PROJECT STEP 2

GROUP TOMATO

Ulises Espinoza-Gonzalez and Jack Stolpman

CS.3339 Computer Architecture

Texas State University

October 29, 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 first section of our project, we will be utilizing 1-bit Not, Nand, Nor, and 1x4-bit input / 1x4-bit output shift circuits 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 section, we are going to go over the circuits and then the Verilog code for each one with the test modules.

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.

```
1   `include "and_gate.v"
2
3   module and_gate_tb;
4       reg [3:0] a, b;
5       wire [3:0] and_out;
6
7       and_gate G1 (.a(a), .b(b), .and_out(and_out));
8
9       initial begin
10          $dumpfile("and_gate.vcd");
11          $dumpvars(0, and_gate_tb);
12
13          a = 4'b0000; b = 4'b0000; #10;
14          a = 4'b1111; b = 4'b0001; #10;
15          a = 4'b1010; b = 4'b1100; #10;
16          a = 4'b1111; b = 4'b1111; #10;
17
18          $finish;
19      end
```

```
20  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.

```
1  module and_gate
2  (
3      input [3:0] a, b,
4      output [3:0] and_out
5  );
6      assign and_out = a & b;
7  endmodule
```
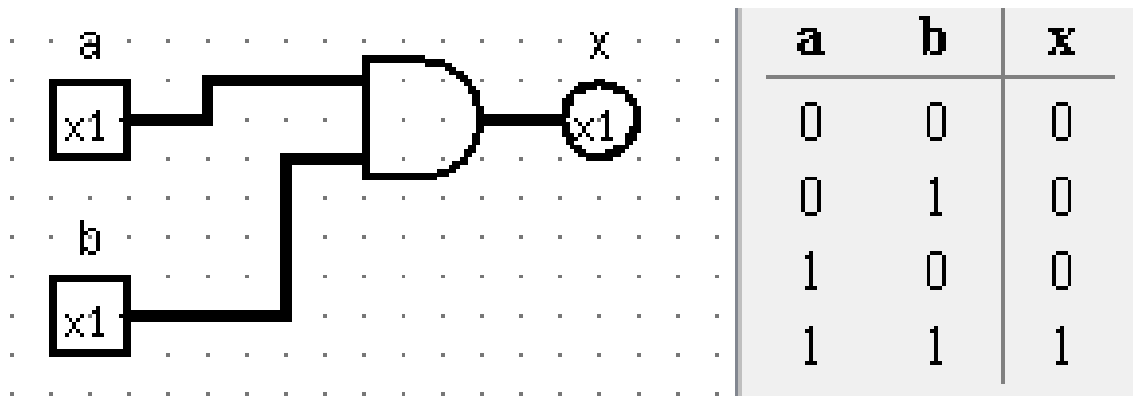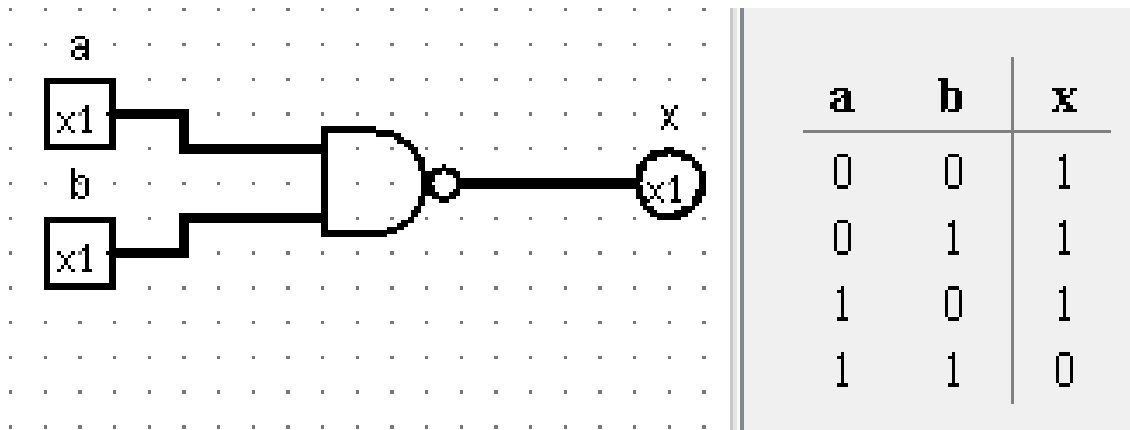

Figure 1: And truth table and gate

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.

```verilog
`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.

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

3

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.

```verilog
1  `include "or_gate.v"
2
3  module or_gate_tb;
4      reg [3:0] a, b;
5      wire [3:0] or_out;
6
7      or_gate G1 (.a(a), .b(b), .or_out(or_out));
8
9      initial begin
10         $dumpfile("or_gate.vcd");
11         $dumpvars(0, or_gate_tb);
12
13         a = 4'b0000; b = 4'b0000; #10;
14         a = 4'b1111; b = 4'b0001; #10;
15         a = 4'b1010; b = 4'b1100; #10;
16         a = 4'b1111; b = 4'b1111; #10;
```

4

```
17
18          $finish;
19      end
20 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.

```
1 module or_gate
2 (
3      input [3:0] a, b,
4      output [3:0] or_out
5 );
6      assign or_out = a | b;
7 endmodule
```
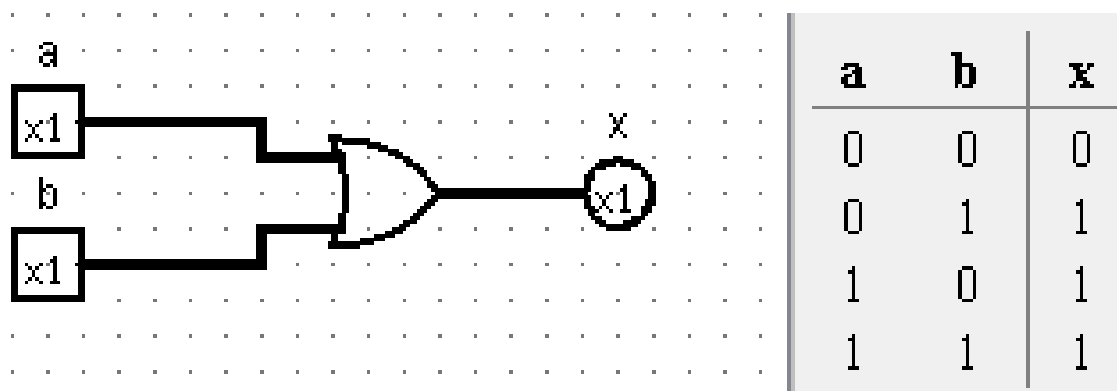

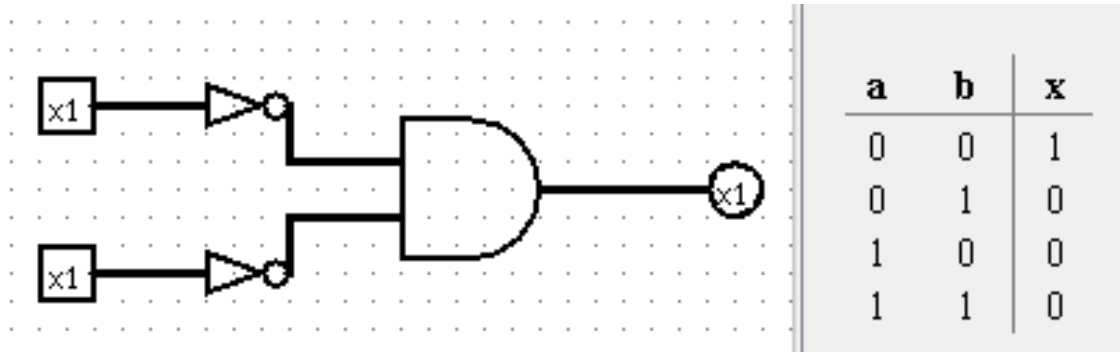| a	b	x
0	0	0
0	1	1
1	0	1
1	1	1

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.

```
1  `include "nor_gate.v"
2
3  module nor_gate_tb;
4      reg [3:0] a, b;
5      wire [3:0] nor_out;
6
7      nor_gate G1 (.a(a), .b(b), .nor_out(nor_out));
8
9      initial begin
10          $dumpfile("nor_gate.vcd");
11          $dumpvars(0, nor_gate_tb);
12
13          a = 4'b0000; b = 4'b0000; #10;
14          a = 4'b1111; b = 4'b0001; #10;
15          a = 4'b1010; b = 4'b1100; #10;
16          a = 4'b1111; b = 4'b1111; #10;
17
18          $finish;
19      end
20  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.

```
1  module nor_gate
2  (
3      input [3:0] a, b,
4      output [3:0] nor_out
5  );
6      assign nor_out = ~(a | b);
7  endmodule
```
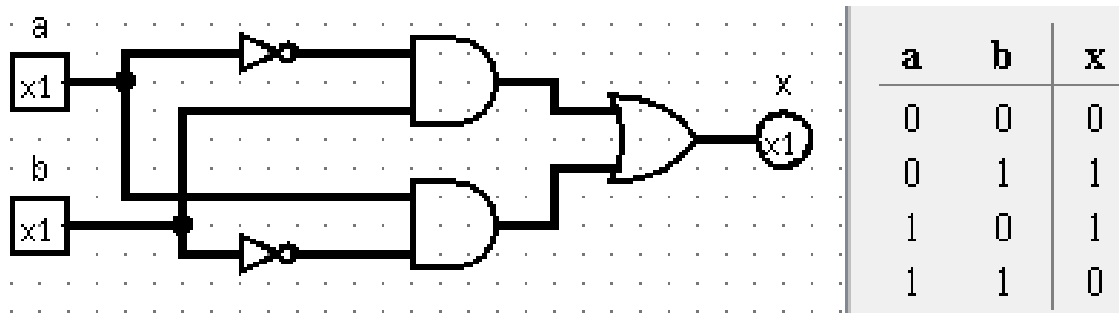
Figure 4: Nor truth table and gate

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.

```verilog
'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
```

```
20  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.

