CS 3339

Project: Step 2

Group: **Sleepy Café**
Authors: Ashley Cook, Sarah Gonzalez, Joseph Sheraden
Instructor: **Dr. Klepetko**
December 10, 2023

1 Introduction

The objective of this report is to demonstrate the process of designing, implementing, and analyzing 4-bit arithmetic circuits. This project focuses on a 4-bit adder, subtractor, and multiplier using Verilog hardware language. By designing and simulating these circuits, the students were able to verify their results.

2 4-bit Full Adder

The 4-bit Full Adder is a combination circuit with the ability to perform binary addition with two 4-bit numbers. Each bit utilizes full adders to obtain Carry-out and Sum output. In total, there are three inputs: A, B, and Carry-in. Then the logic creates the outputs, Sum and Carry-out. The Sum output is determined by the XOR of inputs A, B, and Carry-in. Carry-out is gathered by combining AND and OR gates and propagates the carry bit from one position to the next.

2.1 Execution Code

From a high level, this code implements 4-bit binary addition by coding bit-wise operations and managing our carry values.

- The program is started with a module declaration and variable initialization in order to store inputs and outputs. The [3:0] is an array that indicates the size of our variable.

- Calculations are performed in the module using binary bit-wise addition from LSB to MSB. The 'Assign' operator computes the Sum and C-out for each bit position. **'assign Sum[0] = A[0] B̂[0] Ĉin;'** This line of code calculates sum of bit 0 using XOR(ˆ) operator on the three input bits. The result is then stored in 'Sum[0]'. **assign Cout = (A[0] B[0]) — (A[0] & Cin) — (B[0] & Cin);** This line of code calculates the C-out bit for bit 1 position. The operators AND(&) and OR(—) are combined and applied to the input value. The result is stored in C-out.

- The code follows the previously established patterh for the next three bits, 1 - 3, and calculates the Sum and Cout. The last outputs for Sum and Cout are the 4-bit result of binary addition and Cout from the MSB position.

```
module FullAdder_4bit(
    input [3:0] A, // 4-bit input A
    input [3:0] B, // 4-bit input B
    input Cin,    // Carry-in
    output [3:0] Sum, // 4-bit output sum
    output Cout      // Carry-out
```

```
);
// Intermediate carry values for each bit position
wire [3:0] Carry;
// Calculate sum and carry-out bit-wise for each bit position
assign Sum[0] = A[0] ^ B[0] ^ Cin;
assign Carry[0] = (A[0] & B[0]) | (A[0] & Cin) | (B[0] & Cin);

assign Sum[1] = A[1] ^ B[1] ^ Carry[0];
assign Carry[1] = (A[1] & B[1]) | (A[1] & Carry[0]) | (B[1] & Carry[0]);

assign Sum[2] = A[2] ^ B[2] ^ Carry[1];
assign Carry[2] = (A[2] & B[2]) | (A[2] & Carry[1]) | (B[2] & Carry[1]);

assign Sum[3] = A[3] ^ B[3] ^ Carry[2];
assign Cout = (A[3] & B[3]) | (A[3] & Carry[2]) | (B[3] & Carry[2]);
endmodule
```

2.2 Test Bench

A test bench was created to mimic an environment need to verify the correctness of our computation designs. The test bench includes ten separate cases and different inputs to check the quality of our simulation.

- The 'testbench' module is used to define our 4-bit full adder module. It declares A, B, and Cin for 4-bit reg data types and outputs Sum and Cout. The FullAdder_4bit module is then initialized Design Under Test('dut' to connect the input and output ports to the 'testbench' module corresponding signals. A clock signal is coded to toggle every PERIOD/2 time unit. This clock signal is used to control the timing of the circuit simulation. The 'always' function allows the 'testbench' to have consistent time intervals, to help with synchronous operations. Each test case will be executed at the correct time.

