack Stolpman

CS.3339 Computer Architecture

Texas State University

December 2, 2024

1 Introduction

Digital logic and circuits are an essential part of understanding the hardware components of any electronic device. Hardware descriptive languages such as Verilog, introduce us to the process of designing a new computer circuit from scratch. In this section of the project, we will be incorporating various logical gates, shift circuits, and arithmetic operations to generate simulation waveforms. Some goals that we as a group share is to improve our understanding of the methodology of designing new computer circuits from scratch. We also wish to familiarize ourselves with the Verilog programming language, which is new to us.

2 Verilog Code

In this final step of the project, we will be implementing every ALU module from part 2 and creating a control circuit. These modules include logical AND, NAND, OR, NOR, XOR, NOT, 4-bit input, 4-bit shift register, and various arithmetic operation circuits. We will bring all of these modules together to form a control circuit for the ALU. Below will be a description of each module.

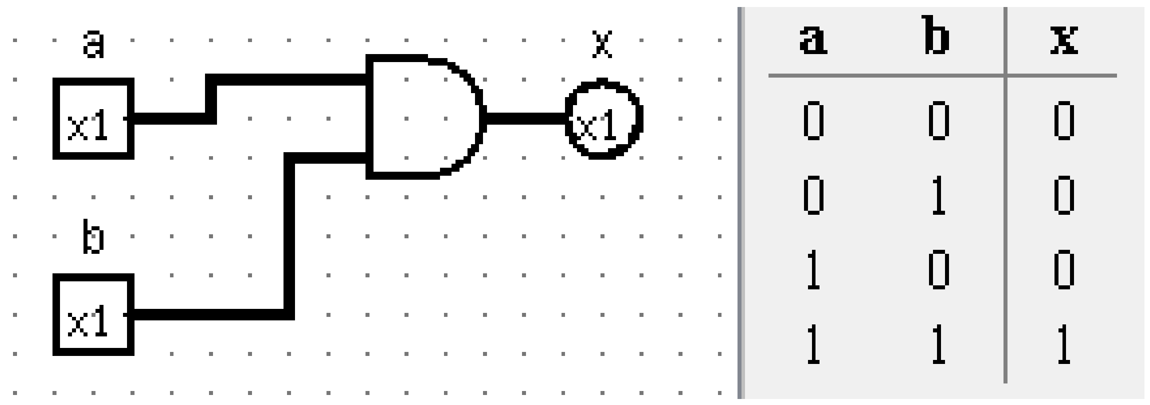
3 And Circuit

The And circuit takes two inputs, A and B, with an output X. In order for the output of this gate to be 1, both inputs must also be 1. This logic gate works well with operations that require all conditions to be met simultaneously.

```verilog
`include "and_gate.v"

module and_gate_tb;
    reg [3:0] a, b;
    wire [3:0] and_out;

    and_gate G1 (.a(a), .b(b), .and_out(and_out));

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

        a = 4'b0000; b = 4'b0000; #10;
        a = 4'b1111; b = 4'b0001; #10;
        a = 4'b1010; b = 4'b1100; #10;
        a = 4'b1111; b = 4'b1111; #10;

        $finish;
    end
endmodule
```

To test the And circuit, we have created two registers, A and B, as well as a wire X. This way we can take two inputs at a time and test each possible input for the circuit. If it's working correctly, X should be equal to 1 for all inputs only when both A and B are equal to 1.

```
module and_gate
(
    input [3:0] a, b,
    output [3:0] and_out
);
    assign and_out = a & b;
endmodule
```



a	b	x
0	0	0
0	1	0
1	0	0
1	1	1

Figure 1: And truth table and gate

3.1 Waveform Tests

The last section we will showcase the waveforms created using our testbenches for each circuit we coded in Verilog. We used GTKWave to create these waveforms.

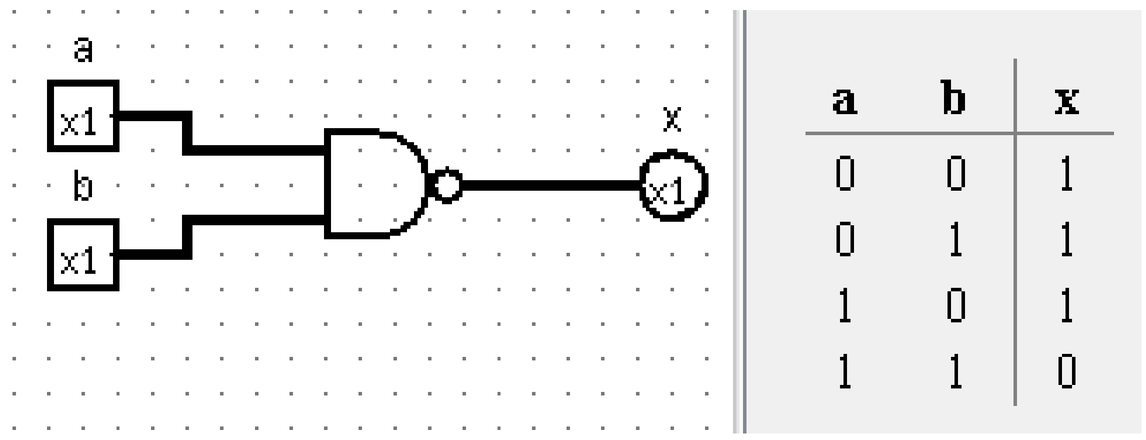
4 Nand Circuit

The Nand circuit, similarly to the And circuit, takes two inputs- A and B, with an output X. This output will only shoot back a 1 assuming that both A and B are not equal to 1, hence the Nand (Not and) logic gate.

```
‘include "nand_gate.v"

module nand_gate_tb;
    reg [3:0] a, b;
    wire [3:0] nand_out;

    nand_gate G1 (.a(a), .b(b), .nand_out(nand_out));

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

        a = 4’b0000; b = 4’b0000; #10;
        a = 4’b1111; b = 4’b0001; #10;
        a = 4’b1010; b = 4’b1100; #10;
        a = 4’b1111; b = 4’b1111; #10;

        $finish;
    end
endmodule
```

To test the Nand circuit, we have created two registers, A and B, as well as a wire X. This way we are able to take two inputs at a time and test each possible input for the circuit. If it’s working correctly, X should be equal to 1 for all inputs except when both A and B are equal to 1. This gate is often used in circuits as a building block as it can be combined to create other logic gates.

```
module nand_gate
(
    input [3:0] a, b,
    output [3:0] nand_out
);
    assign nand_out = ~(a & b);
endmodule
```



a	b	x
0	0	1
0	1	1
1	0	1
1	1	0

Figure 2: Nand truth table and gate

4.1 Waveform Tests

The last section we will showcase the waveforms created using our testbenches for each circuit we coded in Verilog. We used GTKWave to create these waveforms.

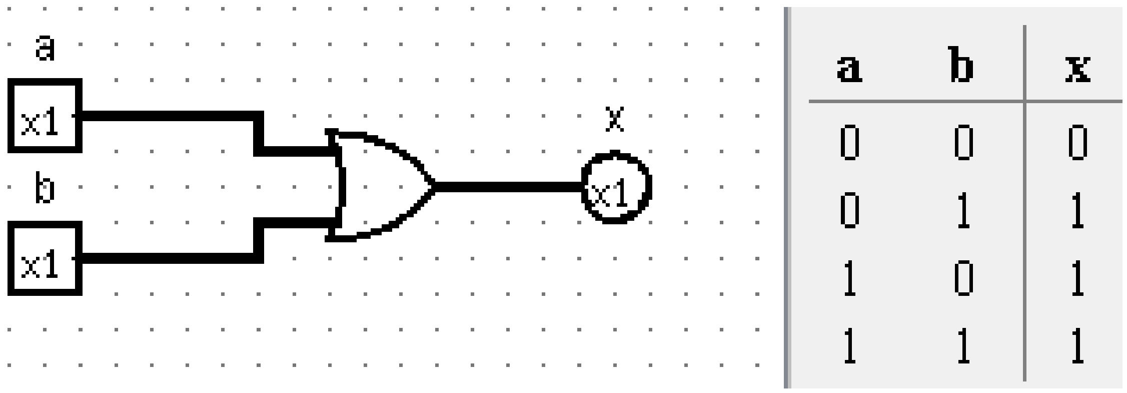
5 Or Circuit

The Or circuit takes two inputs, A and B, with an output X. This output will only shoot back a 1 assuming that either A, B. or both are equal to 1. If both a and b are 0, the output will also be 0. This gate is often used in circuits were an output is required when at least one condition is true.

```
‘include "or_gate.v"

module or_gate_tb;
    reg [3:0] a, b;
    wire [3:0] or_out;

    or_gate G1 (.a(a), .b(b), .or_out(or_out));

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

        a = 4'b0000; b = 4'b0000; #10;
        a = 4'b1111; b = 4'b0001; #10;
        a = 4'b1010; b = 4'b1100; #10;
        a = 4'b1111; b = 4'b1111; #10;

        $finish;
    end
endmodule
```

To test the Or circuit, we have created two registers, A and B, as well as a wire X. This way we are able to take two inputs at a time and test each possible input for the circuit. If it's working correctly, X should be equal to 1 whenever a or b (or both) are equal to 1.

```
module or_gate
(
    input [3:0] a, b,
    output [3:0] or_out
);
    assign or_out = a | b;
endmodule
```



a	b	x
0	0	0
0	1	1
1	0	1
1	1	1

Figure 3: Or truth table and gate

5.1 Waveform Tests

The last section we will showcase the waveforms created using our testbenches for each circuit we coded in Verilog. We used GTKWave to create these waveforms.

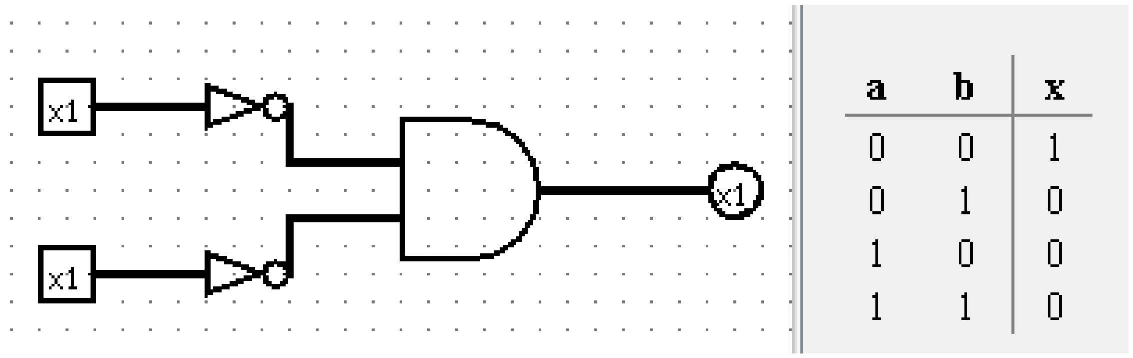
6 Nor Circuit

The Nor circuit takes two inputs, A and B, with an output X. This output will only shoot back a 1 assuming that both A and B are equal to 0, hence the Nor logic gate. For the output to return a high value, both inputs need to be low. This gate is often used in circuits where the output is true when all inputs are false.

```
`include "nor_gate.v"

module nor_gate_tb;
    reg [3:0] a, b;
    wire [3:0] nor_out;

    nor_gate G1 (.a(a), .b(b), .nor_out(nor_out));

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

        a = 4'b0000; b = 4'b0000; #10;
        a = 4'b1111; b = 4'b0001; #10;
        a = 4'b1010; b = 4'b1100; #10;
        a = 4'b1111; b = 4'b1111; #10;

        $finish;
    end
endmodule
```

To test the Nor circuit, we have created two registers, A and B, as well as a wire X. This way we are able to take two inputs at a time and test each possible input for the circuit. If it's working correctly, X should be equal to 1 for all inputs except when both A and B are equal to 0.

```
module nor_gate
(
    input [3:0] a, b,
    output [3:0] nor_out
);
    assign nor_out = ~(a | b);
endmodule
```



a	b	x
0	0	1
0	1	0
1	0	0
1	1	0

Figure 4: Nor truth table and gate

6.1 Waveform Tests

The last section we will showcase the waveforms created using our testbenches for each circuit we coded in Verilog. We used GTKWave to create these waveforms.

7 Xor Circuit

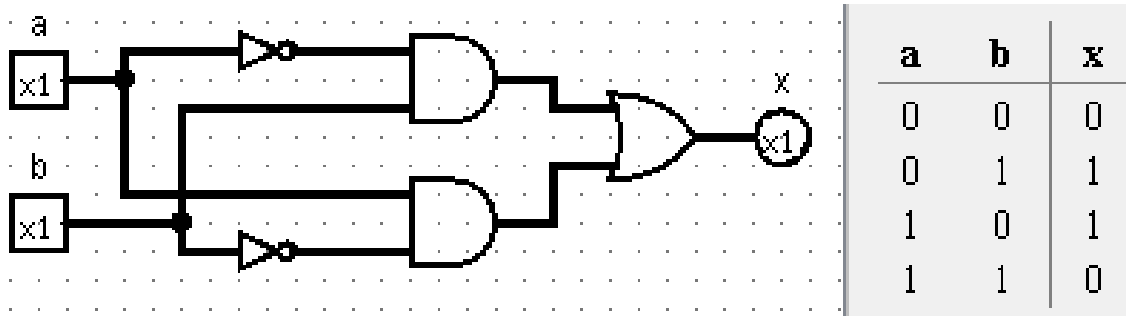
The Xor circuit takes two inputs, A and B, with an output X. This output will only shoot back a 1 assuming that only one of the inputs is 1. if A and B are different (A = 1, B = 0), then the output will be 1.

```
`include "xor_gate.v"

module xor_gate_tb;
    reg [3:0] a, b;
    wire [3:0] xor_out;

    xor_gate G1 (.a(a), .b(b), .xor_out(xor_out));

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

        a = 4'b0000; b = 4'b0000; #10;
        a = 4'b1111; b = 4'b0001; #10;
        a = 4'b1010; b = 4'b1100; #10;
        a = 4'b1111; b = 4'b1111; #10;

        $finish;
    end
```

```
endmodule
```

To test the Xor circuit, we have created two registers, A and B, as well as a wire X. This way we are able to take two inputs at a time and test each possible input for the circuit. If it's working correctly, X should be equal to 1 only when one of the inputs is equal to 1. This gate is often used in circuits that require distinguishing between inputs.

```
module xor_gate
(
    input [3:0] a, b,
    output [3:0] xor_out
);
    assign xor_out = a ^ b;
endmodule
```



a	b	x
0	0	0
0	1	1
1	0	1
1	1	0

Figure 5: Xor truth table and gate

7.1 Waveform Tests

The last section we will showcase the waveforms created using our testbenches for each circuit we coded in Verilog. We used GTKWave to create these waveforms.