```
1  module xor_gate
2  (
3      input [3:0] a, b,
4      output [3:0] xor_out
5  );
6      assign xor_out = a ^ b;
7  endmodule
```


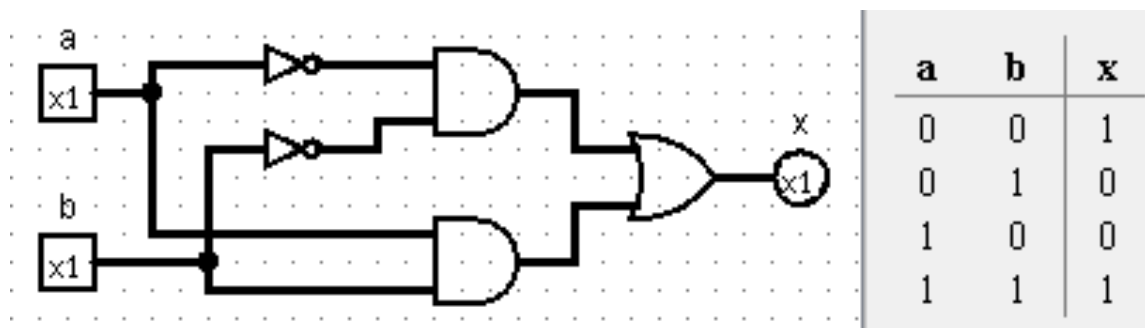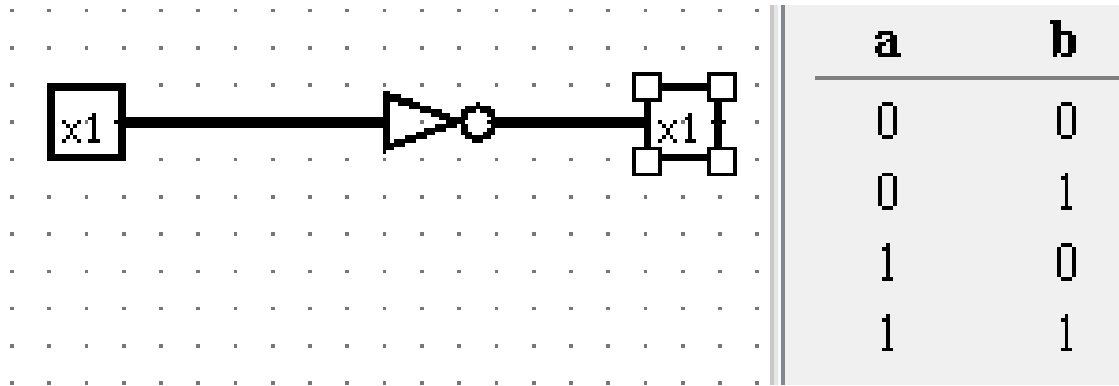| a	b	x
0	0	0
0	1	1
1	0	1
1	1	0

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.

```
1  `include "xnor_gate.v"
2
3  module xnor_gate_tb;
4      reg [3:0] a, b;
```

```
5     wire [3:0] xnor_out;

6

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

8

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

12

13        a = 4'b0000; b = 4'b0000; #10;
14        a = 4'b1111; b = 4'b0001; #10;
15        a = 4'b1010; b = 4'b1100; #10;
16        a = 4'b1111; b = 4'b1111; #10;

17

18        $finish;
19     end
20 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.

```
1 module xnor_gate
2 (
3     input [3:0] a, b,
4     output [3:0] xnor_out
5 );
6     assign xnor_out = ~(a ^ b);
7 endmodule
```


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

Figure 6: Xnor truth table and gate

9

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.

```verilog
`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.

```verilog
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.

```verilog
`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
```

```
20          four_in1 = 4'b1111; four_in2 = 4'b1001; #10; // Test case 3
21          four_in1 = 4'b0001; four_in2 = 4'b1000; #10; // Test case 4
22
23          $finish;
24      end
25
26 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.

```
1 module four_bit_input
2 (
3      input clock,
4      input [3:0] four_in1, four_in2,  // 2x4-bit input signals ([3:0])
5      output reg [3:0] four_out1, four_out2 // 2x4-bit output signals ([3:0])
6 );
7
8      always @(posedge clock) begin
9          four_out1 <= four_in1;
10         four_out2 <= four_in2;
11     end
12
13 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.

```
1 'include "four_bit_shift_circuit.v"
2
3 module four_bit_shift_circuit_tb;
```

```
4
5      reg clock, reset, right_shift;
6      reg [3:0] data_in1, data_in2;
7      wire [3:0] data_out1, data_out2;
8
9      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));
10
11     always begin
12         #5 clock = ~clock; // Clock generation
13     end
14
15     initial begin
16         clock = 0;
17         reset = 1; // Start with reset
18         right_shift = 0; // Initial shift direction
19         data_in1 = 4'b0000; data_in2 = 4'b0000; #10; // Initial values
20
21         reset = 0; #10; // Release reset
22
23         // Test shifting
24         data_in1 = 4'b1010; data_in2 = 4'b0101; #10; // Load test values
25         right_shift = 0; #10; // Left shift
26         right_shift = 1; #10; // Right shift
27
28         // Testing reset functionality
29         reset = 1; #10; // Apply reset
30         reset = 0; #10; // Release reset
31
32         data_in1 = 4'b1100; data_in2 = 4'b0011; #10; // Load more values
33         right_shift = 0; #10; // Left shift again
34         right_shift = 1; #10; // Right shift again
35
36         $finish;
37     end
38
39 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.

```
1 module four_bit_shift_circuit
2 (
```

```
3      input clock,
4      input reset,
5      input right_shift,
6      input [3:0] data_in1, data_in2,
7      output reg [3:0] data_out1, data_out2
8  );
9
10     always @(posedge clock or posedge reset) begin
11         if (reset) begin
12             data_out1 <= 4'b0000; // reset output1 to 0
13             data_out2 <= 4'b0000; // reset output2 to 0
14         end
15         else begin
16             if (right_shift) begin
17                 data_out1 <= data_in1 >> 1; // Shift right
18                 data_out2 <= data_in2 >> 1; // Shift right
19             end
20             else begin
21                 data_out1 <= data_in1 << 1; // Shift left
22                 data_out2 <= data_in2 << 1; // Shift left
23             end
24         end
25     end
26
27 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.

```
1  `include "addition.v"
2
3  module addition_tb;
```

```verilog
 4
 5     reg [3:0] a, b;
 6     reg carry_in;
 7     wire carry_out;
 8     wire [3:0] sum;
 9
10     addition Add (.a(a), .b(b), .carry_in(carry_in), .carry_out(carry_out),
        .sum(sum));
11
12     initial begin
13         // Test cases for addition
14         a = 4'b0000; b = 4'b0000; carry_in = 0; #10; // 0 + 0 = 0
15         a = 4'b0001; b = 4'b0001; carry_in = 0; #10; // 1 + 1 = 2
16         a = 4'b0010; b = 4'b0010; carry_in = 0; #10; // 2 + 2 = 4
17         a = 4'b0011; b = 4'b0001; carry_in = 0; #10; // 3 + 1 = 4
18
19         $finish;
20     end
21
22 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
 1 module addition
 2 (
 3     input [3:0] a, b,
 4     input carry_in,
 5     output carry_out,
 6     output [3:0] sum
 7 );
 8
 9     wire [4:0] addition_result;
10
11     assign addition_result = a + b + carry_in;
12     assign sum = addition_result[3:0];
13     assign carry_out = addition_result[4];
14
15 endmodule
```

15

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.

```verilog
`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.

```verilog
module subtraction
```

```
2  (
3      input [3:0] a, b,
4      input carry_in,
5      output carry_out,
6      output [3:0] difference
7  );
8
9      wire [4:0] subtraction_result;
10
11     assign subtraction_result = a - b - carry_in;
12     assign difference = subtraction_result[3:0];
13     assign carry_out = subtraction_result[4];
14
15 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.

```
1  `include "multiplication.v"
2
3  module multiplication_tb;
4
5      reg [3:0] a, b;
6      wire [7:0] product;
7
8      multiplication Mul (.a(a), .b(b), .product(product));
9
10     initial begin
11         // Test cases for multiplication
12         a = 4'b0001; b = 4'b0010; #10; // 1 * 2 = 2
13         a = 4'b0010; b = 4'b0011; #10; // 2 * 3 = 6
14         a = 4'b0001; b = 4'b0001; #10; // 1 * 1 = 1
15         a = 4'b0011; b = 4'b0010; #10; // 3 * 2 = 6
16
```

17

```
17          $finish;
18      end
19
20 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.

```
1 module multiplication
2 (
3      input [3:0] a, b,
4      output [7:0] product
5 );
6
7      assign product = a * b;
8
9 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.

```
1 `include "division.v"
2
3 module division_tb;
4
5      reg [3:0] a, b;
6      wire [3:0] quotient, remainder;
7
8      division Div (.a(a), .b(b), .quotient(quotient), .remainder(remainder))
         ;
9
10     initial begin
```

18

```
11          // Test cases for division
12          a = 4'b0100; b = 4'b0010; #10; // 4 / 2 = 2
13          a = 4'b0001; b = 4'b0001; #10; // 1 / 1 = 1
14          a = 4'b0010; b = 4'b0001; #10; // 2 / 1 = 2
15          a = 4'b0000; b = 4'b0001; #10; // 0 / 1 = 0
16          a = 4'b0100; b = 4'b0000; #10; // Division by zero
17
18          $finish;
19      end
20
21 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.