- A block is then included to provide 'initial' states of the simulation. Inputs A, B, and Cin are allocated for each of the ten test cases. Our 'PERIOD' is the time dictated for each cases' waiting period before the next test case is executed. An 'always' line is included to ensure the clk continues to follow set parameters until the test cases are complete.

```
module testbench;
  parameter PERIOD = 10;
  // Inputs
  reg [3:0] A;
  reg [3:0] B;
  reg Cin;
  // Outputs
  wire [3:0] Sum;
```

```
wire Cout;

FullAdder_4bit dut (.A(A),.B(B),.Cin(Cin),.Sum(Sum),.Cout(Cout));

// Clock generation
reg clk = 0;
always #((PERIOD)/2) clk = ~clk;

initial begin
  $display("Testbench");
  // Test case 1
  A = 4'b0011;
  B = 4'b0010;
  Cin = 0;
  #PERIOD;
  $display("A = %b, B = %b, Cin = %b, Sum = %b, Cout = %b", A, B, Cin, Sum, Cout);

  // Test case 2
  A = 4'b1111;
  B = 4'b0001;
  Cin = 1;
  #PERIOD;
  $display("A = %b, B = %b, Cin = %b, Sum = %b, Cout = %b", A, B, Cin, Sum, Cout);

  // Test case 3
  A = 4'b0110;
  B = 4'b1001;
  Cin = 0;
  #PERIOD;
  $display("A = %b, B = %b, Cin = %b, Sum = %b, Cout = %b", A, B, Cin, Sum, Cout);

  // Test case 4
  A = 4'b1010;
  B = 4'b0101;
  Cin = 0;
  #PERIOD;
  $display("A = %b, B = %b, Cin = %b, Sum = %b, Cout = %b", A, B, Cin, Sum, Cout);

  // Test case 5
  A = 4'b0000;
  B = 4'b1111;
  Cin = 0;
  #PERIOD;
  $display("A = %b, B = %b, Cin = %b, Sum = %b, Cout = %b", A, B, Cin, Sum, Cout);

  // Test case 6
  A = 4'b1011;
  B = 4'b0100;
  Cin = 0;
  #PERIOD;
```

```
    $display("A = %b, B = %b, Cin = %b, Sum = %b, Cout = %b", A, B, Cin, Sum, Cout);

    // Test case 7
    A = 4'b1100;
    B = 4'b0010;
    Cin = 0;
    #PERIOD;
    $display("A = %b, B = %b, Cin = %b, Sum = %b, Cout = %b", A, B, Cin, Sum, Cout);

    // Test case 8
    A = 4'b1011;
    B = 4'b1010;
    Cin = 0;
    #PERIOD;
    $display("A = %b, B = %b, Cin = %b, Sum = %b, Cout = %b", A, B, Cin, Sum, Cout);

    // Test case 9
    A = 4'b1001;
    B = 4'b1111;
    Cin = 0;
    #PERIOD;
    $display("A = %b, B = %b, Cin = %b, Sum = %b, Cout = %b", A, B, Cin, Sum, Cout);

    // Test case 10
    A = 4'b1000;
    B = 4'b1010;
    Cin = 0;
    #PERIOD;
    $display("A = %b, B = %b, Cin = %b, Sum = %b, Cout = %b", A, B, Cin, Sum, Cout);

    $finish; // End the simulation
  end

  always #((PERIOD)/2) clk = ~clk;
endmodule
```

3 4-bit Full Subtractor

A subtractor can be created by modifying the preexisting adder. The ripple-carry method is applied to the adder to produce a subtractor. The carry bit "ripples" from one position to the next bit position for computation. Each bit for adding/subtracting depends on the previous bit result.

– The module RC_add_sub represent the bulk of our ripple carry add-sub. There are 4-bit inputs A,B, and Op, and single-bit Carry-out. Wire connections are established for the internal connections of our values. The block containing the line: xor(B0, B[0], Op); inverts each value, performing two's compliment on the bits on the condition that Op=1. The Carry and V outputs are obtained by the XOR operator for calculating the Cout and overflow status.