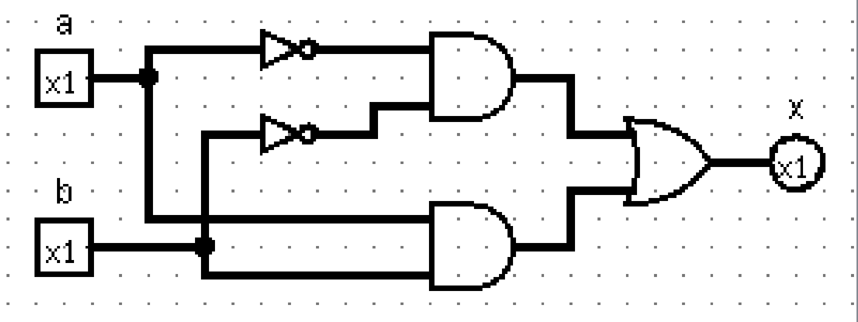
8 Xnor Circuit

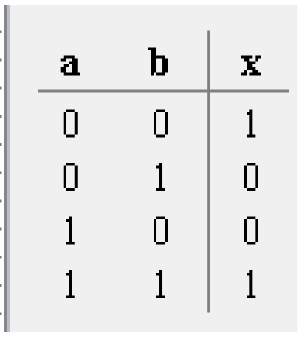
The Xnor circuit takes two inputs, A and B, with an output X. This output will only shoot back a 1 assuming that both A and B are the same value (A = 1, B = 1 || A = 0, B = 0). If the value for A is different from that of B, then the output will be 0.

```
`include "xnor_gate.v"

module xnor_gate_tb;
    reg [3:0] a, b;
    wire [3:0] xnor_out;

    xnor_gate G1 (.a(a), .b(b), .xnor_out(xnor_out));

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

        a = 4'b0000; b = 4'b0000; #10;
        a = 4'b1111; b = 4'b0001; #10;
        a = 4'b1010; b = 4'b1100; #10;
        a = 4'b1111; b = 4'b1111; #10;

        $finish;
    end
endmodule
```

To test the Xnor circuit, we have created two registers, A and B, as well as a wire X. This way we can take two inputs at a time and test each possible input for the circuit. If it's working correctly, X should be equal to 1 only when A and B have the same value. This gate is used in circuits where we need the output to be true when both inputs match.

```
module xnor_gate
(
    input [3:0] a, b,
    output [3:0] xnor_out
);
    assign xnor_out = ~(a ^ b);
endmodule
```





a	b	x
0	0	1
0	1	0
1	0	0
1	1	1

Figure 6: Xnor truth table and gate

8.1 Waveform Tests

The last section we will showcase the waveforms created using our testbenches for each circuit we coded in Verilog. We used GTKWave to create these waveforms.

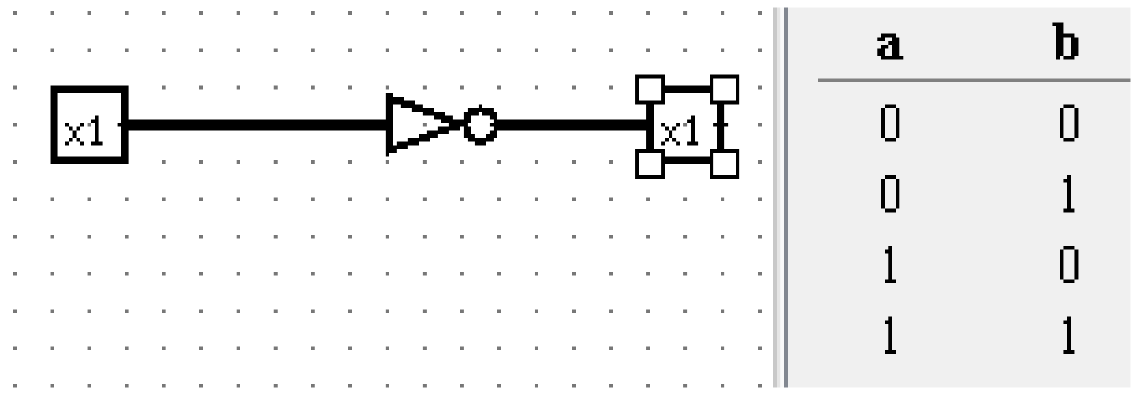
9 Not Circuit

The Not circuit takes two inputs: A and B.The Not gate will revert the value inputted. If the value of A is 1, then the value of B will be 0 and vice versa.

```
`include "not_gate.v"

module not_gate_tb;
    reg [3:0] a, b;
    wire [3:0] not_a, not_b;

    not_gate G1 (.a(a), .b(b), .not_a(not_a), .not_b(not_b));

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

        a = 4'b0000; b = 4'b0001; #10;
        a = 4'b1010; b = 4'b0101; #10;
        a = 4'b1111; b = 4'b1111; #10;

        $finish;
    end
endmodule
```

To test the Not circuit, we have created two registers, A and B. This way we can take two inputs at a time and test each possible input for the circuit. If it's working correctly, A should be equal to 1 whenever B is equal to 0.

```
module not_gate
(
    input [3:0] a, b,
    output [3:0] not_a, not_b
);
    assign not_a = ~a;
    assign not_b = ~b;
endmodule
```



a	b
0	0
0	1
1	0
1	1

Figure 7: Not truth table and gate

9.1 Waveform Tests

The last section we will showcase the waveforms created using our testbenches for each circuit we coded in Verilog. We used GTKWave to create these waveforms.

10 2x4-bit input Circuit

The 4 bit input circuit takes two inputs, fourin and fourout. We create a shift input.

```
`include "four_bit_input.v"

module four_bit_input_tb;

    reg clock;
    reg [3:0] four_in1, four_in2;
    wire [3:0] four_out1, four_out2;

    four_bit_input G1 (.clock(clock), .four_in1(four_in1), .four_in2(
   four_in2), .four_out1(four_out1), .four_out2(four_out2));

    always begin
        #5 clock = ~clock; // Clock generation
    end

    initial begin
        clock = 0;
        four_in1 = 4'b0000; four_in2 = 4'b0000; #10; // Initial values
        four_in1 = 4'b1010; four_in2 = 4'b0101; #10; // Test case 1
        four_in1 = 4'b1100; four_in2 = 4'b0011; #10; // Test case 2
        four_in1 = 4'b1111; four_in2 = 4'b1001; #10; // Test case 3
        four_in1 = 4'b0001; four_in2 = 4'b1000; #10; // Test case 4

        $finish;
    end

endmodule
```

To test this 4 bit circuit, we create a register that takes in a 4 bit value. This value works alongside the clock which when executed using the @posedge and the clock signal goes from high to low, the input will be transfered into the output.

```
module four_bit_input
(
    input clock,
    input [3:0] four_in1, four_in2,  // 2x4-bit input signals ([3:0])
    output reg [3:0] four_out1, four_out2 // 2x4-bit output signals ([3:0])
);

    always @(posedge clock) begin
        four_out1 <= four_in1;
        four_out2 <= four_in2;
    end

endmodule
```

10.1 Waveform Tests

The last section we will showcase the waveforms created using our testbenches for each circuit we coded in Verilog. We used GTKWave to create these waveforms.

11 2x4-bit output Shift Circuit

The 4 bit input circuit takes two inputs, datain and dataout. We use a clock/ reset and create a shift input in order to determine the shift direction (high = 1 = right, low = 0 = left). On the rising edge of the reset, we set the output to 4'b0000 (0 in verilog) which clears the shift register. While this process is happening, if the shift is high, the data is shifted right by 1 bit and when the shift is low, the data is shifted left by 1 bit.

```
`include "four_bit_shift_circuit.v"

module four_bit_shift_circuit_tb;
```

```

    reg clock, reset, right_shift;
    reg [3:0] data_in1, data_in2;
    wire [3:0] data_out1, data_out2;

    four_bit_shift_circuit G1 (.clock(clock), .reset(reset), .right_shift(
   right_shift), .data_in1(data_in1), .data_in2(data_in2), .data_out1(
   data_out1), .data_out2(data_out2));

    always begin
        #5 clock = ~clock; // Clock generation
    end

    initial begin
        clock = 0;
        reset = 1; // Start with reset
        right_shift = 0; // Initial shift direction
        data_in1 = 4'b0000; data_in2 = 4'b0000; #10; // Initial values

        reset = 0; #10; // Release reset

        // Test shifting
        data_in1 = 4'b1010; data_in2 = 4'b0101; #10; // Load test values
        right_shift = 0; #10; // Left shift
        right_shift = 1; #10; // Right shift

        // Testing reset functionality
        reset = 1; #10; // Apply reset
        reset = 0; #10; // Release reset

        data_in1 = 4'b1100; data_in2 = 4'b0011; #10; // Load more values
        right_shift = 0; #10; // Left shift again
        right_shift = 1; #10; // Right shift again

        $finish;
    end

endmodule
```

To test the 4bit shift circuit, we have created two registers, A and B, as well as a wire X. We also create a.

```
module four_bit_shift_circuit
(
    input clock,
    input reset,
    input right_shift,
    input [3:0] data_in1, data_in2,
    output reg [3:0] data_out1, data_out2
);

    always @(posedge clock or posedge reset) begin
        if (reset) begin
            data_out1 <= 4'b0000; // reset output1 to 0
            data_out2 <= 4'b0000; // reset output2 to 0
        end
        else begin
            if (right_shift) begin
                data_out1 <= data_in1 >> 1; // Shift right
                data_out2 <= data_in2 >> 1; // Shift right
            end
            else begin
                data_out1 <= data_in1 << 1; // Shift left
                data_out2 <= data_in2 << 1; // Shift left
            end
        end
    end

endmodule
```

11.1 Waveform Tests

The last section we will showcase the waveforms created using our testbenches for each circuit we coded in Verilog. We used GTKWave to create these waveforms.

12 Addition

The addition module takes in two inputs - A and B - which are both 4-bit. These represent the numbers that will be added. We then create the sum output which is the 4-bit result of the addition. the carry in is a single bit representing any carry coming in from previous calculations. Due to us adding 4-bit values, we create a 5 bit wire called addition result which accounts for any overflow we may encounter.

```
`include "addition.v"

module addition_tb;
```

```verilog

    reg [3:0] a, b;
    reg carry_in;
    wire carry_out;
    wire [3:0] sum;

    addition Add (.a(a), .b(b), .carry_in(carry_in), .carry_out(carry_out),
     .sum(sum));

    initial begin
        // Test cases for addition
        a = 4'b0000; b = 4'b0000; carry_in = 0; #10; // 0 + 0 = 0
        a = 4'b0001; b = 4'b0001; carry_in = 0; #10; // 1 + 1 = 2
        a = 4'b0010; b = 4'b0010; carry_in = 0; #10; // 2 + 2 = 4
        a = 4'b0011; b = 4'b0001; carry_in = 0; #10; // 3 + 1 = 4

        $finish;
    end

endmodule
```

To test the addition of the 4-bit values, we assign the sum of A and B and the carry in to our 5 bit variable. Once assigned, the 5-bit wire stores the 4 least significant bits and stores them into our sum output. We finally assign the overflow bit to the carry out.

```verilog
module addition
(
    input [3:0] a, b,
    input carry_in,
    output carry_out,
    output [3:0] sum
);

    wire [4:0] addition_result;

    assign addition_result = a + b + carry_in;
    assign sum = addition_result[3:0];
    assign carry_out = addition_result[4];

endmodule
```

12.1 Waveform Tests

The last section we will showcase the waveforms created using our testbenches for each circuit we coded in Verilog. We used GTKWave to create these waveforms.

13 Subtraction

The subtraction module takes in two inputs - A and B - which are both 4-bit. These represent the numbers that will be added. We then create the difference output which is the 4-bit result of the subtraction. the carry in is a single bit that acts as a borrow input in our calculations. Due to us subtracting 4-bit values, we create a 5 bit wire called subtraction result which accounts for any overflow we may encounter.

```
‘include "subtraction.v"

module subtraction_tb;

    reg [3:0] a, b;
    reg carry_in;
    wire carry_out;
    wire [3:0] difference;

    subtraction Sub (.a(a), .b(b), .carry_in(carry_in), .carry_out(
   carry_out), .difference(difference));

    initial begin
        // Test cases for subtraction
        a = 4’b0100; b = 4’b0010; carry_in = 0; #10; // 4 - 2 = 2
        a = 4’b0010; b = 4’b0001; carry_in = 0; #10; // 2 - 1 = 1
        a = 4’b0001; b = 4’b0001; carry_in = 0; #10; // 1 - 1 = 0
        a = 4’b0000; b = 4’b0001; carry_in = 0; #10; // 0 - 1 = -1 (
   underflow)

        $finish;
    end

endmodule
```

To test the subtraction of the 4-bit values, we assign the difference of A and B and the carry in to our 5 bit variable. Once assigned, the 5-bit wire stores the 4 least significant bits which the difference output later recieves. The carry out will hold the borrow bit only if A < B.

```
module subtraction
```

```verilog
(
    input [3:0] a, b,
    input carry_in,
    output carry_out,
    output [3:0] difference
);

    wire [4:0] subtraction_result;

    assign subtraction_result = a - b - carry_in;
    assign difference = subtraction_result[3:0];
    assign carry_out = subtraction_result[4];

endmodule
```

13.1 Waveform Tests

The last section we will showcase the waveforms created using our testbenches for each circuit we coded in Verilog. We used GTKWave to create these waveforms

14 Multiplication

The multiplication module takes in two 4-bit inputs - A and B. The inputs represent the numbers that will be multiplied. We also create an output called product in order to store the value of the multiplication.

```verilog
`include "multiplication.v"

module multiplication_tb;

    reg [3:0] a, b;
    wire [7:0] product;

    multiplication Mul (.a(a), .b(b), .product(product));

    initial begin
        // Test cases for multiplication
        a = 4'b0001; b = 4'b0010; #10; // 1 * 2 = 2
        a = 4'b0010; b = 4'b0011; #10; // 2 * 3 = 6
        a = 4'b0001; b = 4'b0001; #10; // 1 * 1 = 1
        a = 4'b0011; b = 4'b0010; #10; // 3 * 2 = 6

        $finish;
    end

endmodule
```

In order to multiply the two 4-bit values, we have to ensure that our product can handle any overflow. We can achieve this by making the product an 8-bit value. This is because the multiplication of two 4-bit values may produce up to an 8-bit value. We then simply assign the multiplication of A and B to the product variable.

```
module multiplication
(
    input [3:0] a, b,
    output [7:0] product
);

    assign product = a * b;

endmodule
```

14.1 Waveform Tests

The last section we will showcase the waveforms created using our testbenches for each circuit we coded in Verilog. We used GTKWave to create these waveforms.

15 Division

The division module takes in two 4-bit inputs - A and B. The inputs represent the numbers that will be divided. We also create two 4-bit outputs which are the quotient and remainder. These will be used to store their respective values.

```
‘include "division.v"

module division_tb;

    reg [3:0] a, b;
    wire [3:0] quotient, remainder;

    division Div (.a(a), .b(b), .quotient(quotient), .remainder(remainder))
    ;

    initial begin
```

```
        // Test cases for division
        a = 4'b0100; b = 4'b0010; #10; // 4 / 2 = 2
        a = 4'b0001; b = 4'b0001; #10; // 1 / 1 = 1
        a = 4'b0010; b = 4'b0001; #10; // 2 / 1 = 2
        a = 4'b0000; b = 4'b0001; #10; // 0 / 1 = 0
        a = 4'b0100; b = 4'b0000; #10; // Division by zero

        $finish;
    end

endmodule
```

In order to divide the two 4-bit values, we simply assign the division of A and B to the quotient variable. We then use modulo in order to determine if there is a remainder and then assign the value to its corresponding output. We also use the modulo as a check to see if b is equal to 0. We do this in order to ensure that we aren't dividing A by 0.

```
module division
(
    input [3:0] a, b,
    output [3:0] quotient, remainder
);

    assign quotient = (b != 0) ? (a / b) : 4'b0000;
    assign remainder = (b != 0) ? (a % b) : 4'b0000;

endmodule
```

16 Control Circuit

The control circuit module was created with the purpose to perform various arithmetic and logical operations on two operands. We introduce 2 new variables: operation type which is our 4-bit input responsible for representing the type of operation the ALU will perform and controller which is our output responsible for signaling the ALU to perform a specific operation. We chose to make the operation type 4-bit because we want it to represent values ranging from 0 - 15. This is the case because we will be assigning binary values to each operation type. The always@(*) is a combinational logic block which is evaluated every time the values of the operation type input change. Inside this block we utilize a case statement in order to check the values operation type and to help us assign the value of the controller to fit the input.