```
1 module division
2 (
3     input [3:0] a, b,
4     output [3:0] quotient, remainder
5 );
6
7     assign quotient = (b != 0) ? (a / b) : 4'b0000;
8     assign remainder = (b != 0) ? (a % b) : 4'b0000;
9
10 endmodule
```

15.1   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.

16   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 (NOT,

NAND, NOR) that are so critical to all electronics. 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.

17 Waveforms

17.1 And Circuit Waveform

At 0 s, we can see that both A and B are 0, so the output is 0
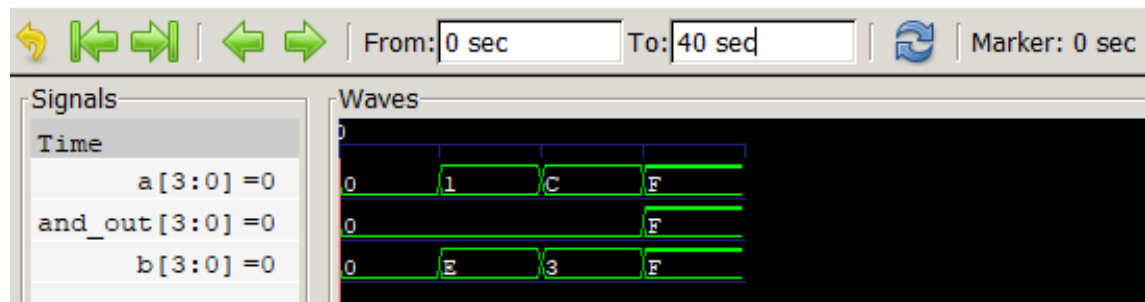
Figure 8: And Circuit with marker at 0s
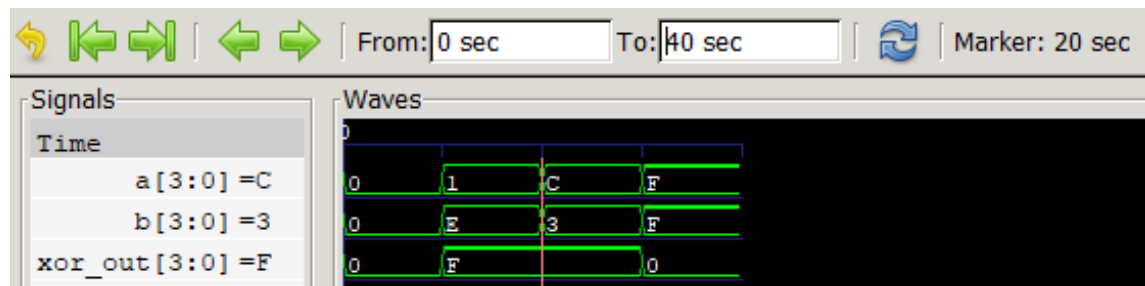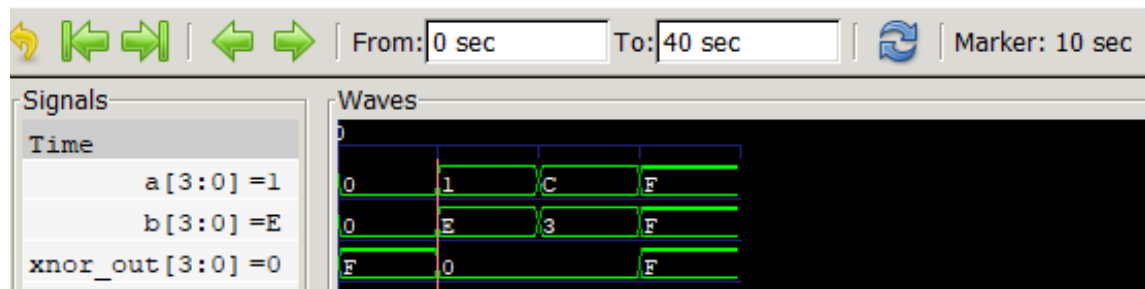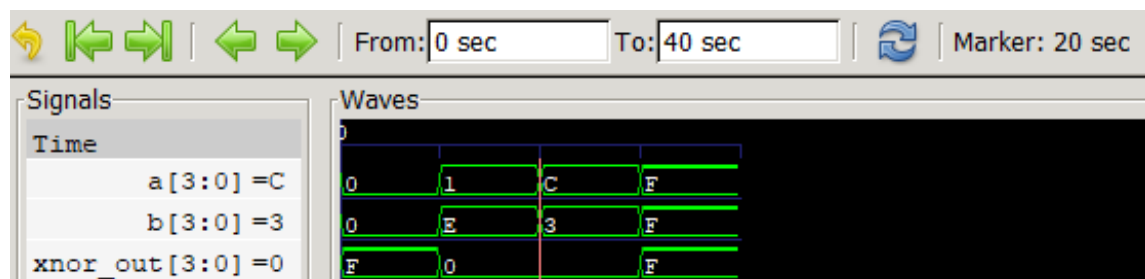
At 10 s, A is 1 and B is E, so the output is 0
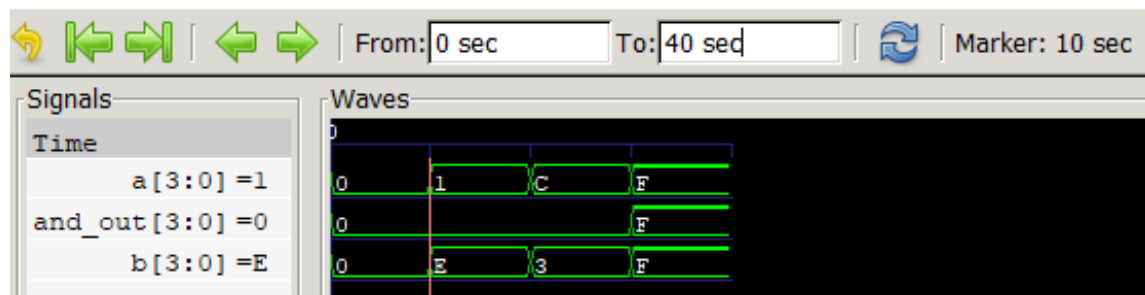
Figure 9: And Circuit with marker at 10s

At 20 s, A is 1 and B is E, so the output is 0
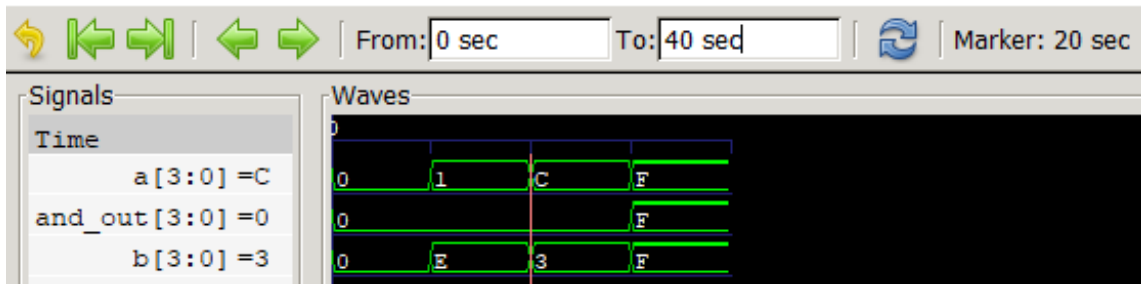
Figure 10: And Circuit with marker at 20s

At 30 s, A is F and B is F, so the output is F
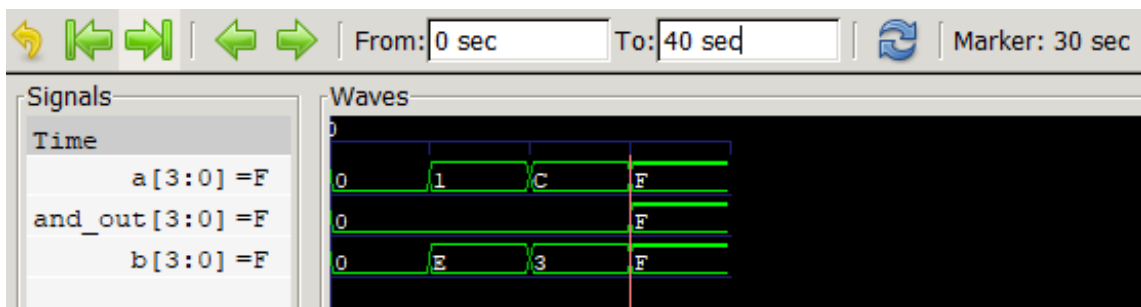
Figure 11: And Circuit with marker at 30s

17.2   Nand Circuit Waveform

At 0 s, we can see that both A and B are 0, so the output is F
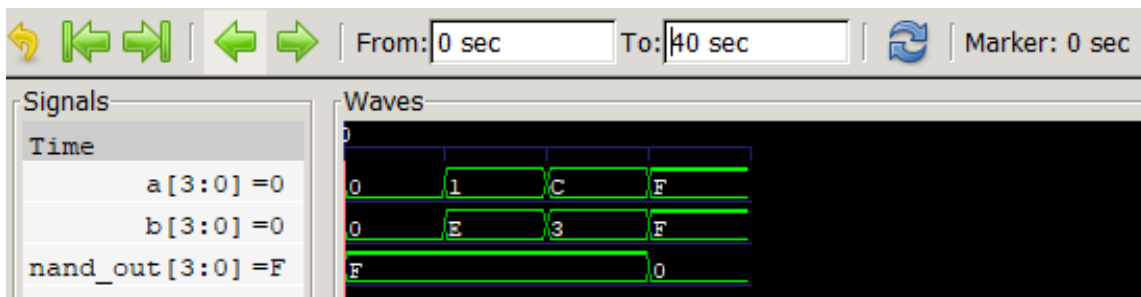
Figure 12: Nand Circuit with marker at 0s

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

22
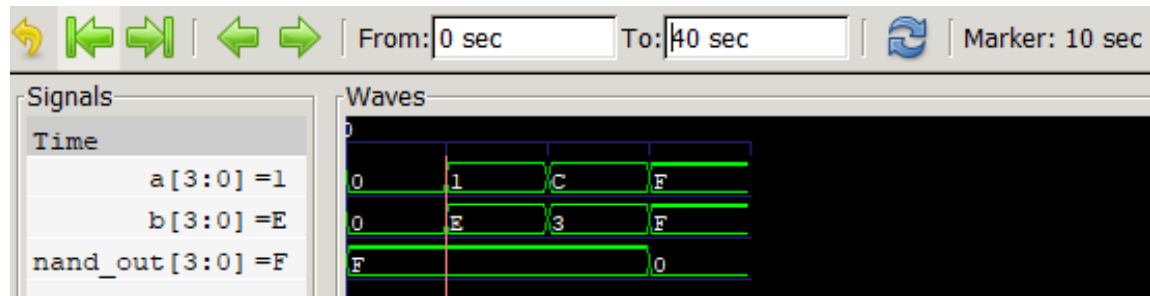
Figure 13: Nand Circuit with marker at 10s