We assign each ALU module with a binary value. 4'b0000 for the AND, 4'b0001 for the NAND, 4'b0010 for the OR, 4'b0011 for the NOR, 4'b0100 for the XOR, 4'b0101 for the XNOR, 4'b0110 for the NOT, 4'b0111 for the addition, 4'b1000 for the subtraction, 4'b1001 for the multiplication, 4'b 1010 for the division. Once we have these values, we are able to direct the controller to the correct module when controlling. The controller signals are the control inputs that direct the ALU to execute the chosen operation.

```verilog
// ALU modules control circuit
module control_circuit
(
    input [3:0] operation_type,
    output reg [3:0] controller
);

    always @(*) begin

        case (operation_type)

            4'b0000: controller = 4'b0000; // AND
            4'b0001: controller = 4'b0001; // NAND
            4'b0010: controller = 4'b0010; // OR
            4'b0011: controller = 4'b0011; // NOR
            4'b0100: controller = 4'b0100; // XOR
            4'b0101: controller = 4'b0101; // XNOR
            4'b0110: controller = 4'b0110; // NOT
            4'b0111: controller = 4'b0111; // Addition
            4'b1000: controller = 4'b1000; // Subtraction
            4'b1001: controller = 4'b1001; // Multiplication
            4'b1010: controller = 4'b1010; // Division

            default: controller = 4'b0000; // Default to AND

        endcase

    end

endmodule
```

Similar to previous test benches, we begin by creating some variables which are important for specific modules. These include a clock, reset, carry in, inputs, and our controller. We instantiate the control circuit which connects the inputs and the outputs which are the operation type and controller respectively. The use of the clock is important as it is used to synchronize the behavior of the control circuit. We initialize our operation type to the default (AND). The testing portion of the code consists of several test cases where the operation type is changed similar to before, but the a and b values are specified. These tests allow for us to check whether the control circuit generates the correct outputs for various modules. Using the previous 4-bit binary values stated, we assign values to a and b that will prove that the operation type is the correct one. For example, we set the operation type to 4'b0000, which we know corresponds to the AND gate. We set the operands (a and b) equal to an arbitrary value that we will keep consistent throughout the tests. The expected behavior of the AND operation is that a AND b results in 4'b0000. In order to make the waveforms more readable, we use a delay of 5 time units. This same process is repeated for different operations, each time setting the operation type to the corresponding binary value and using the values of a and b.

```
`include "ControlCircuit.v"

module Final_tb;

    reg clock, reset, carry_in;
    reg [3:0] a, b;
    reg [3:0] operation_type;

    wire [3:0] controller;

    control_circuit CC
    (
        .operation_type(operation_type),
        .controller(controller)
    );


    always begin

        #10 clock = ~clock;

    end

    initial begin
```

```

        $dumpfile("ModuleWaves.vcd"); // Waveform simulations
        $dumpvars(0, Final_tb);

        // Initialize signals
        clock = 0;
        reset = 1;
        operation_type = 4'b0000;

        // control circuit
        operation_type = 4'b0000; a = 4'b1010; b = 4'b0101; #5; // AND
        operation_type = 4'b0001; a = 4'b1010; b = 4'b0101; #5; // NAND
        operation_type = 4'b0010; a = 4'b1010; b = 4'b0101; #5; // OR
        operation_type = 4'b0011; a = 4'b1010; b = 4'b0101; #5; // NOR
        operation_type = 4'b0100; a = 4'b1010; b = 4'b0101; #5; // XOR
        operation_type = 4'b0101; a = 4'b1010; b = 4'b0101; #5; // XNOR
        operation_type = 4'b0110; a = 4'b1010; #5; // NOT

        // Test arithmetic operations
        operation_type = 4'b0111; a = 4'b0011; b = 4'b0001; carry_in = 0;
    #5; // Addition
        operation_type = 4'b1000; a = 4'b0100; b = 4'b0010; carry_in = 0;
    #5; // Subtraction
        operation_type = 4'b1001; a = 4'b0011; b = 4'b0010; #5; //
    Multiplication
        operation_type = 4'b1010; a = 4'b0100; b = 4'b0010; #20; //
    Division

        $finish;

    end

endmodule
```

16.1 Waveform Tests

The last section we will showcase the waveforms created using our testbenches for each circuit we coded in Verilog. We used GTKWave to create these waveforms.

17 Conclusion

In conclusion, we successfully created and tested various circuits using Verilog and GTK-Wave. Learning how to code in Verilog, at least for our purposes, was relatively simple with few difficulties. Using GTKWave was also fairly intuitive. The most difficulty our group has had so far was most likely creating this report in LaTeX, as we have some trouble with coding certain formatting as well as issues with figures floating around the document. Creating these logic circuits from scratch allowed us to work on the basic logic gates (AND, NAND, OR, NOR, XOR) that are so critical to all electronics. This final portion of this project has been the culmination of everything we have learned throughout this project. We learned how to create a control circuit in order to test and measure every single module we have written. The use of Verilog and GTKWave works as a stepping stone to furthering our engineering careers and building on our knowledge of both the software and hardware components of electronics.

18 Waveforms

18.1 And Circuit Waveform

At 0 s, we can see that both A and B are 0, so the output is 0

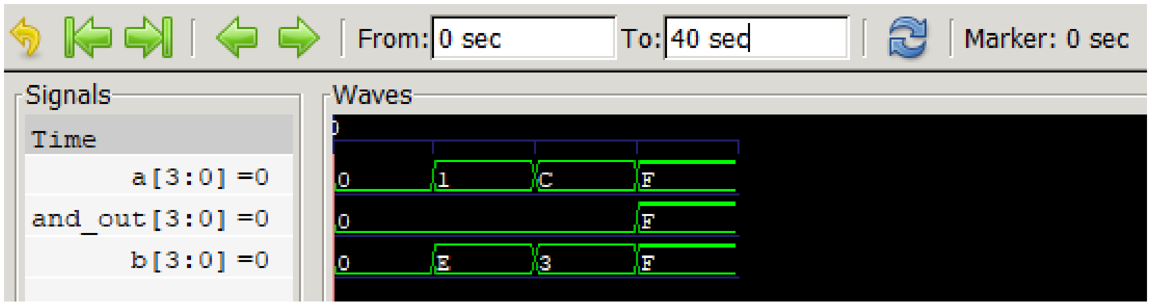


Figure 8: And Circuit with marker at 0s

At 10 s, A is 1 and B is E, so the output is 0

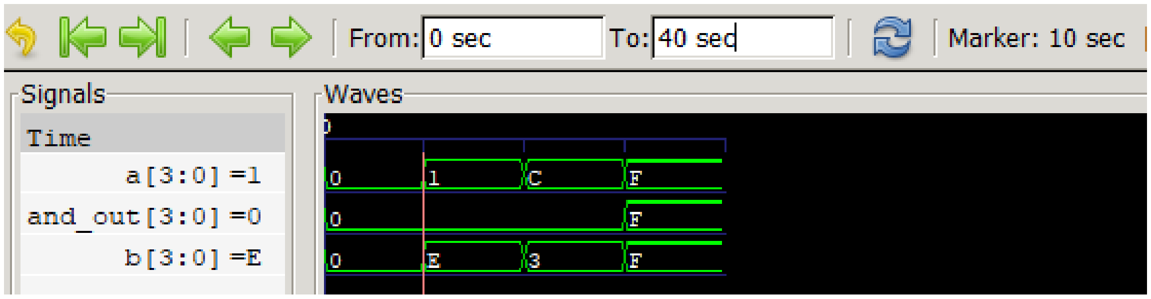


Figure 9: And Circuit with marker at 10s

At 20 s, A is 1 and B is E, so the output is 0

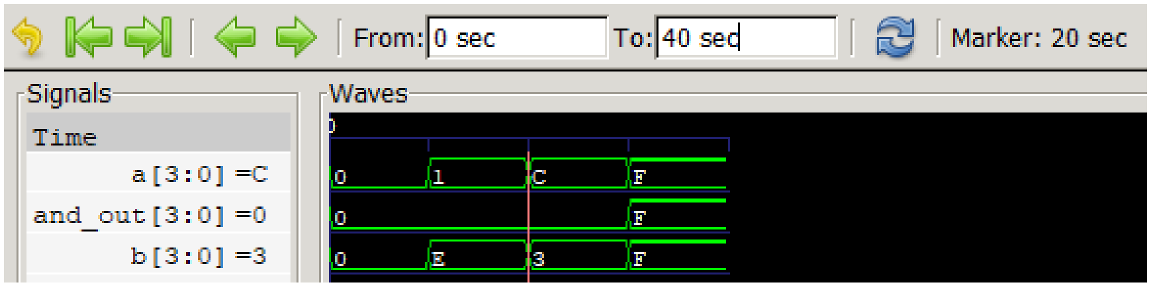


Figure 10: And Circuit with marker at 20s

At 30 s, A is F and B is F, so the output is F

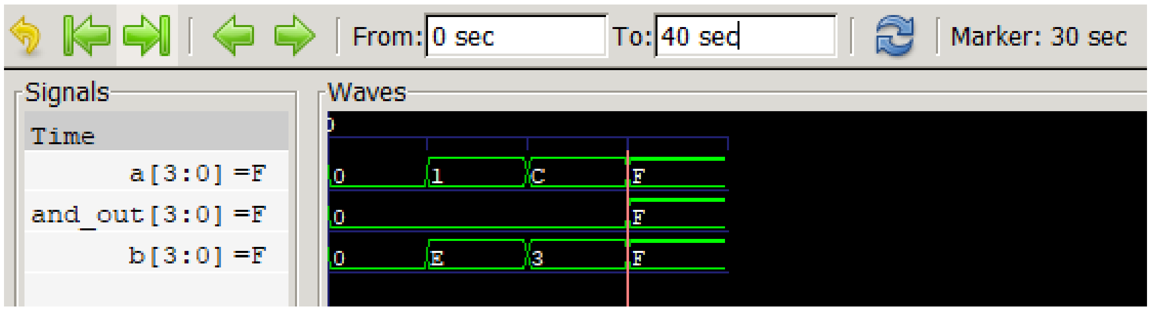


Figure 11: And Circuit with marker at 30s

18.2 Nand Circuit Waveform

At 0 s, we can see that both A and B are 0, so the output is F

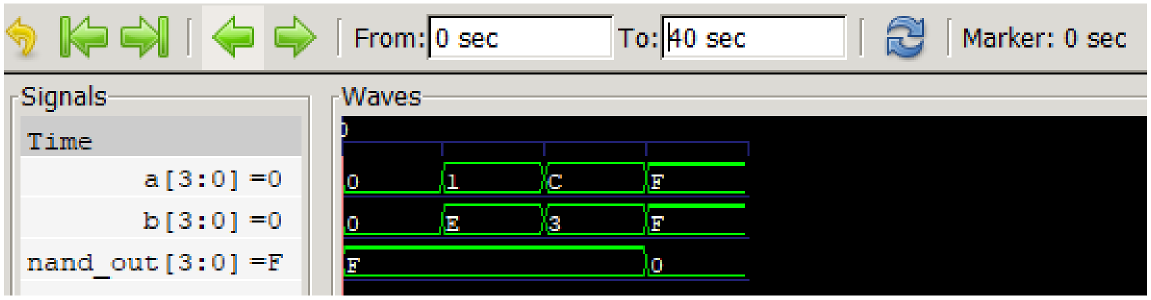
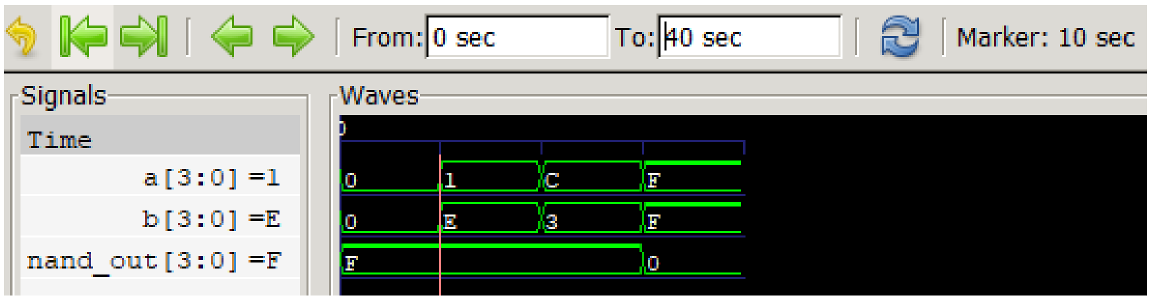


Figure 12: Nand Circuit with marker at 0s

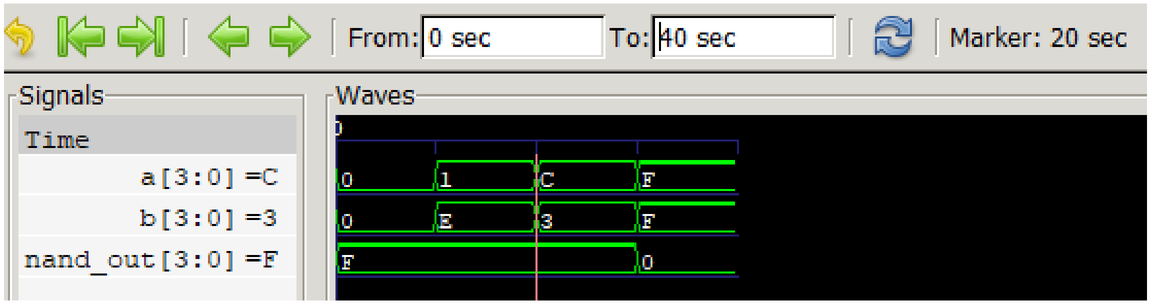
At 10 s, A is 1 and B is E, so the output is F



```
a[3:0] =1
b[3:0] =E
nand_out[3:0] =F
```

Figure 13: Nand Circuit with marker at 10s

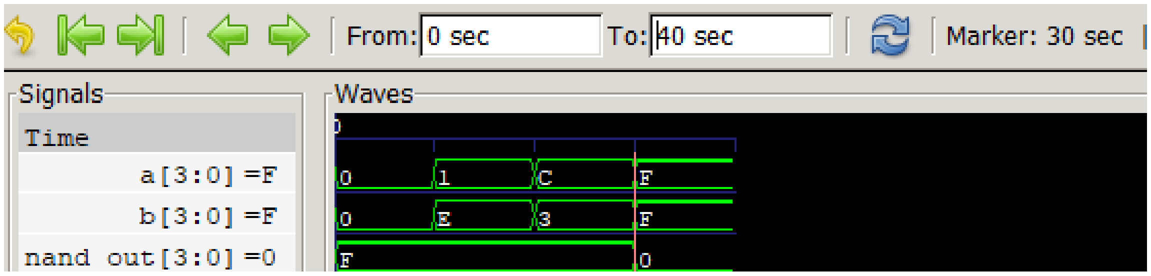
At 20 s, A is C and B is 3, so the output is F



```
a[3:0] =C
b[3:0] =3
nand_out[3:0] =F
```

Figure 14: Nand Circuit with marker at 20s

At 30 s, A is F and B is F, so the output is 0

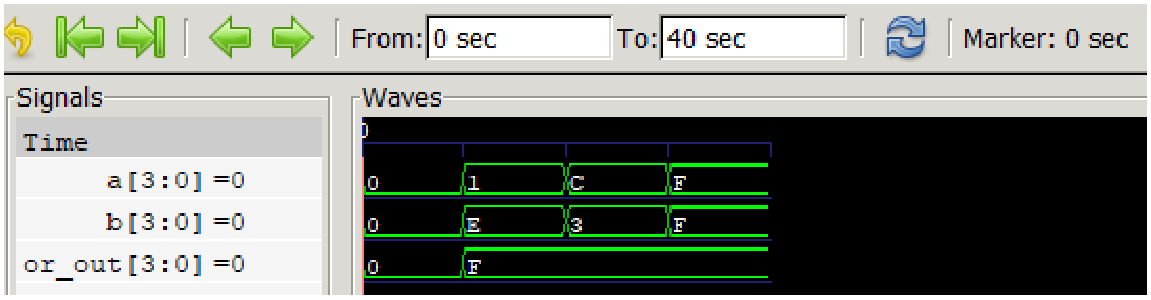


```
a[3:0] =F
b[3:0] =F
nand_out[3:0] =0
```