At 20 s, A is C and B is 3, so the output is F
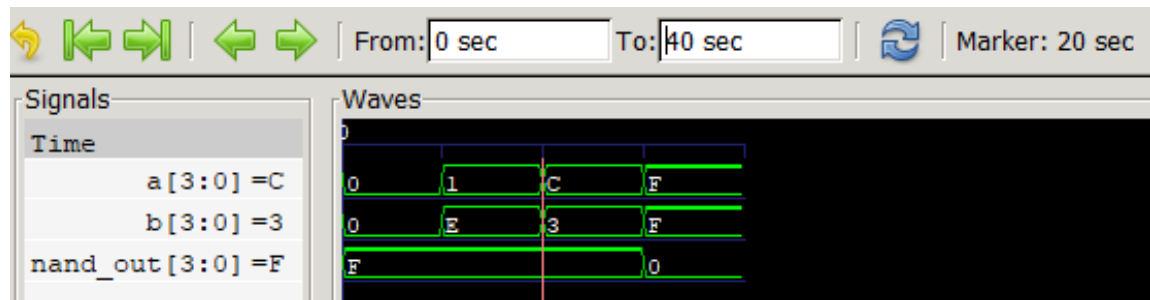
Figure 14: Nand Circuit with marker at 20s

At 30 s, A is F and B is F, so the output is 0
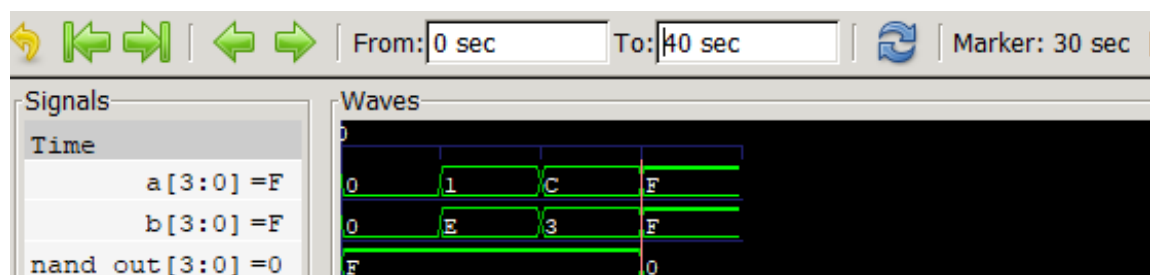
Figure 15: Nand Circuit with marker at 30s

23

17.3 Or Circuit Waveform

At 0 s, we can see that both A and B are 0, so the output is 0
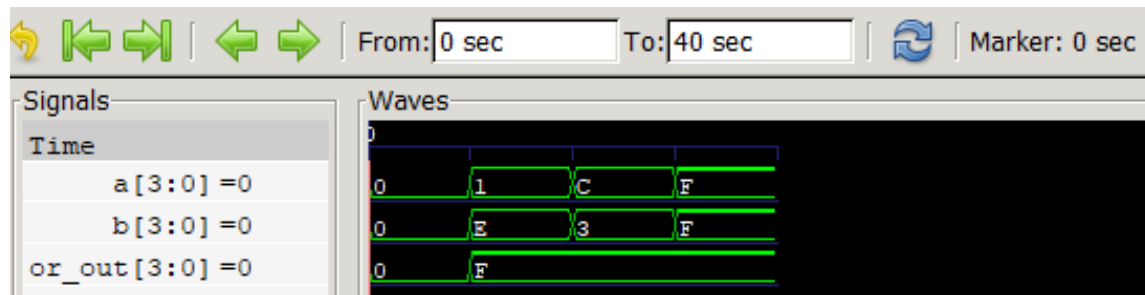
Figure 16: Or Circuit with marker at 0s

At 10 s, we can see that A is 1 and B is E, so the output is F
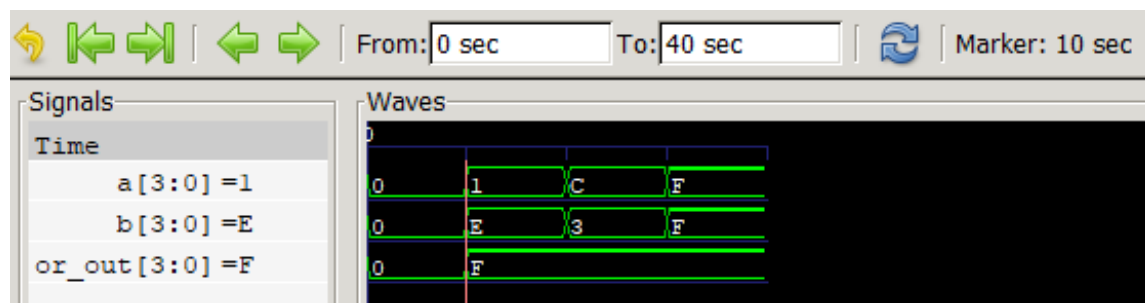
Figure 17: Or Circuit with marker at 10s

At 20 s, A is C and B is 3, so the output is F
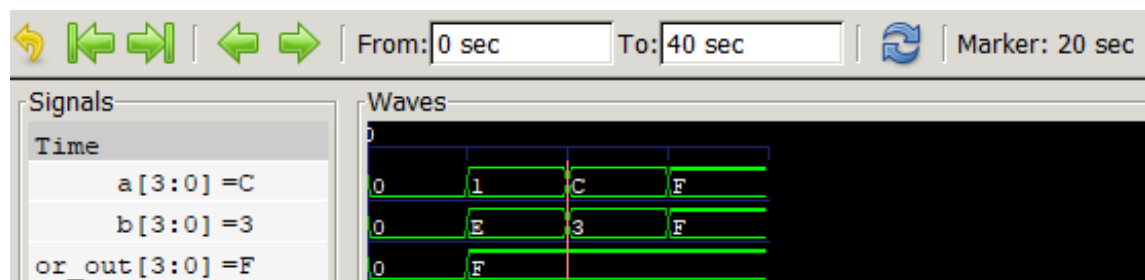
Figure 18: Or Circuit with marker at 20s

At 30 s, A is F and B is F, so the output is F
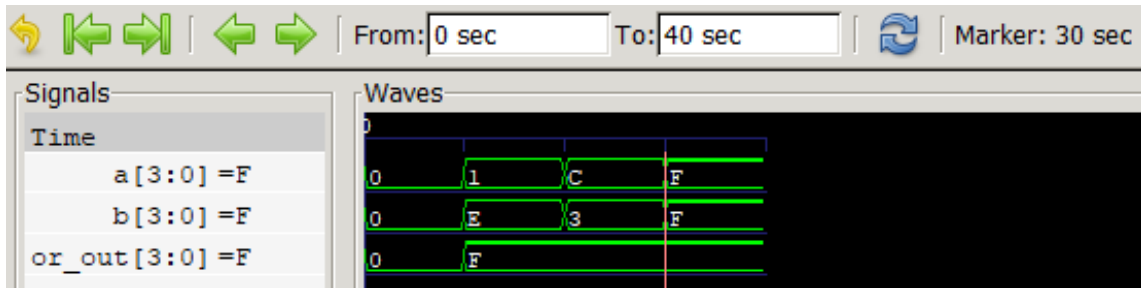
Figure 19: Or Circuit with marker at 30s

17.4 Nor Circuit Waveform

At 0 s, we can see that both A and B are 0, so the output is F
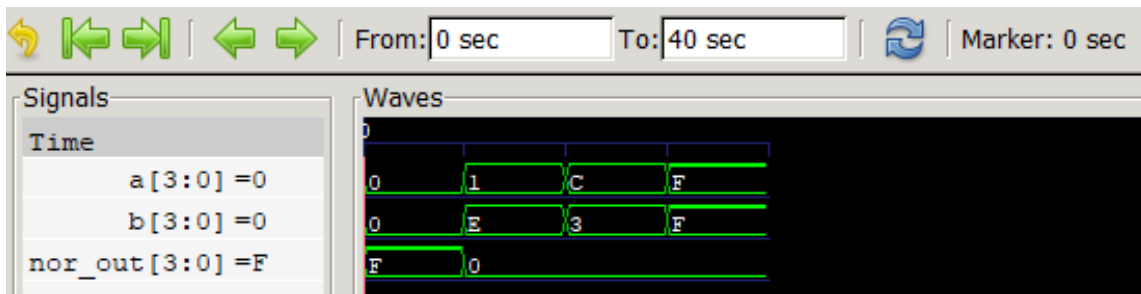
Figure 20: Nor Circuit with marker at 0s

At 10 s, we can see that A is 1 and B is E, so the output is 0
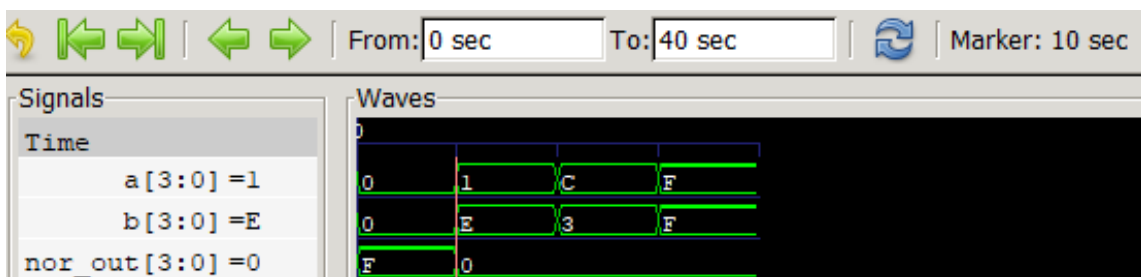
Figure 21: Nor Circuit with marker at 10s

At 20 s, A is C and B is 3, so the output is 0
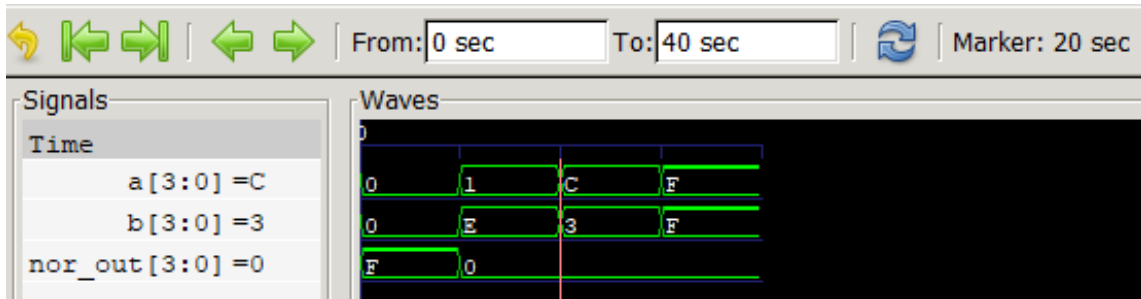
Figure 22: Nor Circuit with marker at 20s

At 30 s, A is F and B is F, so X is 0
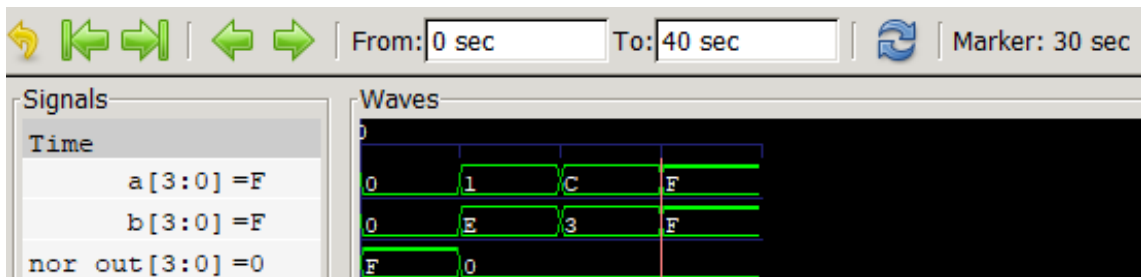
Figure 23: Nor Circuit with marker at 30s

17.5 Xor Circuit Waveform

At 0 s, we can see that both A and B are 0, so the output is 0