Figure 15: Nand Circuit with marker at 30s

18.3 Or Circuit Waveform

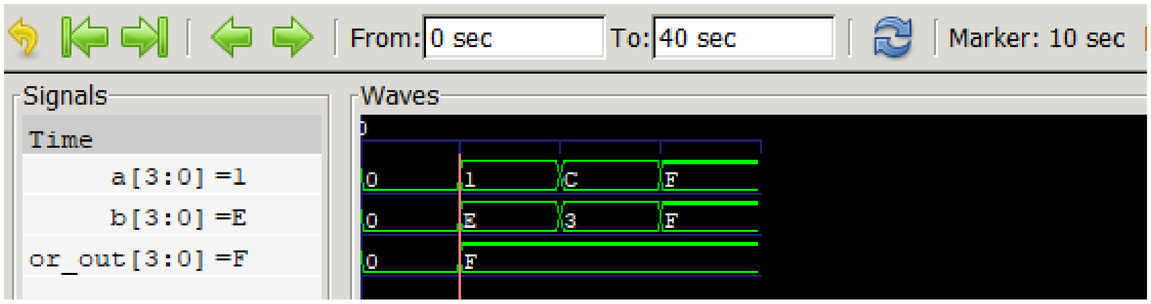
At 0 s, we can see that both A and B are 0, so the output is 0



```
a[3:0] =0
b[3:0] =0
or_out[3:0] =0
```

Figure 16: Or Circuit with marker at 0s

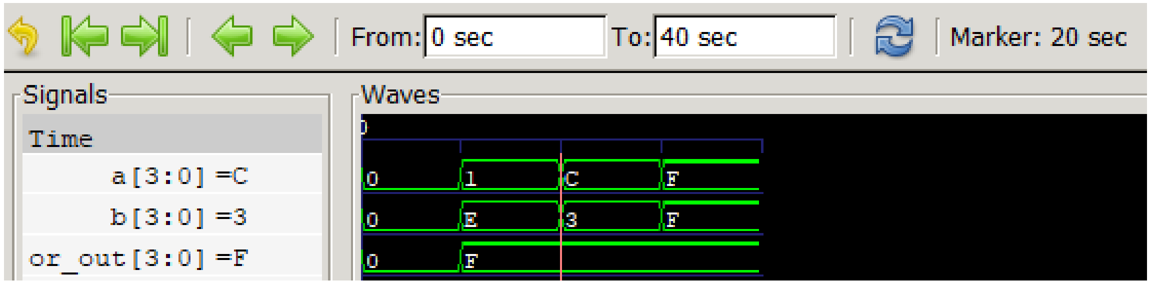
At 10 s, we can see that A is 1 and B is E, so the output is F



```
a[3:0] =1
b[3:0] =E
or_out[3:0] =F
```

Figure 17: Or Circuit with marker at 10s

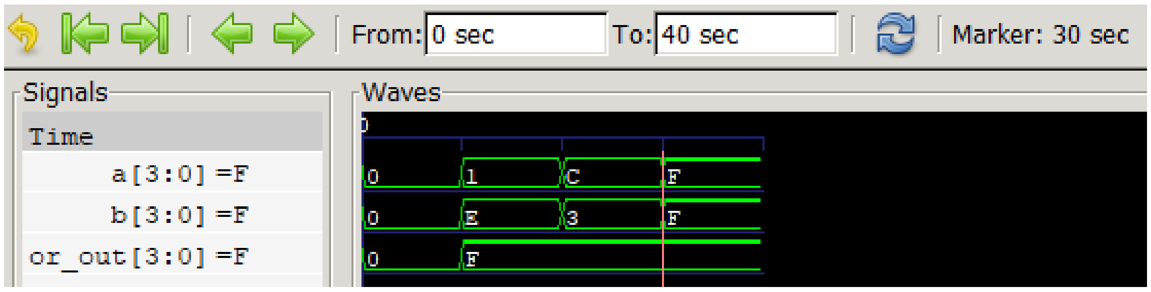
At 20 s, A is C and B is 3, so the output is F



```
a[3:0] =C
b[3:0] =3
or_out[3:0] =F
```

Figure 18: Or Circuit with marker at 20s

At 30 s, A is F and B is F, so the output is F

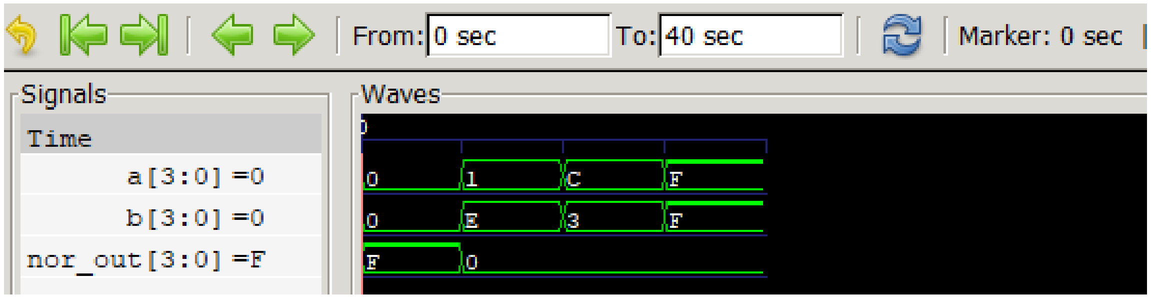


```
Time
      a[3:0] =F
      b[3:0] =F
or_out[3:0] =F
```

Figure 19: Or Circuit with marker at 30s

18.4 Nor Circuit Waveform

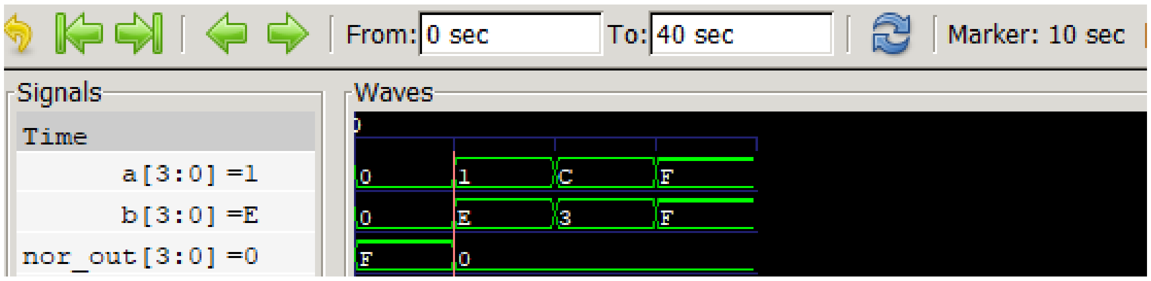
At 0 s, we can see that both A and B are 0, so the output is F



```
Time
       a[3:0] =0
       b[3:0] =0
nor_out[3:0] =F
```

Figure 20: Nor Circuit with marker at 0s

At 10 s, we can see that A is 1 and B is E, so the output is 0



```
Time
       a[3:0] =1
       b[3:0] =E
nor_out[3:0] =0
```