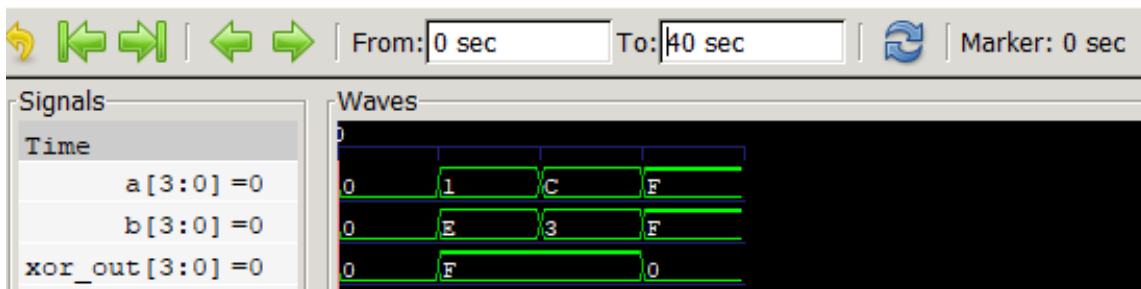
Figure 24: Xor Circuit with marker at 0s

26

At 10 s, we can see that A is 1 and B is E, so the output is F
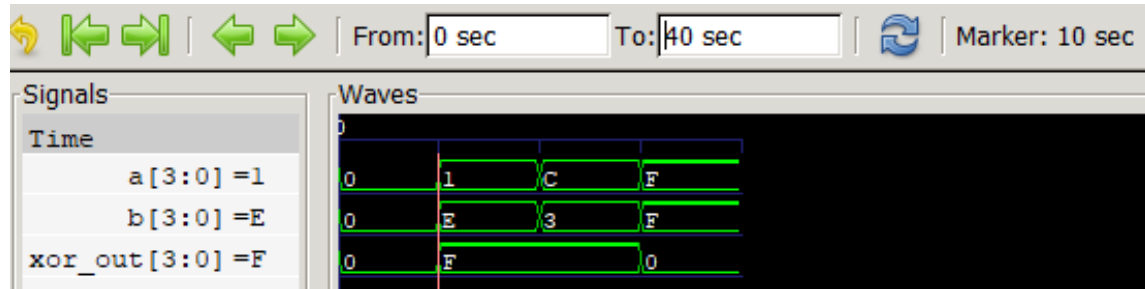
Figure 25: Xor Circuit with marker at 10s

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
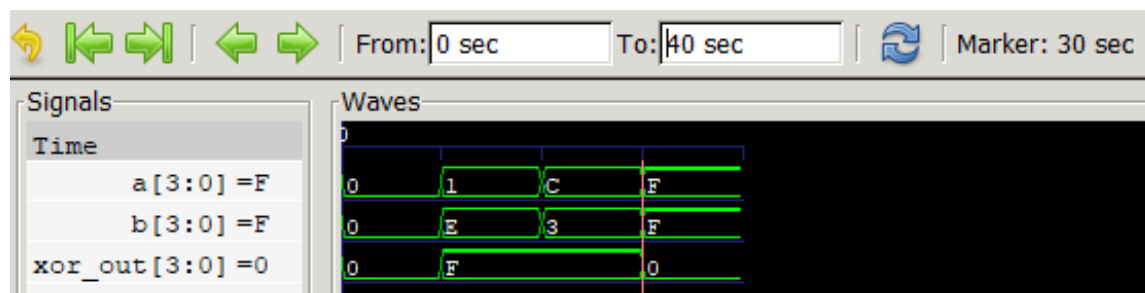
Figure 27: Xor Circuit with marker at 30s

17.6 Xnor Circuit Waveform

At 0 s, we can see that both A and B are 0, so the output is F
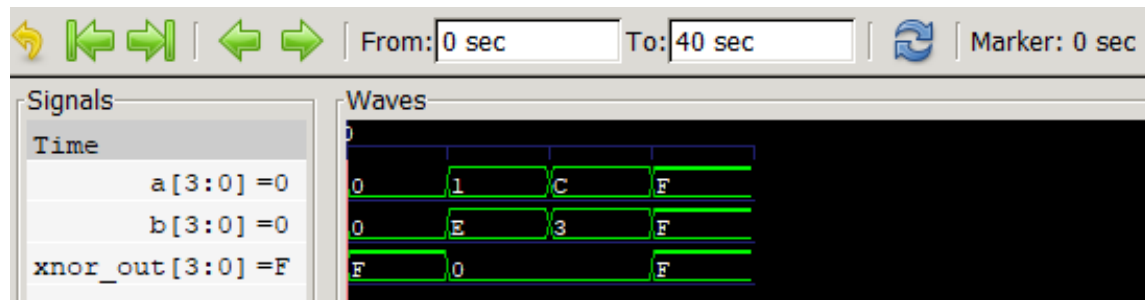
Figure 28: Xnor Circuit with marker at 0s

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
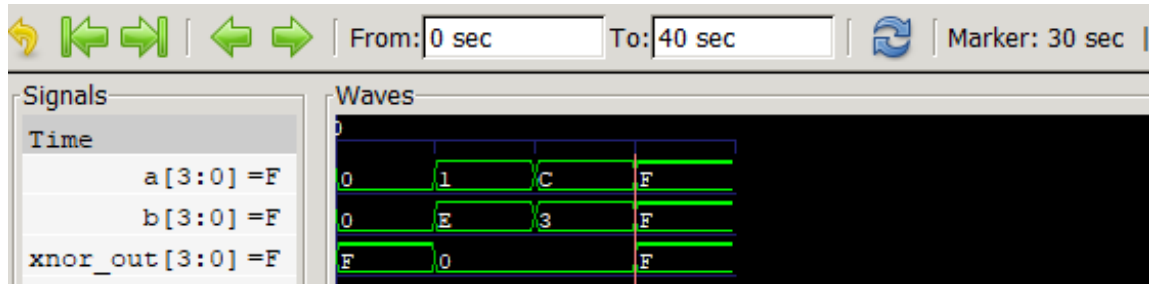
Figure 31: Xnor Circuit with marker at 30s

17.7   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
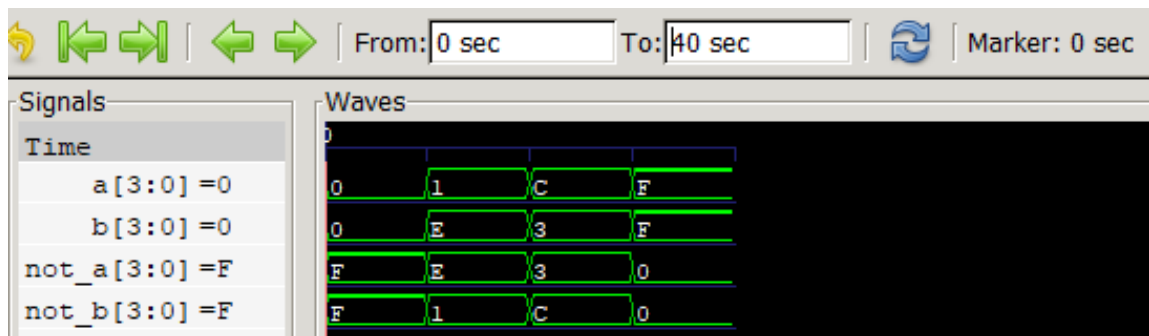
Figure 32: Not Circuit with marker at 0s

29

At 10 s, A is 1 and B is E, so the not A is E and the not B is 1
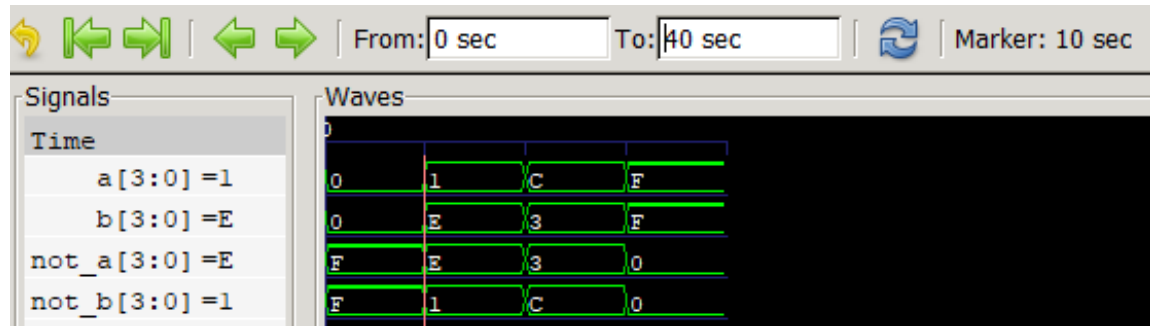
Figure 33: Not Circuit with marker at 10s

At 20 s, A is C and B is 3, so the not A is 3 and the not B is C
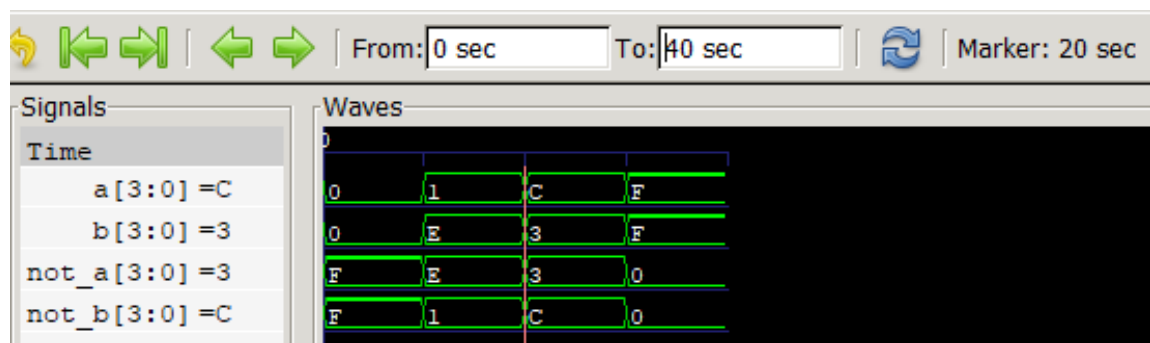
Figure 34: Not Circuit with marker at 20s

At 30 s, both A and B are F, so the not A is 0 and the not B is 0
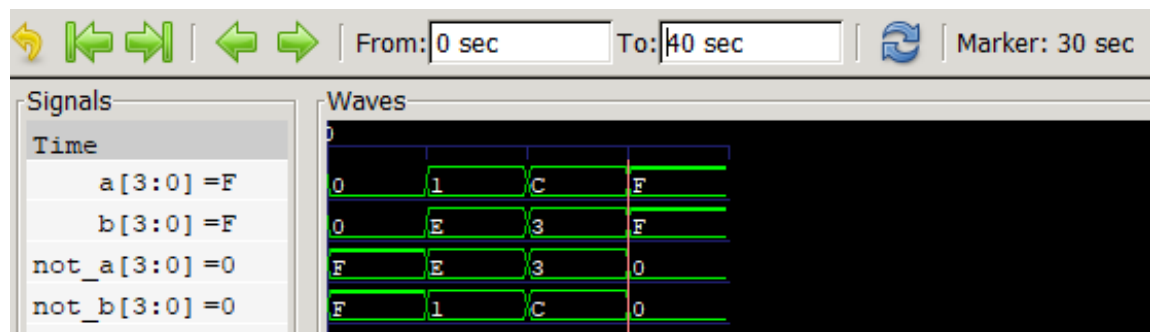
Figure 35: Not Circuit with marker at 30s

17.8 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
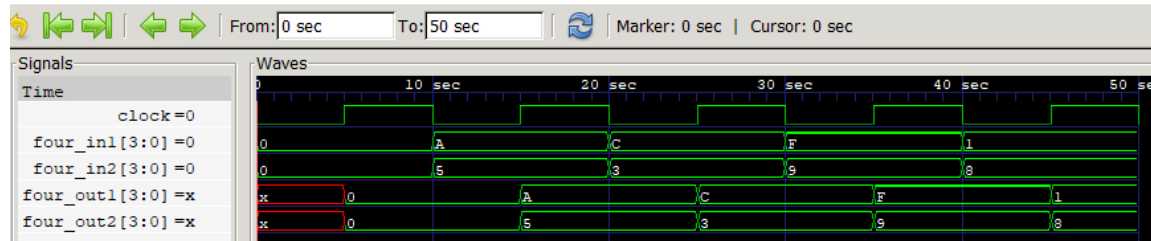
Figure 36: 2x4-bit input Circuit with marker at 0s

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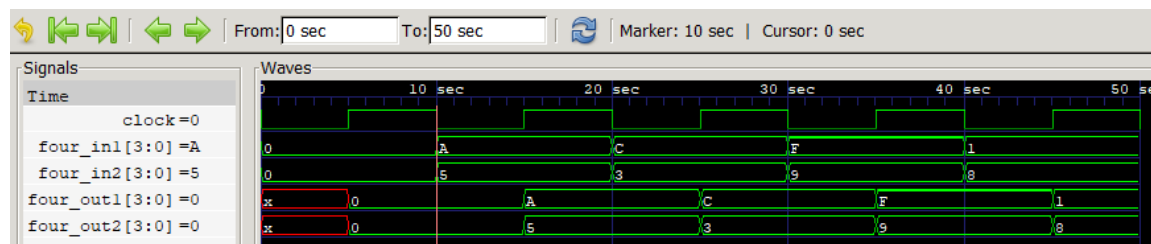
Figure 37: 2x4-bit input Circuit with marker at 10s

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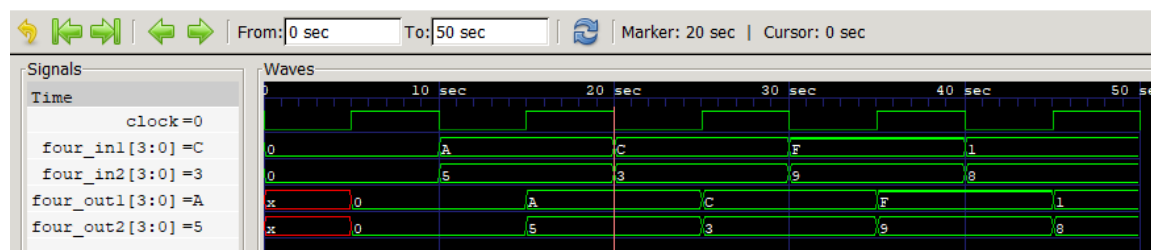
Figure 38: 2x4-bit input Circuit with marker at 20s

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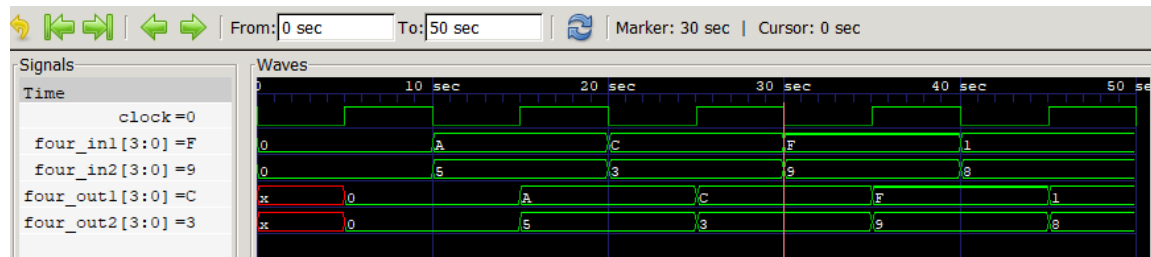
Figure 39: 2x4-bit input Circuit with marker at 30s

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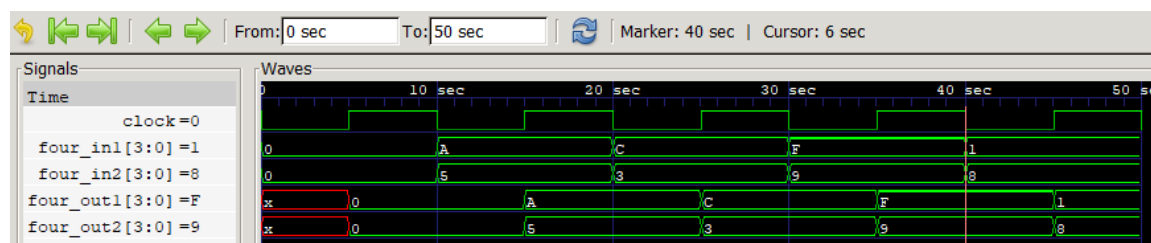
Figure 40: 2x4-bit input Circuit with marker at 40s

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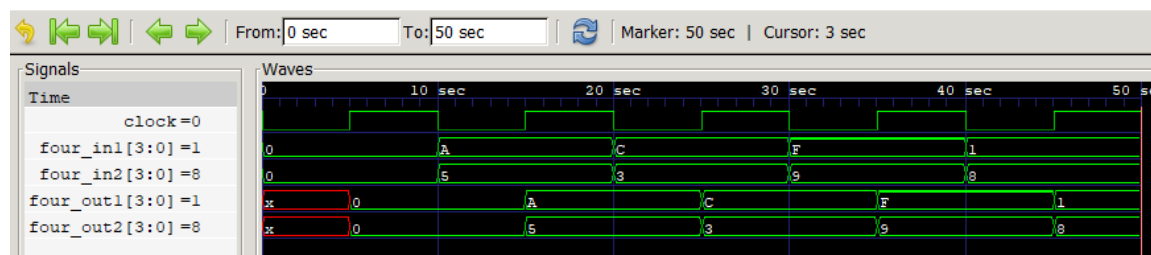
Figure 41: 2x4-bit input Circuit with marker at 50s

17.9   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
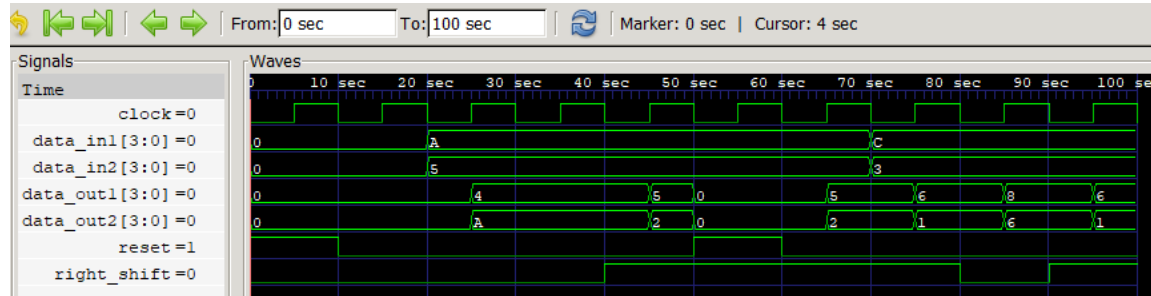
Figure 42: 2x4-bit output Shift Circuit with marker at 0s

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
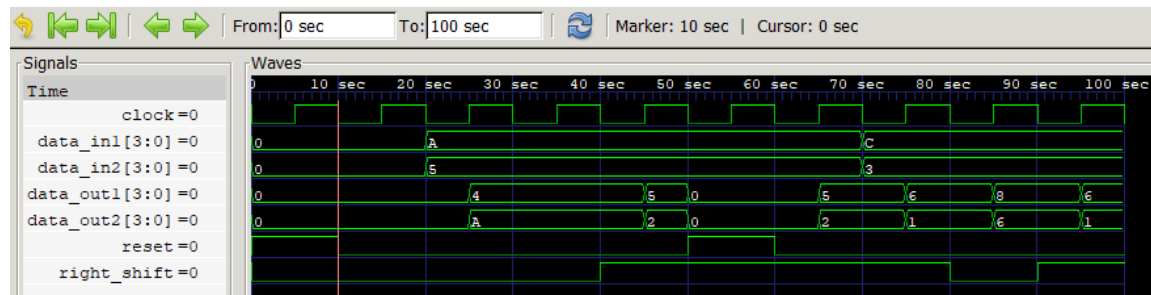
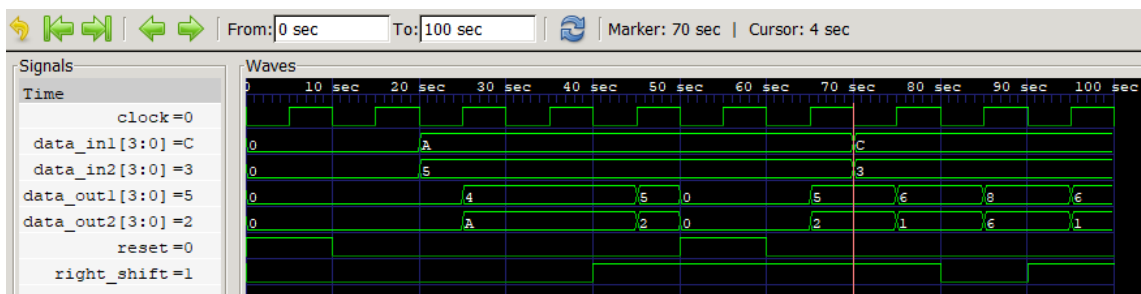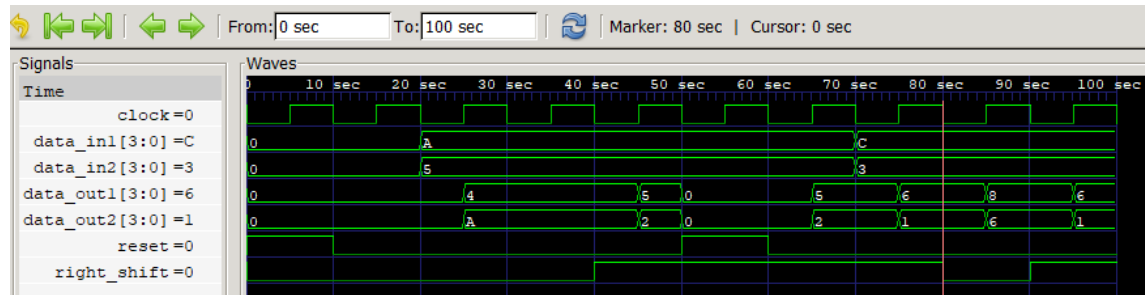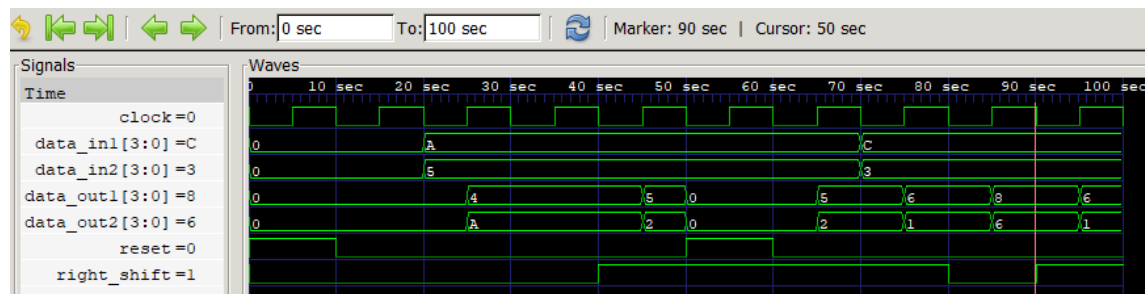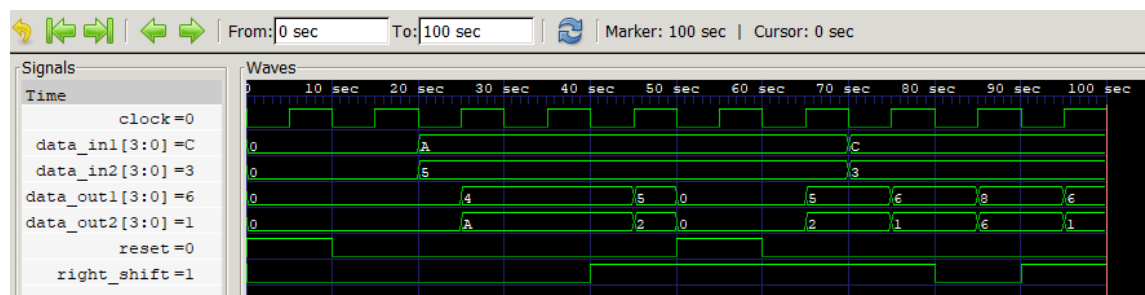

Figure 43: 2x4-bit output Shift Circuit with marker at 10s

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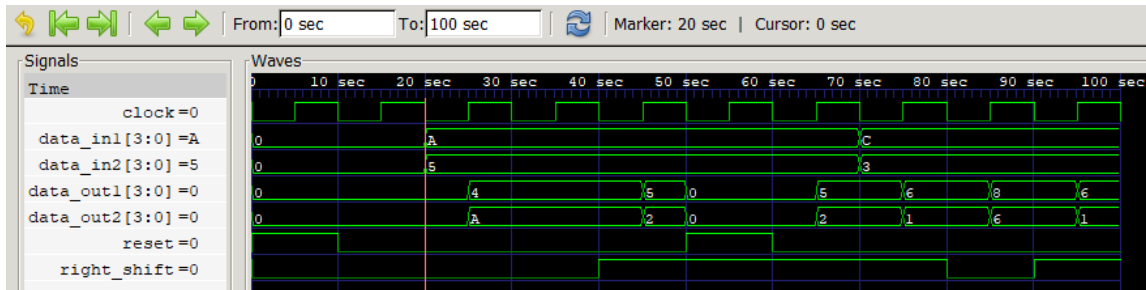
Figure 44: 2x4-bit output Shift Circuit with marker at 20s

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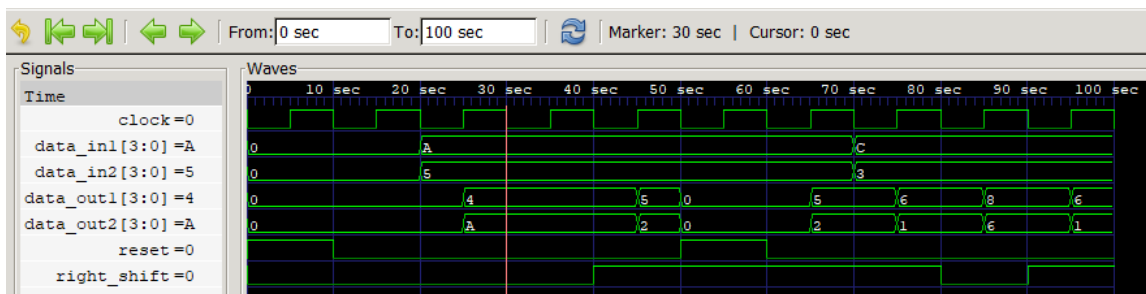
Figure 45: 2x4-bit output Shift Circuit with marker at 30s

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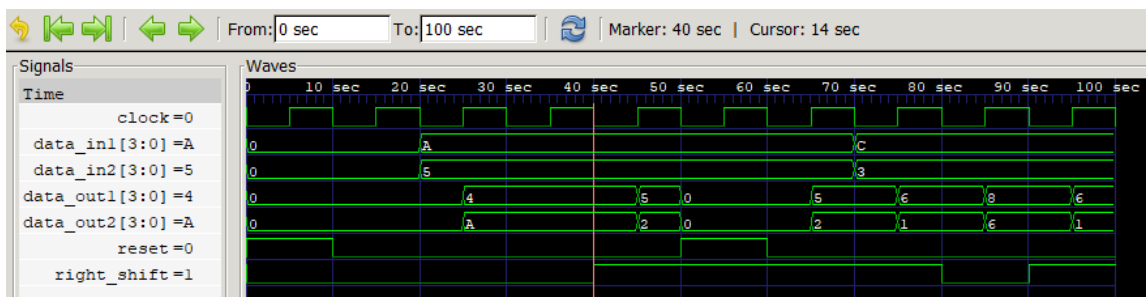