Figure 21: Nor Circuit with marker at 10s

At 20 s, A is C and B is 3, so the output is 0

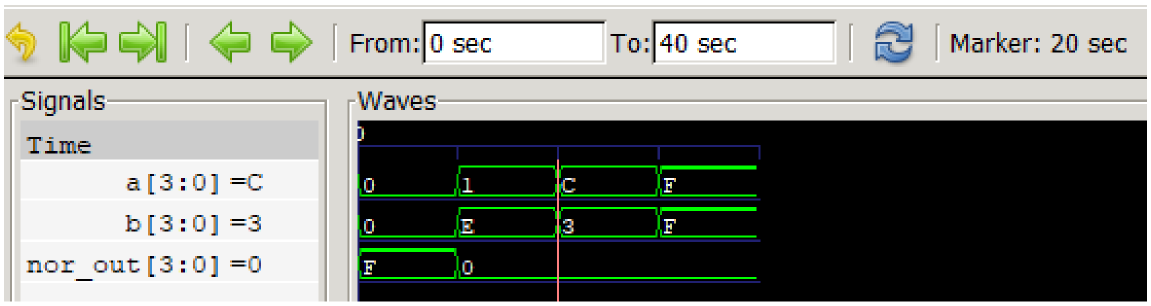


Figure 22: Nor Circuit with marker at 20s

At 30 s, A is F and B is F, so X is 0

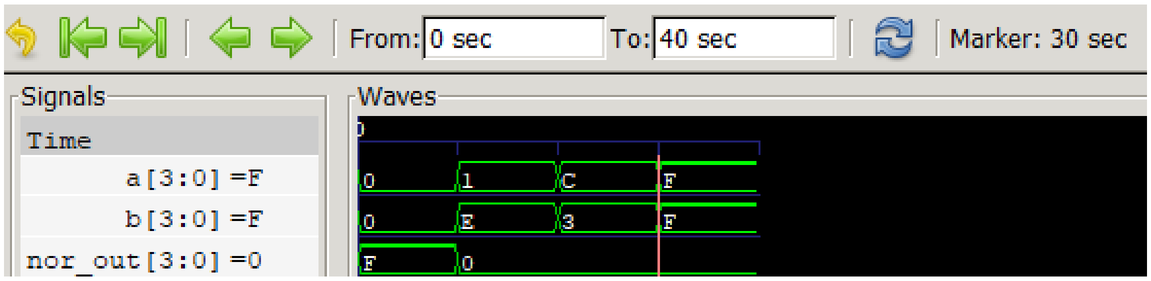


Figure 23: Nor Circuit with marker at 30s

18.5 Xor Circuit Waveform

At 0 s, we can see that both A and B are 0, so the output is 0

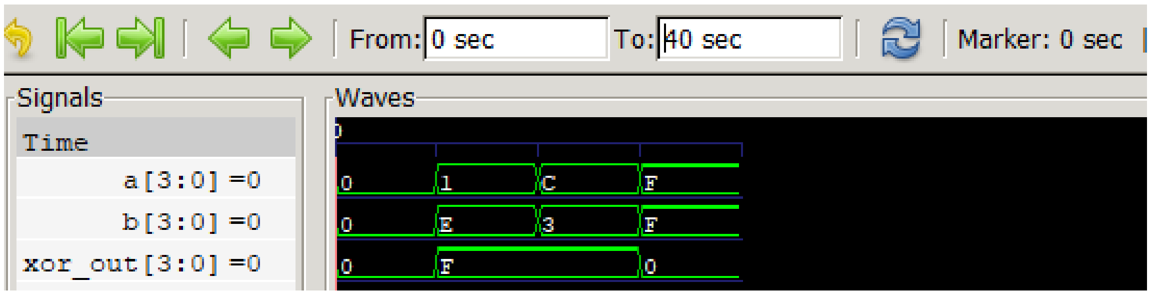


Figure 24: Xor Circuit with marker at 0s

At 10 s, we can see that A is 1 and B is E, so the output is F

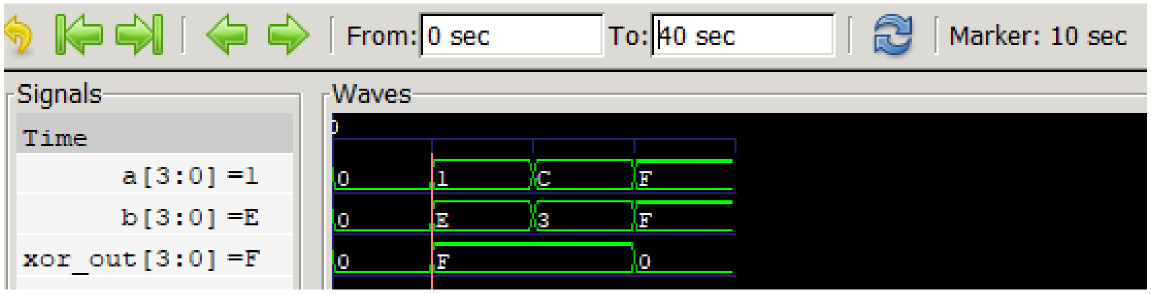


Figure 25: Xor Circuit with marker at 10s

At 20 s, A is C and B is 3, so the output is F

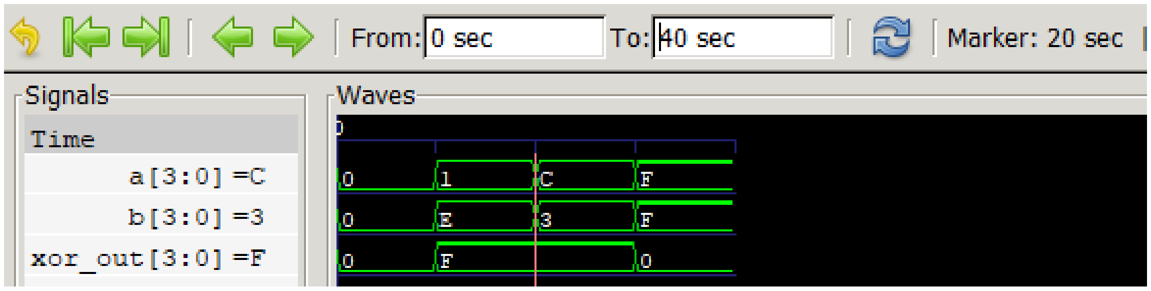


Figure 26: Xor Circuit with marker at 20s

At 30 s, A is F and B is F, so the output is 0

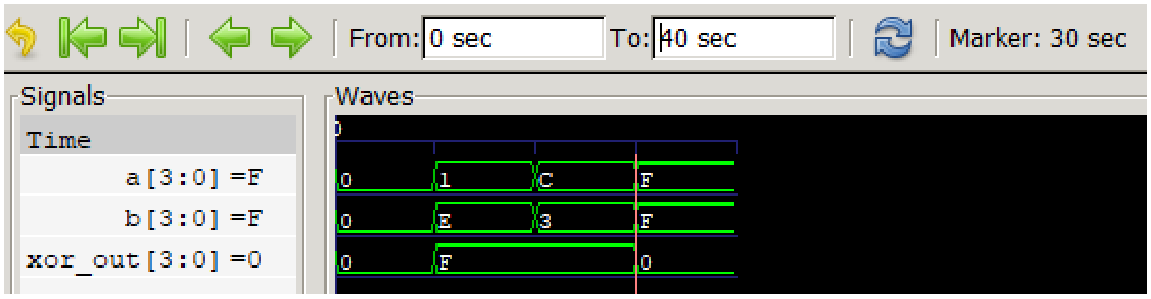


Figure 27: Xor Circuit with marker at 30s

18.6 Xnor Circuit Waveform

At 0 s, we can see that both A and B are 0, so the output is F

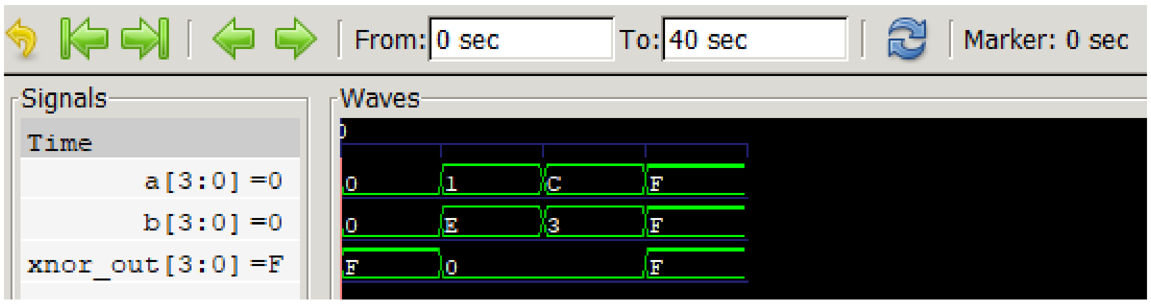


Figure 28: Xnor Circuit with marker at 0s

At 10 s, we can see that A is 1 and B is E, so the output is 0

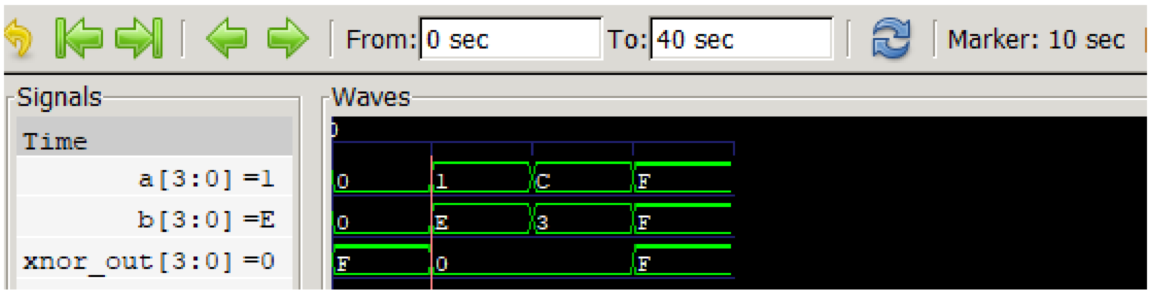


Figure 29: Xnor Circuit with marker at 10s

At 20 s, A is C and B is 3, so the output is 0

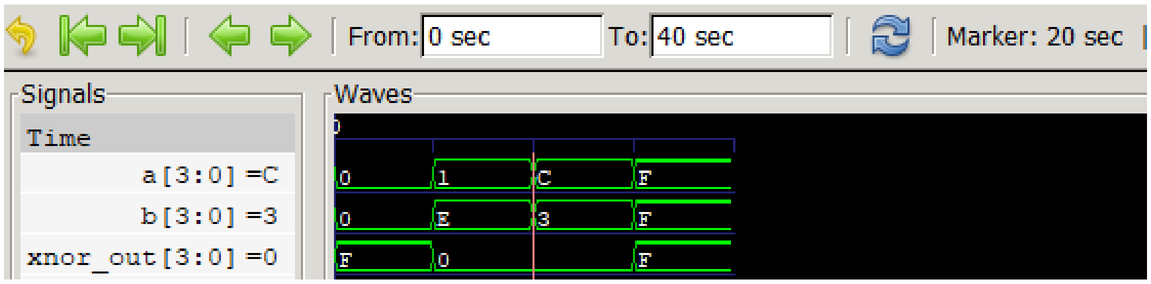
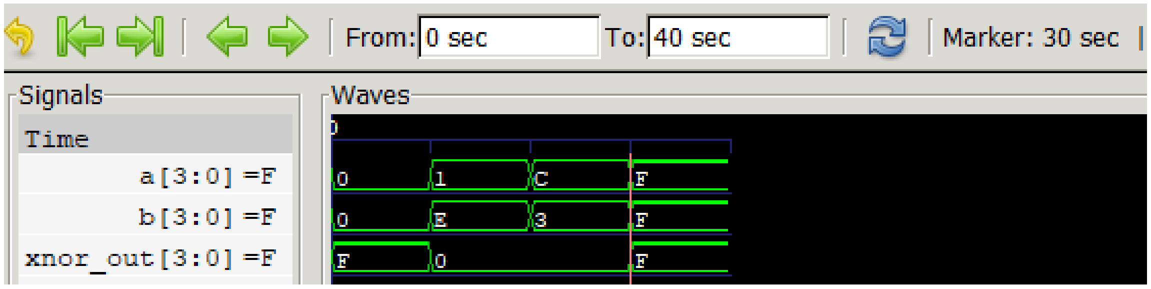


Figure 30: Xnor Circuit with marker at 20s

At 30 s, A is F and B is F, so the output is F



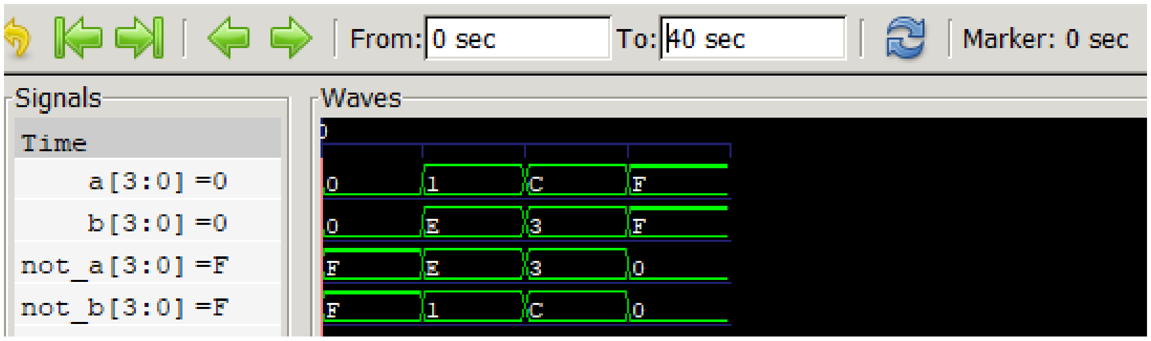
a[3:0] =F
b[3:0] =F
xnor_out[3:0] =F

```
0    1    C    F
0    E    3    F
F    0         F
```

Figure 31: Xnor Circuit with marker at 30s

18.7 Not Circuit Waveform

At 0 s, we can see that both A and B are 0, so the not A is F and the not B is F

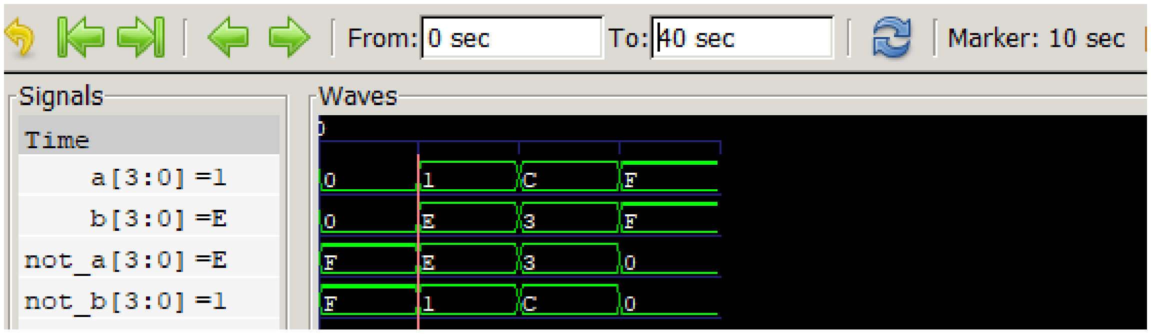


a[3:0] =0
b[3:0] =0
not_a[3:0] =F
not_b[3:0] =F

```
0    1    C    F
0    E    3    F
F    E    3    0
F    1    C    0
```

Figure 32: Not Circuit with marker at 0s

At 10 s, A is 1 and B is E, so the not A is E and the not B is 1

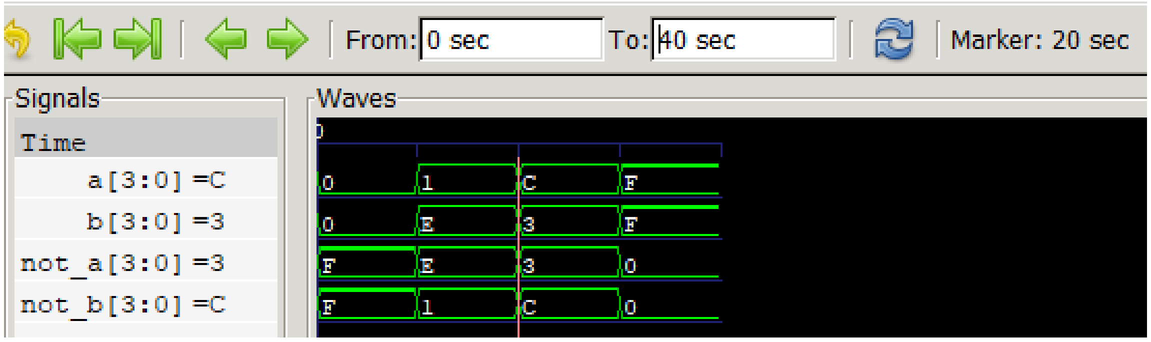


```
Time
    a[3:0] =1
    b[3:0] =E
not_a[3:0] =E
not_b[3:0] =1
```

```
0  1  C  F
0  E  3  F
F  E  3  0
F  1  C  0
```

Figure 33: Not Circuit with marker at 10s

At 20 s, A is C and B is 3, so the not A is 3 and the not B is C

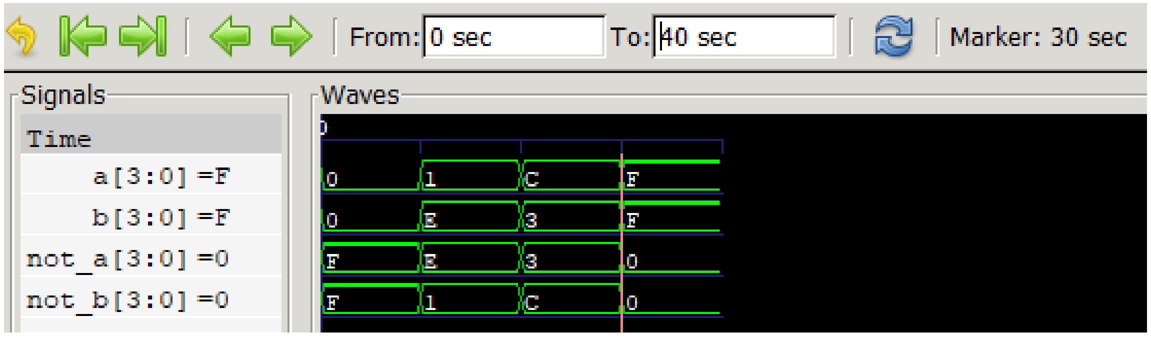


```
Time
    a[3:0] =C
    b[3:0] =3
not_a[3:0] =3
not_b[3:0] =C
```

```
0  1  C  F
0  E  3  F
F  E  3  0
F  1  C  0
```

Figure 34: Not Circuit with marker at 20s

At 30 s, both A and B are F, so the not A is 0 and the not B is 0



```
Time
    a[3:0] =F
    b[3:0] =F
not_a[3:0] =0
not_b[3:0] =0
```

```
0  1  C  F
0  E  3  F
F  E  3  0
F  1  C  0
```

Figure 35: Not Circuit with marker at 30s

18.8 2x4-bit input Circuit Waveform

At 0 s, we can see that four in1 is0X and four in2 is 0, so the output for four out1 is X and four out2 is X

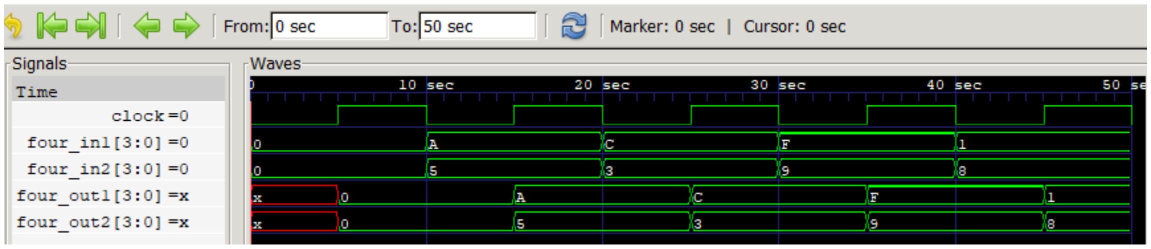


Figure 36: 2x4-bit input Circuit with marker at 0s

At 10 s, we can see that four in1 is A and four in2 is 5, so the output for four out1 is 0 and four out2 is 0

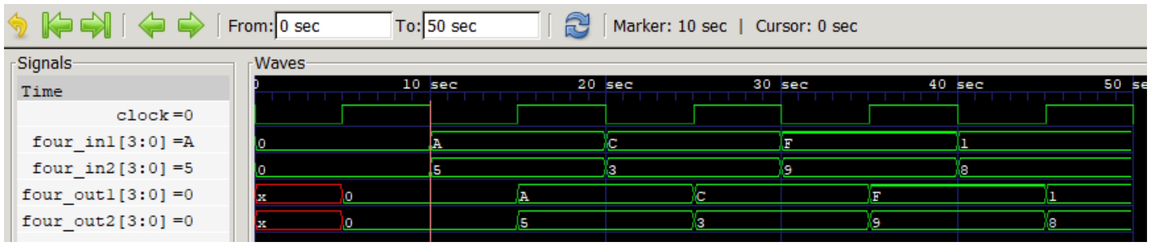


Figure 37: 2x4-bit input Circuit with marker at 10s

At 20 s, we can see that four in1 is C and four in2 is 3, so the output for four out1 is A and four out2 is 5

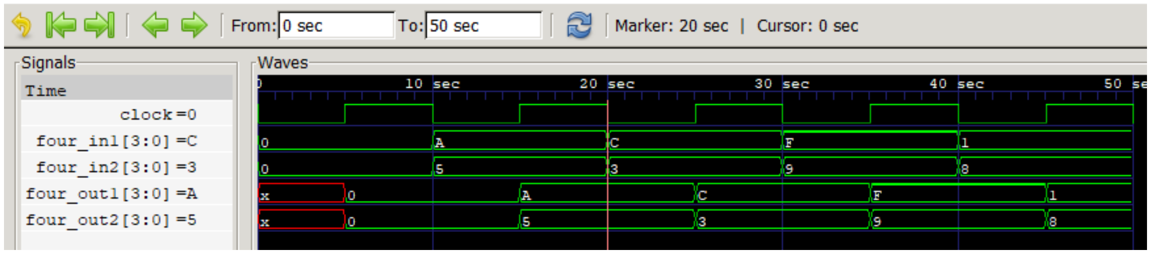
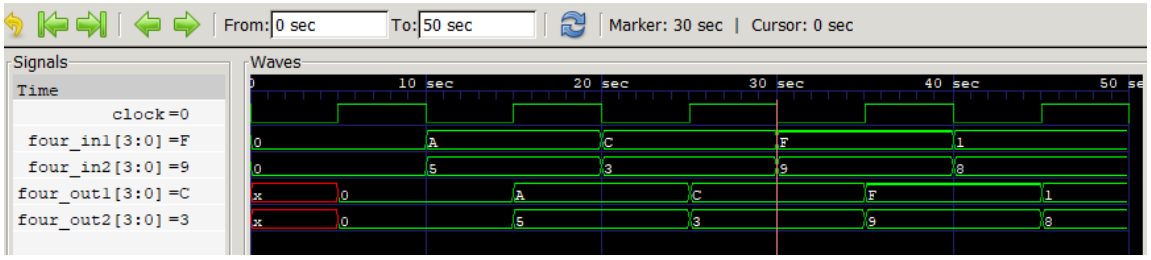


Figure 38: 2x4-bit input Circuit with marker at 20s

At 30 s, we can see that four in1 is F and four in2 is 9, so the output for four out1 is C and four out2 is 3

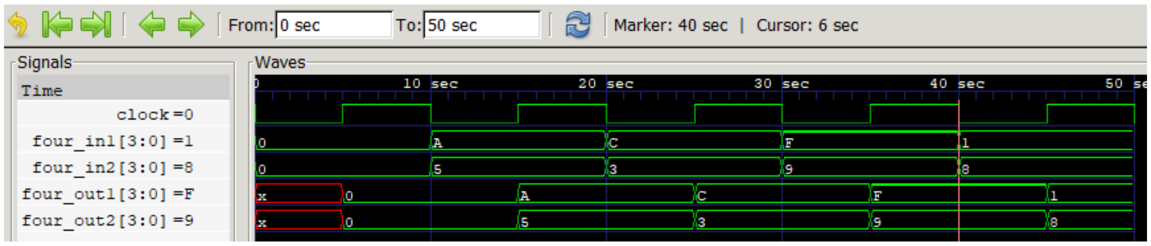

From: 0 sec To: 50 sec | Marker: 30 sec | Cursor: 0 sec

Signals

```
Time
        clock=0
  four_in1[3:0]=F
  four_in2[3:0]=9
four_out1[3:0]=C
four_out2[3:0]=3
```

Figure 39: 2x4-bit input Circuit with marker at 30s

At 40 s, we can see that four in1 is 1 and four in2 is 8, so the output for four out1 is F and four out2 is 9

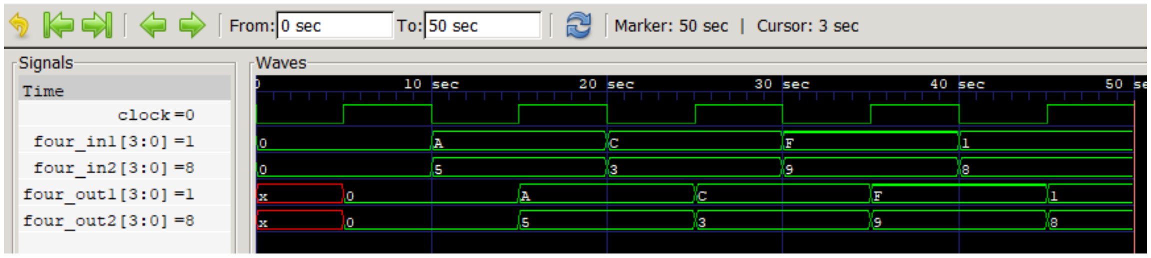

From: 0 sec To: 50 sec | Marker: 40 sec | Cursor: 6 sec

Signals

```
Time
        clock=0
  four_in1[3:0]=1
  four_in2[3:0]=8
four_out1[3:0]=F
four_out2[3:0]=9
```

Figure 40: 2x4-bit input Circuit with marker at 40s

At 50 s, we can see that four in1 is 1 and four in2 is 8, so the output for four out1 is 1 and four out2 is 8


From: 0 sec To: 50 sec | Marker: 50 sec | Cursor: 3 sec

Signals

```
Time
        clock=0
  four_in1[3:0]=1
  four_in2[3:0]=8
four_out1[3:0]=1
four_out2[3:0]=8
```

Figure 41: 2x4-bit input Circuit with marker at 50s

18.9 2x4-bit output Shift Circuit Waveform

At 0 s, we can see that data in1 is 0 and data in2 is 0, so that means that data out1 is 0 and data out2 is 0

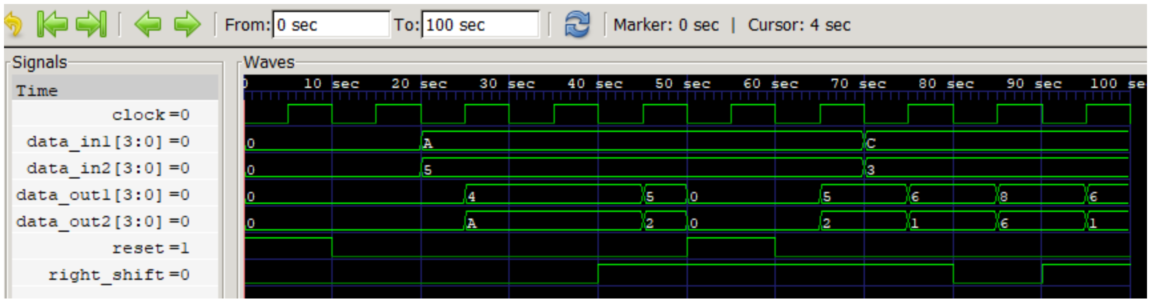


Figure 42: 2x4-bit output Shift Circuit with marker at 0s

At 10 s,we can see that data in1 is 0 and data in2 is 0, so that means that data out1 is 0 and data out2 is 0

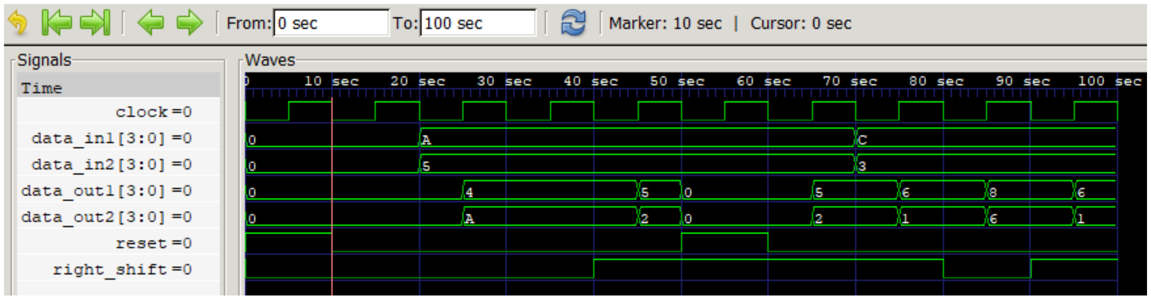


Figure 43: 2x4-bit output Shift Circuit with marker at 10s

At 20 s, we can see that data in1 is A and data in2 is 5, so that means that data out1 is 0 and data out2 is 0

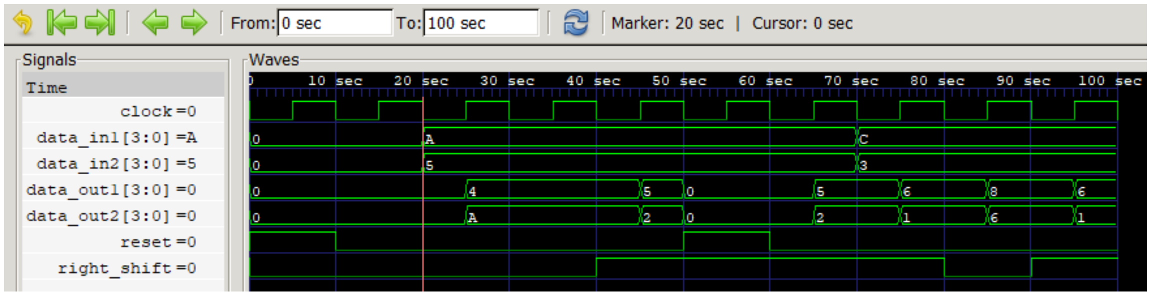


Figure 44: 2x4-bit output Shift Circuit with marker at 20s

At 30 s, we can see that data in1 is A and data in2 is 5, so that means that data out1 is 4 and data out2 is A

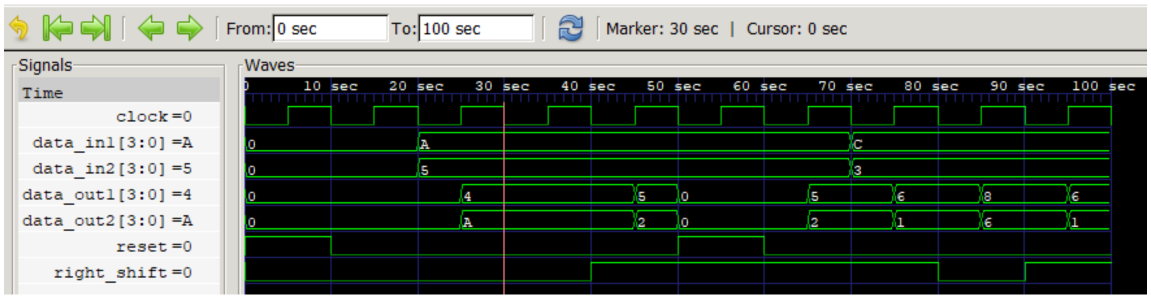


Figure 45: 2x4-bit output Shift Circuit with marker at 30s

At 40 s, we can see that data in1 is A and data in2 is 5, so that means that data out1 is 4 and data out2 is A

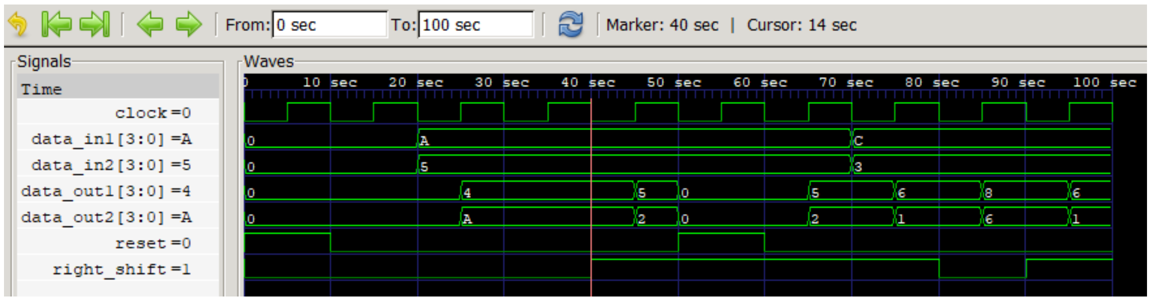


Figure 46: 2x4-bit output Shift Circuit with marker at 40s

At 50 s, we can see that data in1 is A and data in2 is 5, so that means that data out1 is 0 and data out2 is 0

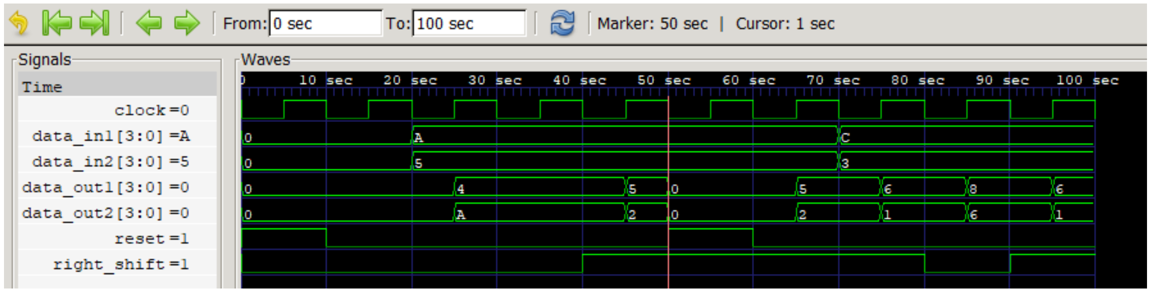


Figure 47: 2x4-bit output Shift Circuit with marker at 50s

At 60 s, we can see that data in1 is A and data in2 is 5, so that means that data out1 is 0 and data out2 is 0

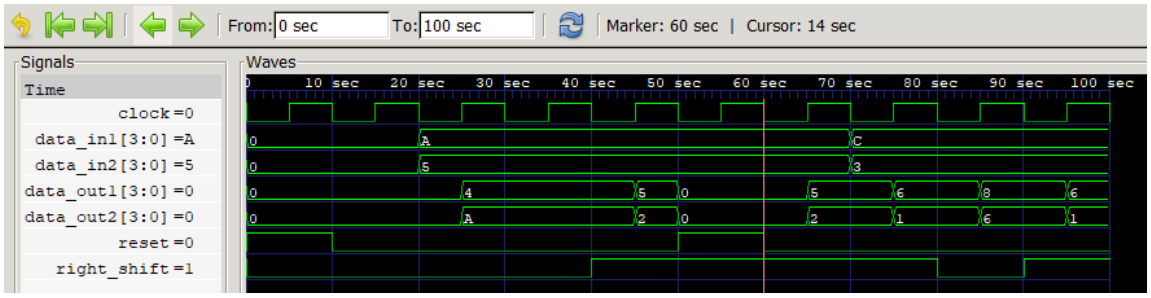


Figure 48: 2x4-bit output Shift Circuit with marker at 60s

At 70 s, we can see that data in1 is C and data in2 is 3, so that means that data out1 is 5 and data out2 is 2

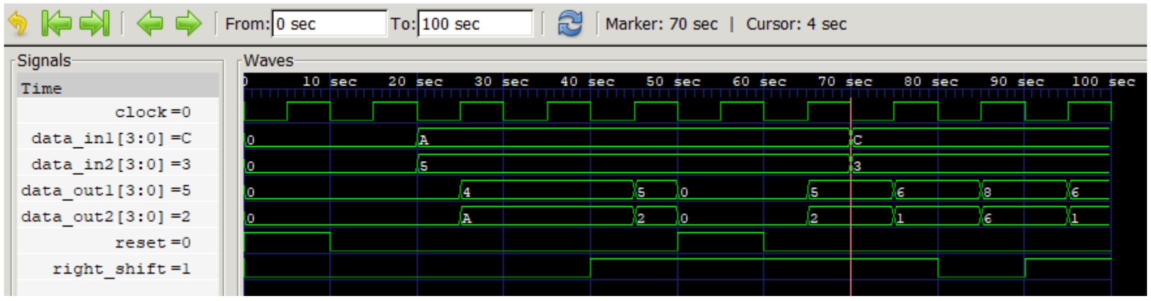


Figure 49: 2x4-bit output Shift Circuit with marker at 70s

At 80 s, we can see that data in1 is C and data in2 is 3, so that means that data out1 is 6 and data out2 is 1

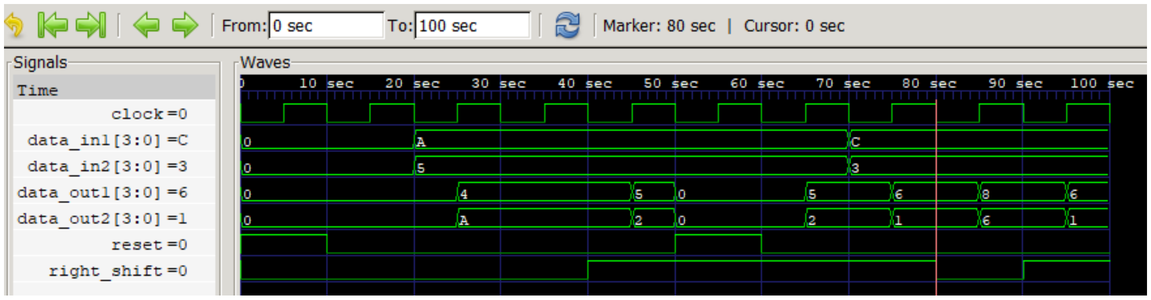


Figure 50: 2x4-bit output Shift Circuit with marker at 80s

At 90 s, we can see that data in1 is C and data in2 is 3, so that means that data out1 is 8 and data out2 is 6

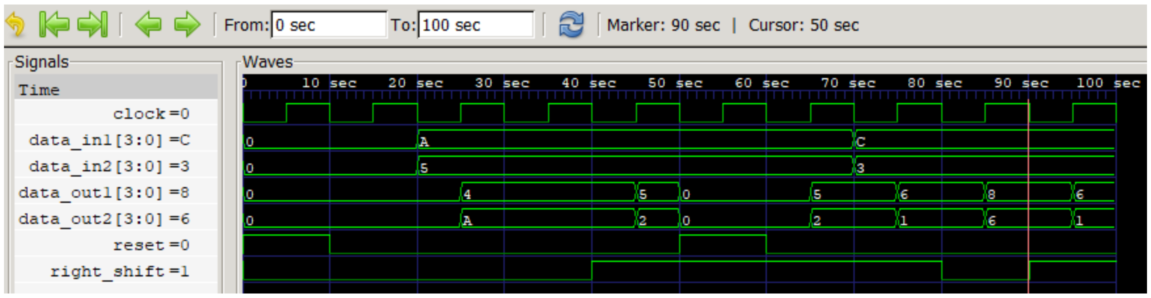


Figure 51: 2x4-bit output Shift Circuit with marker at 90s

At 100 s, we can see that data in1 is C and data in2 is 3, so that means that data out1 is 6 and data out2 is 1