Figure 46: 2x4-bit output Shift Circuit with marker at 40s

34

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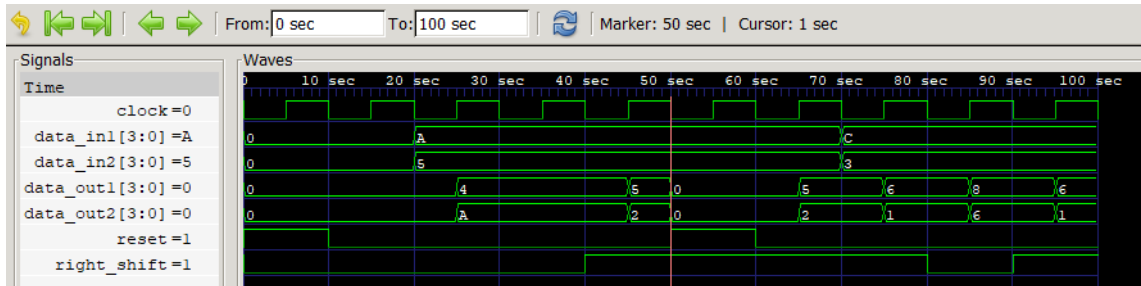
Figure 47: 2x4-bit output Shift Circuit with marker at 50s

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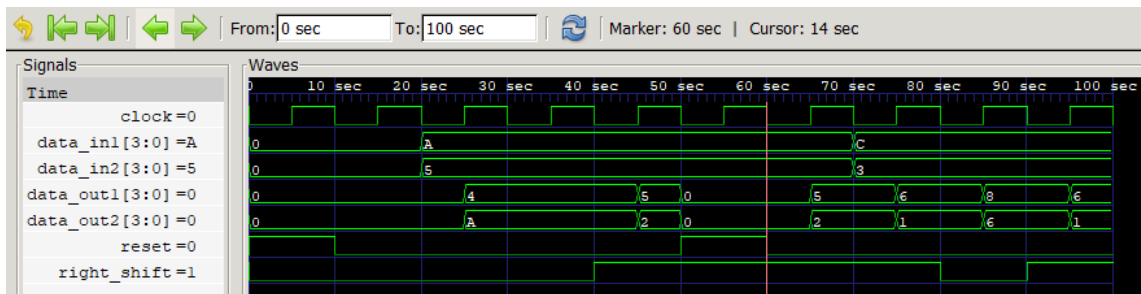
Figure 48: 2x4-bit output Shift Circuit with marker at 60s

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

35

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

17.10 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
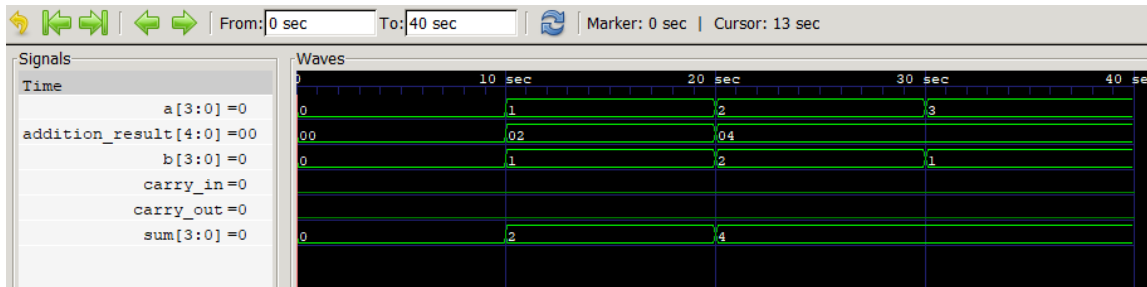
Figure 53: Addition Circuit with marker at 0s

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
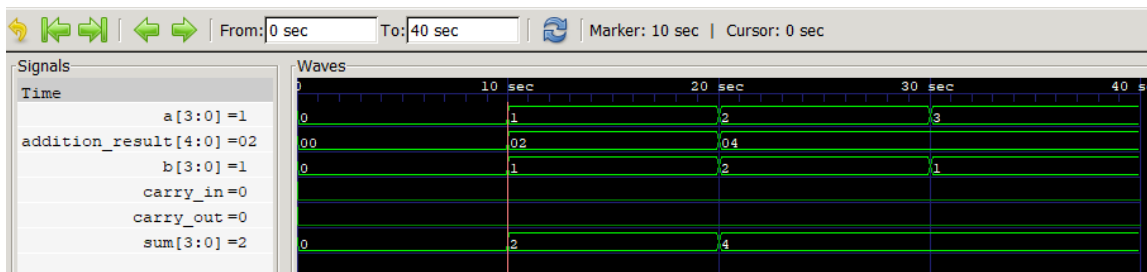
Figure 54: Addition Circuit with marker at 10s

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
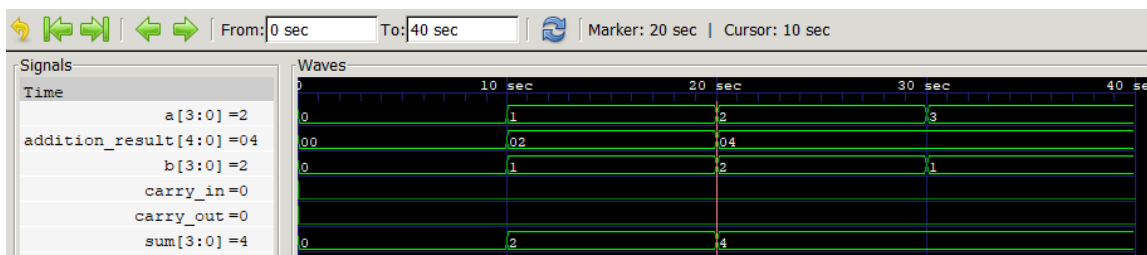
Figure 55: Addition Circuit with marker at 20s

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
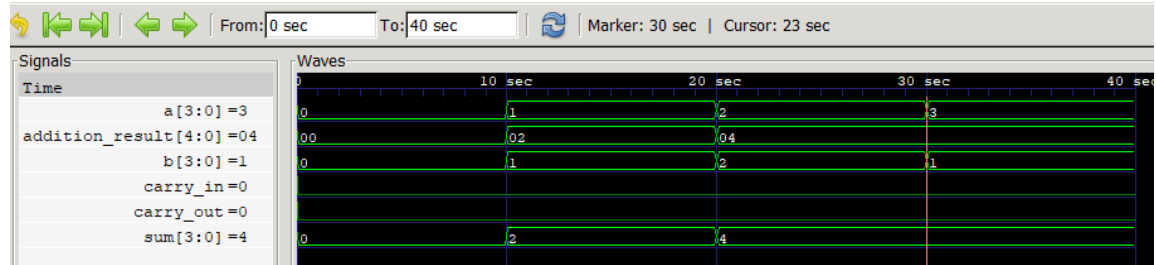
Figure 56: Addition Circuit with marker at 30s

17.11   Subtraction

At 0 s, we can see that A is 4 and B is 2, so then the difference is 2
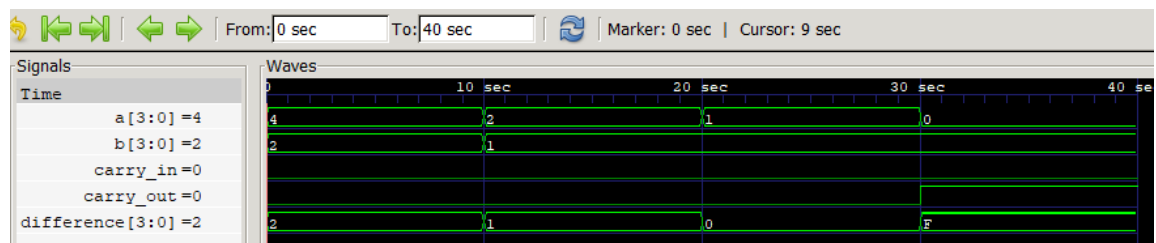
Figure 57: Subtraction Circuit with marker at 0s

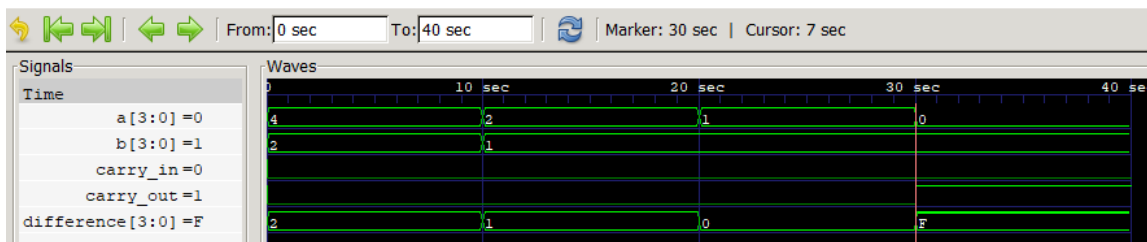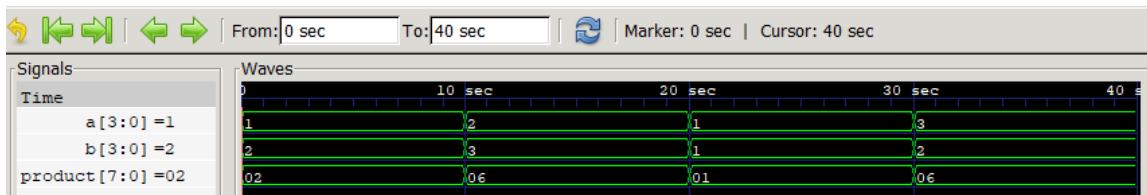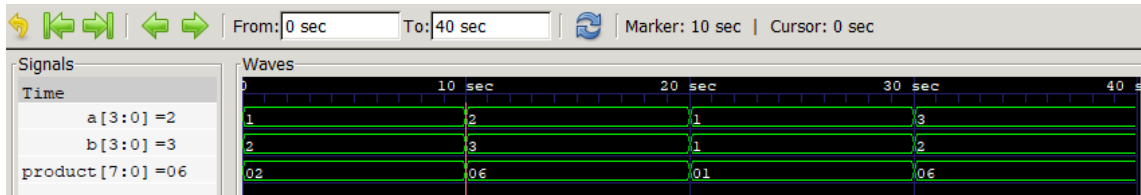At 10 s, we can see that A is 2 and B is 1, so then the difference is 1
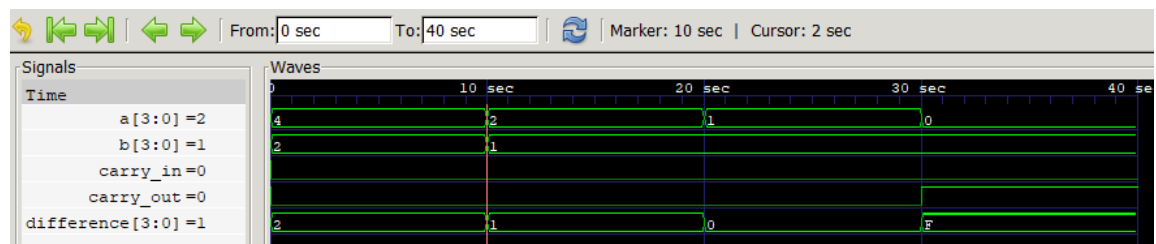