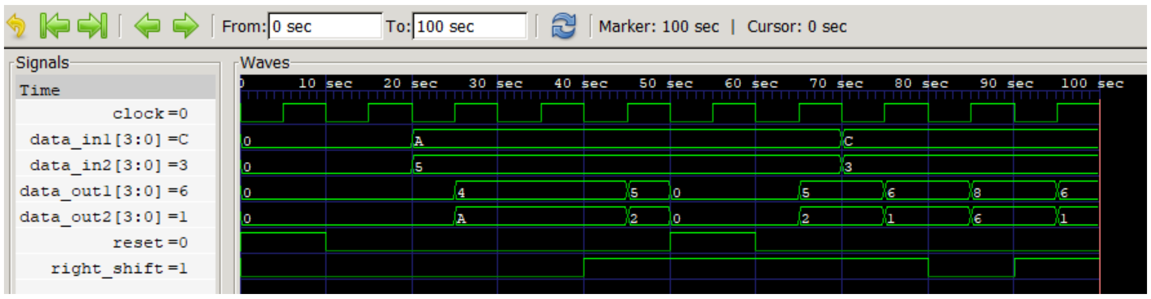
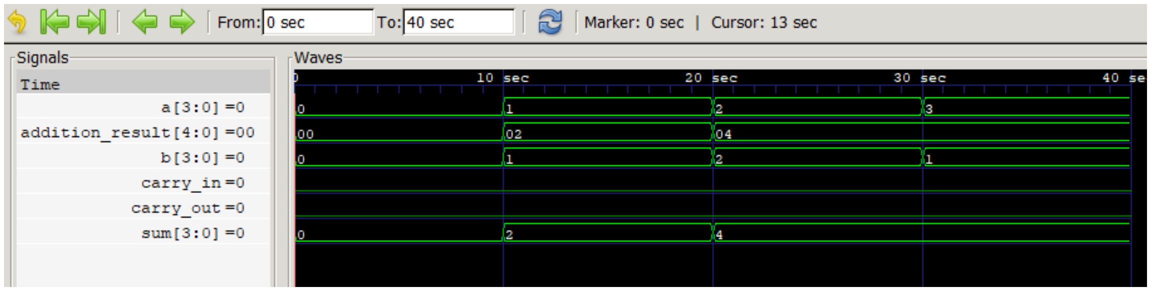


Figure 52: 2x4-bit output Shift Circuit with marker at 100s

18.10 Addition

At 0 s, we can see that a is 0 and b is 0, so that means addition result is 00 and sum is 0

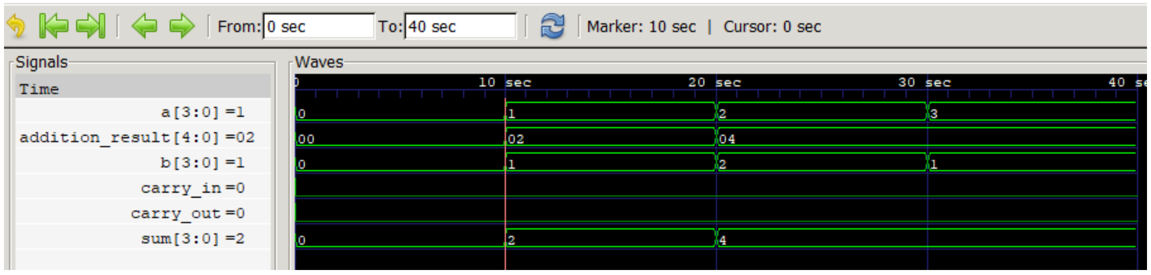


From: 0 sec To: 40 sec Marker: 0 sec | Cursor: 13 sec

```
Time
a[3:0] =0
addition_result[4:0] =00
b[3:0] =0
carry_in =0
carry_out =0
sum[3:0] =0
```

Figure 53: Addition Circuit with marker at 0s

At 10 s, we can see that a is 1 and b is 1, so that means addition result is 02 and sum is 2

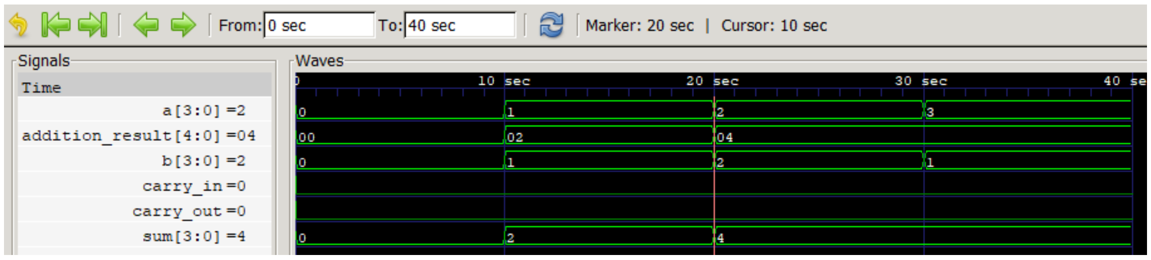


From: 0 sec To: 40 sec Marker: 10 sec | Cursor: 0 sec

```
Time
a[3:0] =1
addition_result[4:0] =02
b[3:0] =1
carry_in =0
carry_out =0
sum[3:0] =2
```

Figure 54: Addition Circuit with marker at 10s

At 20 s, we can see that a is 2 and b is 2, so that means addition result is 04 and sum is 4



From: 0 sec To: 40 sec Marker: 20 sec | Cursor: 10 sec

```
Time
a[3:0] =2
addition_result[4:0] =04
b[3:0] =2
carry_in =0
carry_out =0
sum[3:0] =4
```

Figure 55: Addition Circuit with marker at 20s

At 30 s, we can see that a is 3 and b is 1, so that means addition result is 04 and sum is 4

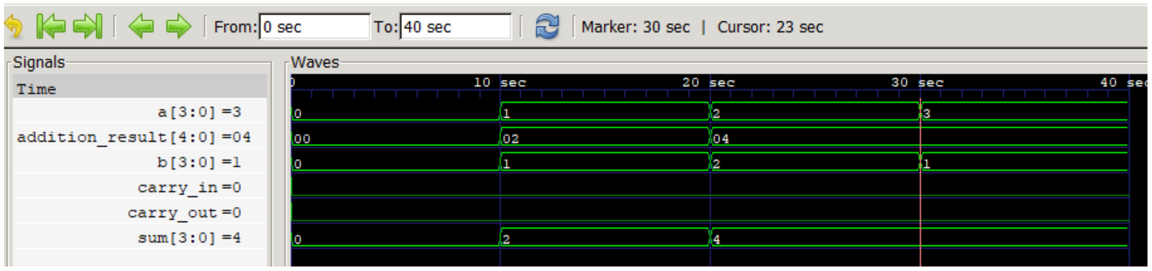


Figure 56: Addition Circuit with marker at 30s

18.11 Subtraction

At 0 s, we can see that A is 4 and B is 2, so then the difference is 2

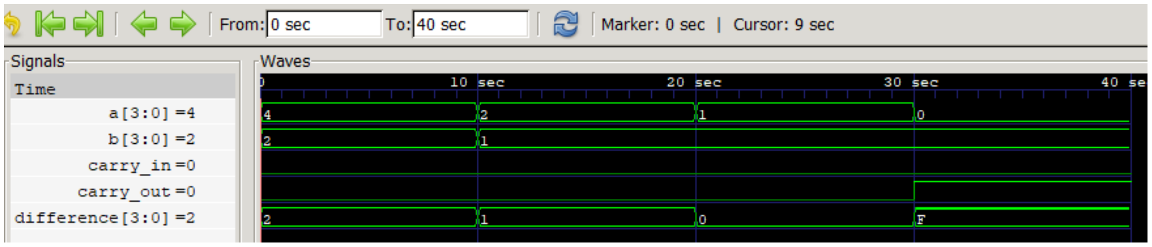


Figure 57: Subtraction Circuit with marker at 0s

At 10 s, we can see that A is 2 and B is 1, so then the difference is 1

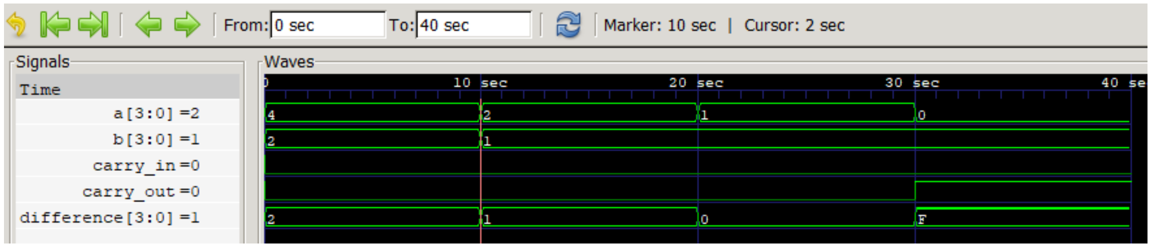
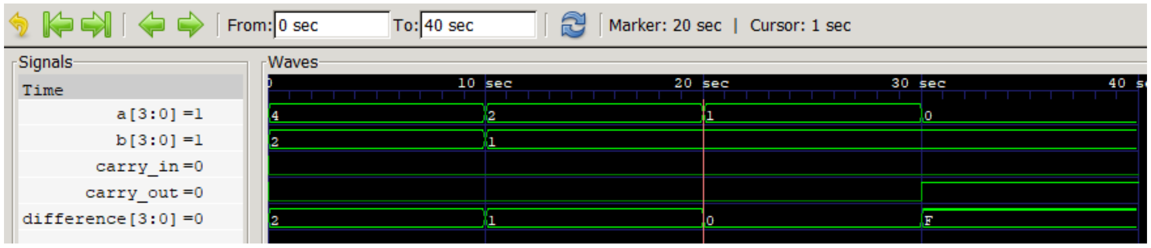


Figure 58: Subtraction Circuit with marker at 10s

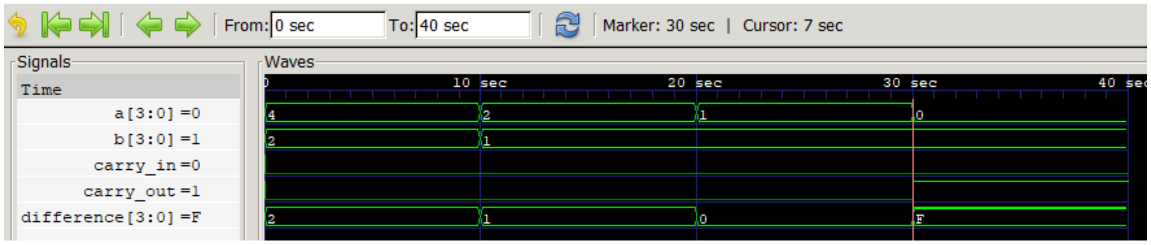
At 20 s, we can see that A is 1 and B is 1, so then the difference is 0



```
Time
a[3:0] =1
b[3:0] =1
carry_in =0
carry_out =0
difference[3:0] =0
```

Figure 59: Subtraction Circuit with marker at 20s

At 30 s, we can see that A is 0 and B is 1, so then the difference is F

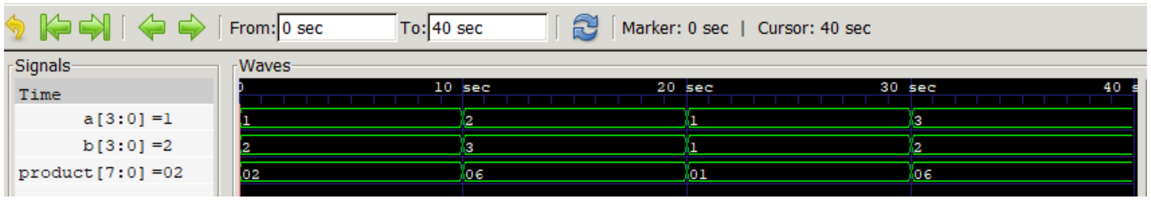


```
Time
a[3:0] =0
b[3:0] =1
carry_in =0
carry_out =1
difference[3:0] =F
```

Figure 60: Subtraction Circuit with marker at 30s

18.12 Multiplication

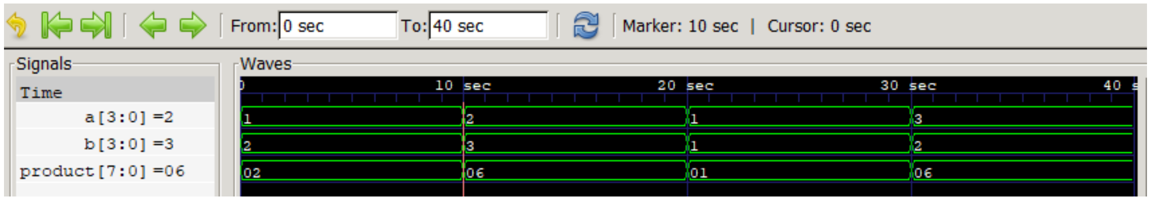
At 0 s, we can see that A is 1 and B is 2, so then the product is 02



```
Time
a[3:0] =1
b[3:0] =2
product[7:0] =02
```

Figure 61: Multiplication Circuit with marker at 0s

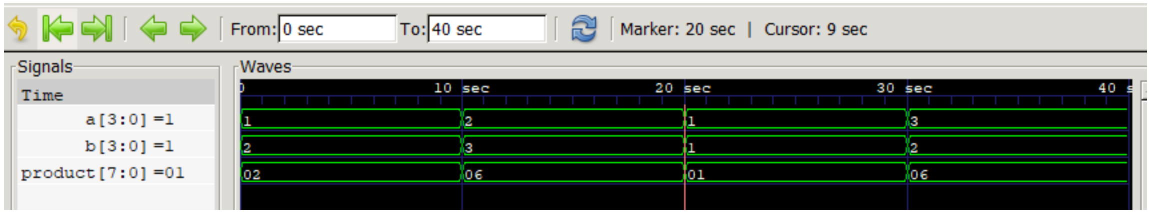
At 10 s, we can see that A is 2 and B is 3, so then the product is 06


From: 0 sec To: 40 sec | Marker: 10 sec | Cursor: 0 sec

```
Time
     a[3:0] =2
     b[3:0] =3
product[7:0] =06
```

Figure 62: Multiplication Circuit with marker at 10s

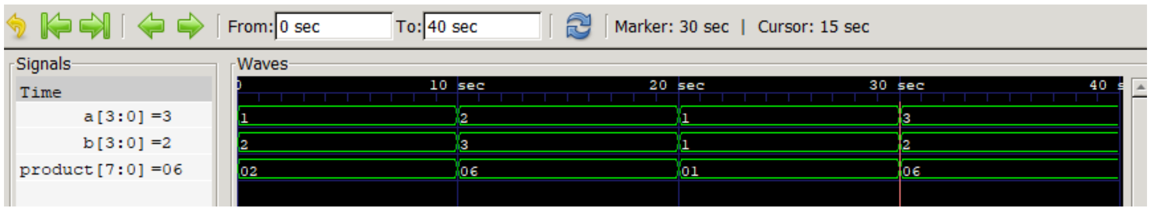
At 20 s, we can see that A is 1 and B is 1, so then the product is 01


From: 0 sec To: 40 sec | Marker: 20 sec | Cursor: 9 sec

```
Time
     a[3:0] =1
     b[3:0] =1
product[7:0] =01
```

Figure 63: Multiplication Circuit with marker at 20s

At 30 s, we can see that A is 3 and B is 2, so then the product is 06

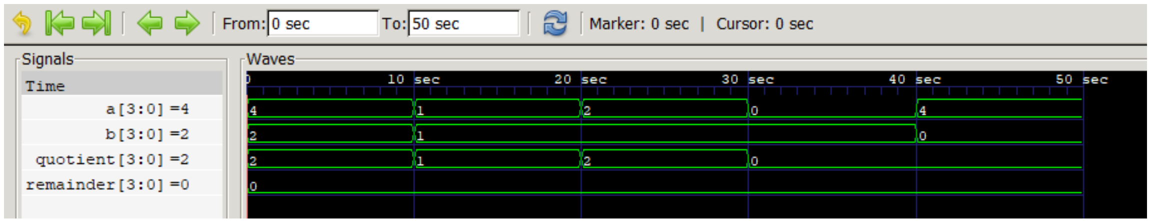

From: 0 sec To: 40 sec | Marker: 30 sec | Cursor: 15 sec

```
Time
     a[3:0] =3
     b[3:0] =2
product[7:0] =06
```

Figure 64: Multiplication Circuit with marker at 30s

18.13 Division

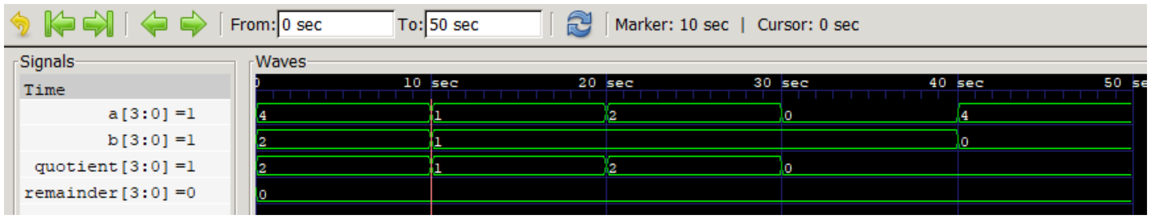
At 0 s, we can see that A is 4 and B is 2, so that means that the quotient is 2 and the remainder is 0


From: 0 sec To: 50 sec Marker: 0 sec | Cursor: 0 sec

```
Signals
Time
        a[3:0] =4
        b[3:0] =2
 quotient[3:0] =2
remainder[3:0] =0
```

Figure 65: Division Circuit with marker at 0s

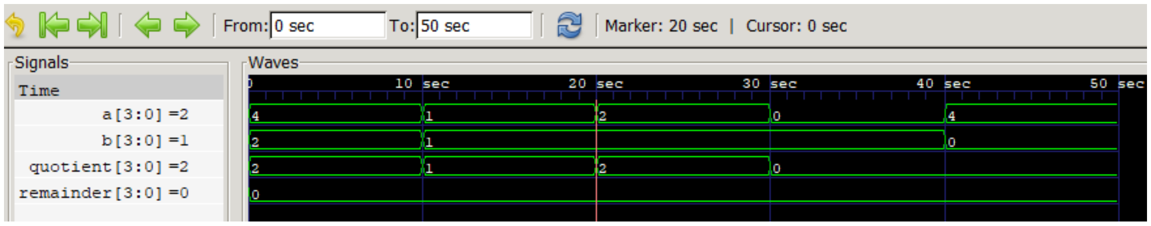
At 10 s, we can see that A is 1 and B is 1, so that means that the quotient is 1 and the remainder is 0


From: 0 sec To: 50 sec Marker: 10 sec | Cursor: 0 sec

```
Signals
Time
        a[3:0] =1
        b[3:0] =1
 quotient[3:0] =1
remainder[3:0] =0
```

Figure 66: Division Circuit with marker at 10s

At 20 s, we can see that A is 2 and B is 1, so that means that the quotient is 2 and the remainder is 0


From: 0 sec To: 50 sec Marker: 20 sec | Cursor: 0 sec

```
Signals
Time
        a[3:0] =2
        b[3:0] =1
 quotient[3:0] =2
remainder[3:0] =0
```

Figure 67: Division Circuit with marker at 20s

At 30 s, we can see that A is 0 and B is 1, so that means that the quotient is 0 and the remainder is 0

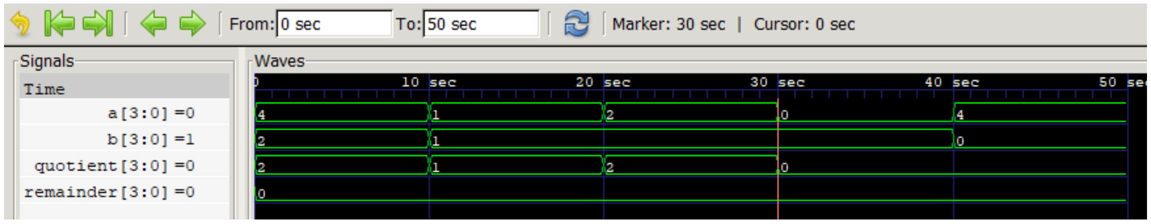


Figure 68: Division Circuit with marker at 30s

At 40 s, we can see that A is 4 and B is 0, so that means that the quotient is 0 and the remainder is 0

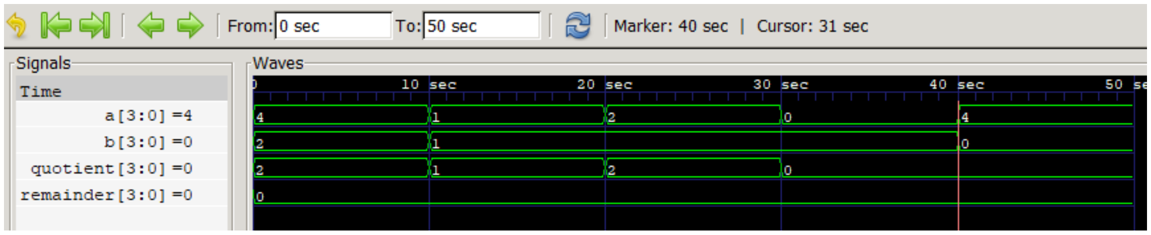
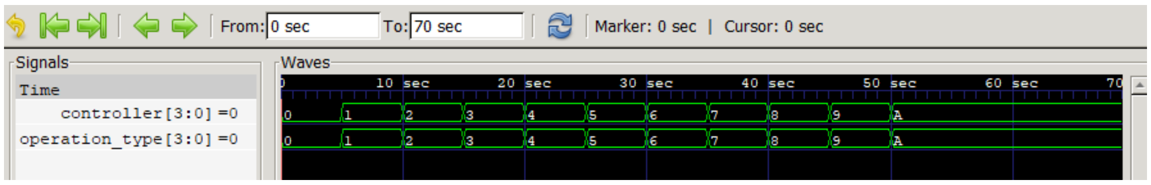


Figure 69: Division Circuit with marker at 40s

18.14 Control Circuit

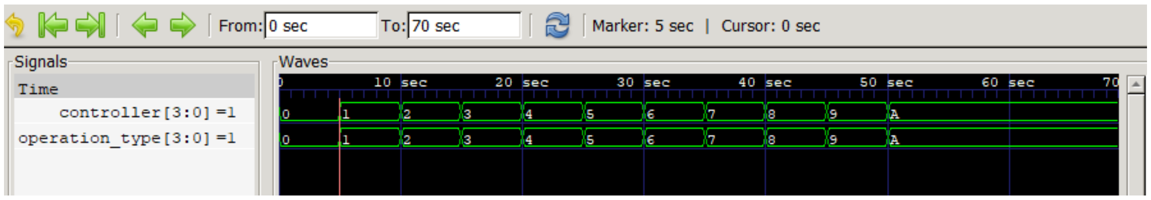
At 0 s, we can see that controller is 0 and operation type is 0



```
controller[3:0] =0
operation_type[3:0] =0
```

Figure 70: Control Circuit with marker at 0 s

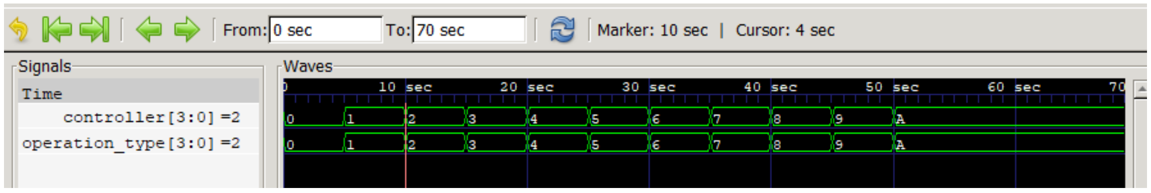
At 5 s, we can see that controller is 1 and operation type is 1



```
controller[3:0] =1
operation_type[3:0] =1
```

Figure 71: Control Circuit with marker at 5 s

At 10 s, we can see that controller is 2 and operation type is 2



```
controller[3:0] =2
operation_type[3:0] =2
```

Figure 72: Control Circuit with marker at 10 s

At 15 s, we can see that controller is 3 and operation type is 3

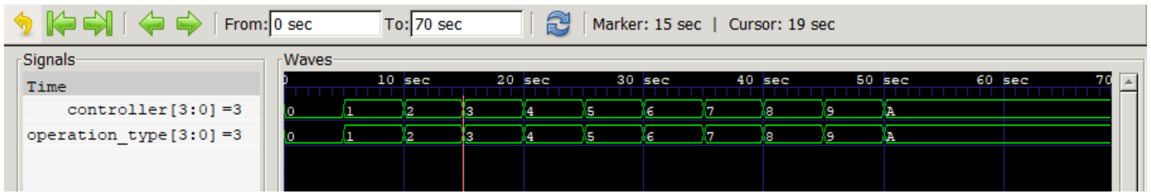


Figure 73: Control Circuit with marker at 15 s

At 20 s, we can see that controller is 4 and operation type is 4

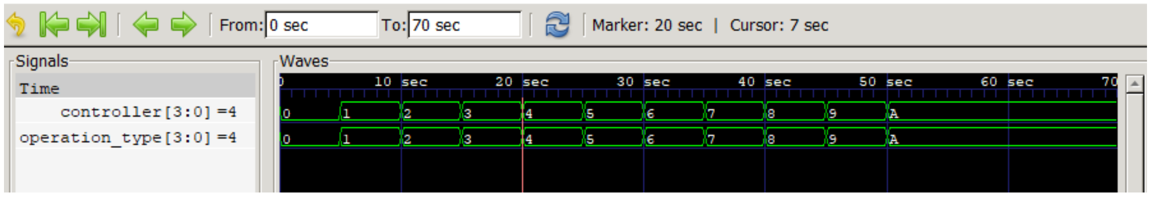


Figure 74: Control Circuit with marker at 20 s

At 25 s, we can see that controller is 5 and operation type is 5

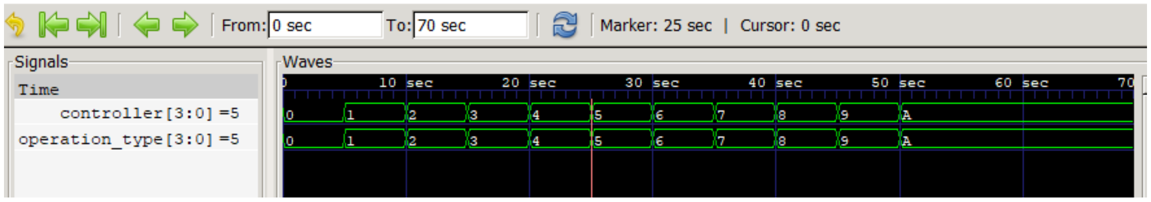
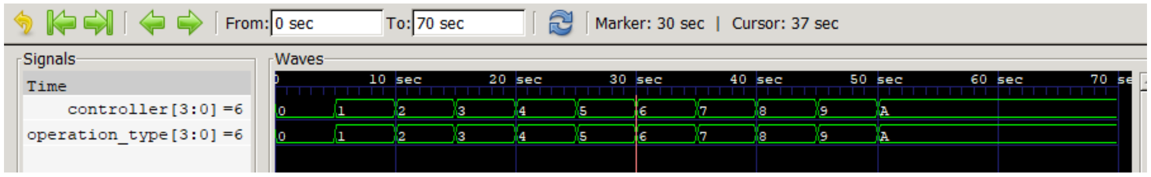


Figure 75: Control Circuit with marker at 25 s

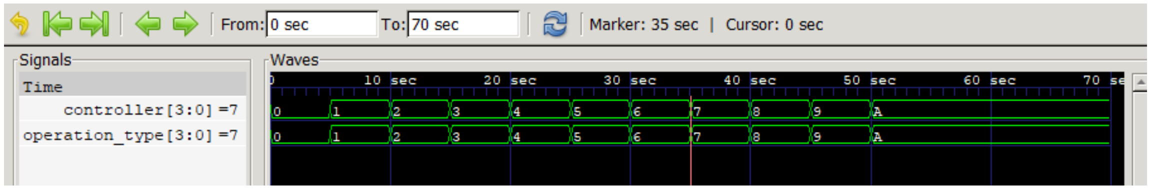
At 30 s, we can see that controller is 6 and operation type is 6


From: 0 sec To: 70 sec Marker: 30 sec | Cursor: 37 sec

controller[3:0] =6
operation_type[3:0] =6

Figure 76: Control Circuit with marker at 30 s

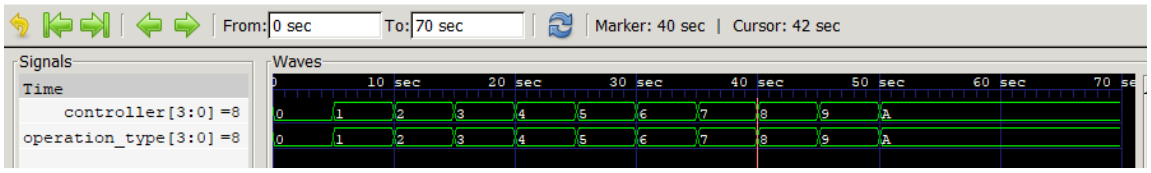
At 35 s, we can see that controller is 7 and operation type is 7


From: 0 sec To: 70 sec Marker: 35 sec | Cursor: 0 sec

controller[3:0] =7
operation_type[3:0] =7

Figure 77: Control Circuit with marker at 35 s

At 40 s, we can see that controller is 8 and operation type is 8


From: 0 sec To: 70 sec Marker: 40 sec | Cursor: 42 sec

controller[3:0] =8
operation_type[3:0] =8

Figure 78: Control Circuit with marker at 40 s

At 45 s, we can see that controller is 9 and operation type is 9

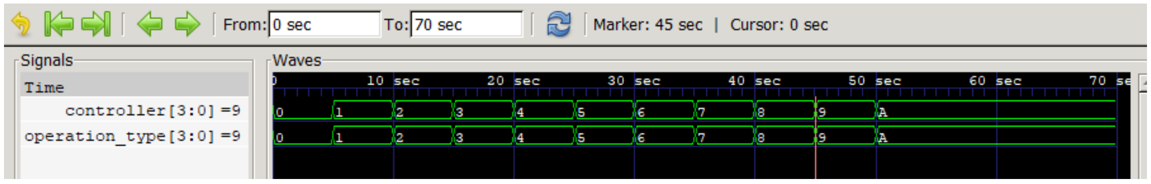


Figure 79: Control Circuit with marker at 45 s

At 50 s, we can see that controller is A and operation type is A

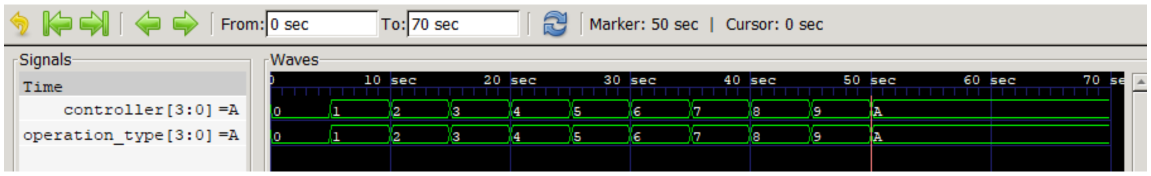


Figure 80: Control Circuit with marker at 50 s