Figure 58: Subtraction Circuit with marker at 10s
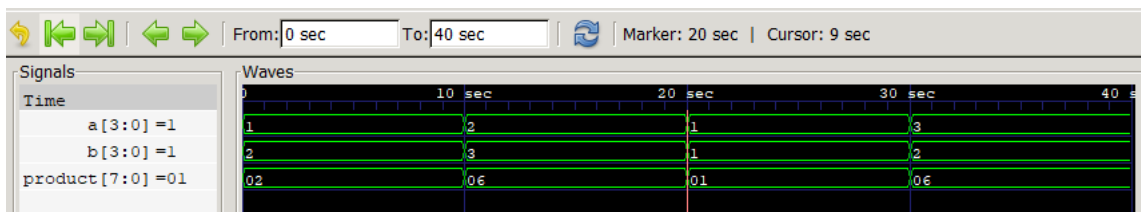
At 20 s, we can see that A is 1 and B is 1, so then the difference is 0
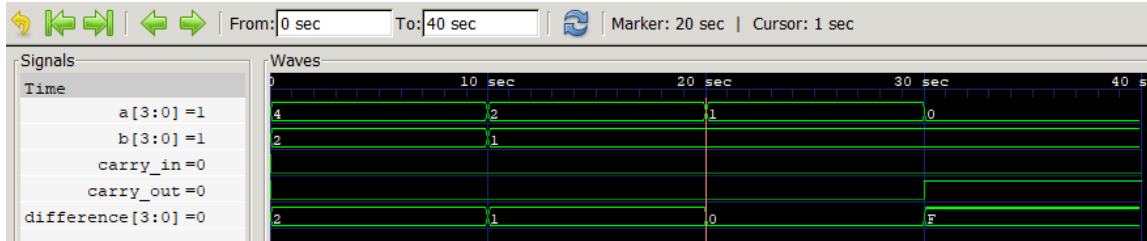
Figure 59: Subtraction Circuit with marker at 20s

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

Figure 60: Subtraction Circuit with marker at 30s

17.12   Multiplication

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

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

Figure 63: Multiplication Circuit with marker at 20s

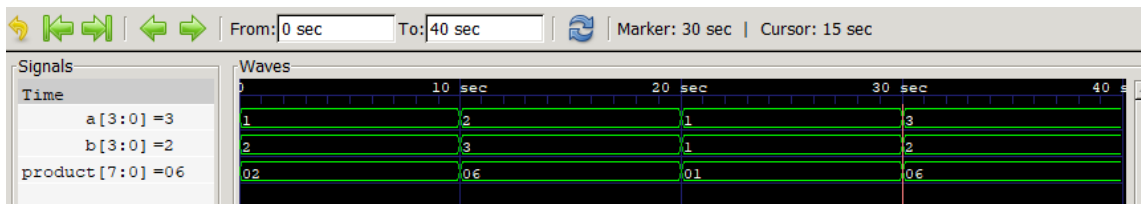At 30 s, we can see that A is 3 and B is 2, so then the product is 06

Figure 64: Multiplication Circuit with marker at 30s

17.13   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
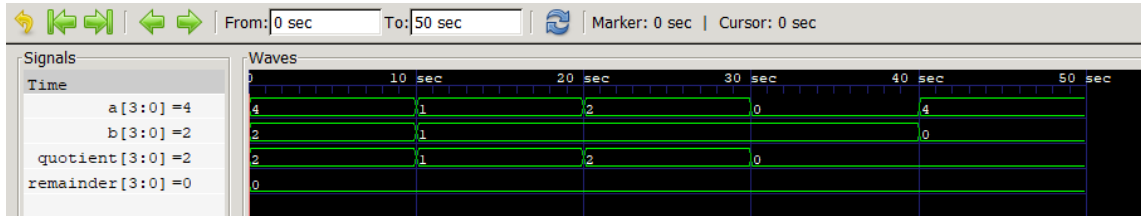

Figure 65: Division Circuit with marker at 0s

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
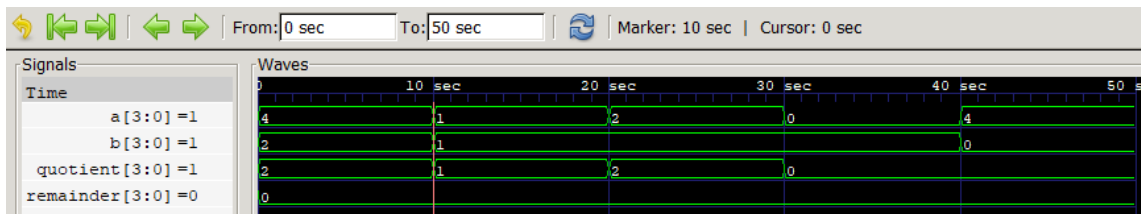

Figure 66: Division Circuit with marker at 10s

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
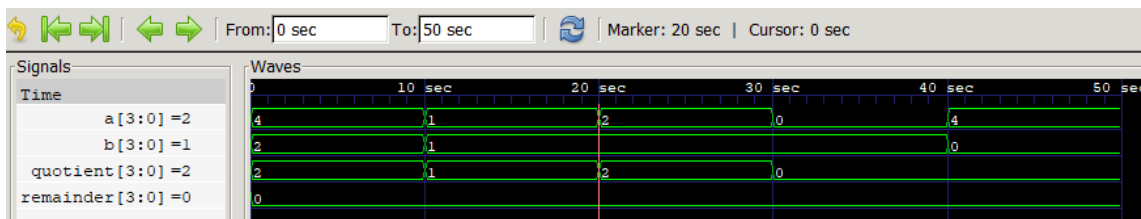

Figure 67: Division Circuit with marker at 20s

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
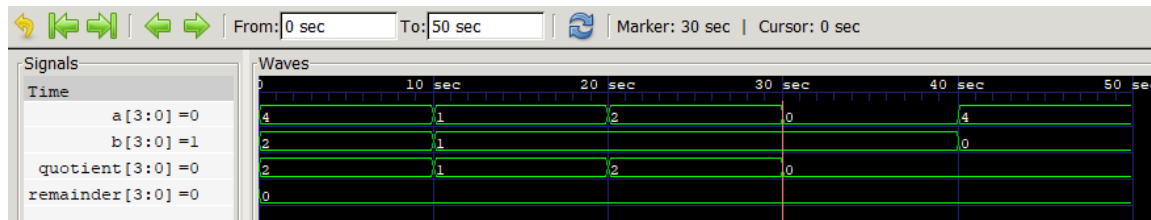

Figure 68: Division Circuit with marker at 30s

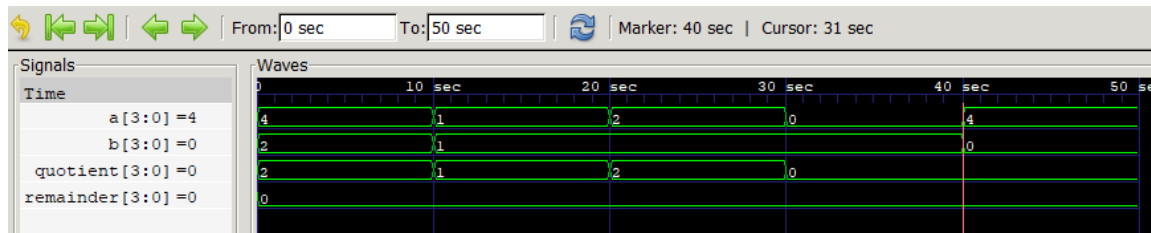At 40 s, we can see that A is 4 and B is 0, so that means that the quotient is 0 and the remainder is 0

Figure 69: Division Circuit with marker at 40s