3.1 Execution Code

```
module RC_add_sub(Sum, Carry, V, A, B, Op);
   output [3:0] Sum;
   output      Carry;
   output      V; //overflow status
   input [3:0] A;
   input [3:0] B;
   input       Op; //operation 0 = add, 1 = subtract

   wire Carry0;
   wire Carry1;
   wire Carry2;
   wire Carry3;

   wire B0;
   wire B1;
   wire B2;
   wire B3;

   xor(B0, B[0], Op);
   xor(B1, B[1], Op);
   xor(B2, B[2], Op);
   xor(B3, B[3], Op);
   xor(Carry, Carry3, Op);
   xor(V, Carry3, Carry2);

   full_adder fa0(Sum[0], Carry0, A[0], B0, Op);
   full_adder fa1(Sum[1], Carry1, A[1], B1, Carry0);
   full_adder fa2(Sum[2], Carry2, A[2], B2, Carry1);
   full_adder fa3(Sum[3], Carry3, A[3], B3, Carry2);

endmodule // ripple carry adder subtractor

module full_adder(Sum, Cout, A, B, Cin);
   output Sum;
   output Cout;
   input A;
   input B;
   input Cin;

   wire w1;
   wire w2;
   wire w3;
   wire w4;
```

```
    xor(w1, A, B);
    xor(Sum, Cin, w1);
    and(w2, A, B);
    and(w3, A, Cin);
    and(w4, B, Cin);
    or(Cout, w2, w3, w4);
endmodule //full_adder
```

3.2 Test Bench

The testbenchs were created to test the RC adder/subtractor. Both test benches use ten cases each to verify the overall functionality. Since the test benches are independent of once another, the summary can solely focus on the subtractor test bench. But both test benches are used to verify that 'RC_add_sub' module works properly and for all possible scenarios.

– The 'testbench_sub' module tests the validity of the 'RC_add_sub' module. The two's complement method is used to perform the subtraction. The inputs A, B, and Op are the numbers to be subtracted under the Op=1 condition. When Op=1, the XOR operation inverts the current bit. Different input values are used to test various scenarios of subtraction. The test bench uses a clk to trigger the execution of the test cases. After each test case, A, B, Op, Sum, Carry, and V are $display to show the resulted value.

```
module testbench_add;
  parameter PERIOD = 10;
  // Inputs
  reg [3:0] A;
  reg [3:0] B;
  reg Op;
  // Outputs
  wire [3:0] Sum;
  wire Carry;
  wire V;

  RC_add_sub dut (.Sum(Sum),.Carry(Carry),.V(V),.A(A),.B(B),.Op(Op));

  reg clk = 0;
  always #((PERIOD)/2) clk = ~clk;

  // Test stimulus for addition
  initial begin

    $display("Addition testbench");

    // Test case 1
    A = 4'b0011;
```

```
    B = 4'b0010;
    Op = 0;
    #PERIOD;
    $display("A = %b, B = %b, Op = %b, Sum = %b, Carry = %b, V = %b", A, B, Op, Sum, Carry, V);

    // Test case 2
    A = 4'b1111;
    B = 4'b0001;
    Op = 0;
    #PERIOD;
    $display("A = %b, B = %b, Op = %b, Sum = %b, Carry = %b, V = %b", A, B, Op, Sum, Carry, V);

    //$finish;
  end

  always #((PERIOD)/2) clk = ~clk;
endmodule


module testbench_sub;
  // Parameters
  parameter PERIOD = 10;

  // Inputs
  reg [3:0] A;
  reg [3:0] B;
  reg Op;
  // Outputs
  wire [3:0] Sum;
  wire Carry;
  wire V;

  RC_add_sub dut (.Sum(Sum),.Carry(Carry),.V(V),.A(A),.B(B),.Op(Op));

  reg clk = 0;
  always #((PERIOD)/2) clk = ~clk;

  initial begin

    $display("Subtraction testbench");

    // Test case 1
    A = 4'b0110;
    B = 4'b0001;
    Op = 1;
    #PERIOD;
    $display("A = %b, B = %b, Op = %b, Sum = %b, Carry = %b, V = %b", A, B, Op, Sum, Carry, V);

    // Test case 2
    A = 4'b1010;
```

```
B = 4'b1001;
Op = 1;
#PERIOD;
$display("A = %b, B = %b, Op = %b, Sum = %b, Carry = %b, V = %b", A, B, Op, Sum, Carry, V);

// Test case 3
A = 4'b1111;
B = 4'b1010;
Op = 1;
#PERIOD;
$display("A = %b, B = %b, Op = %b, Sum = %b, Carry = %b, V = %b", A, B, Op, Sum, Carry, V);

// Test case 4
A = 4'b1010;
B = 4'b0101;
Op = 1;
#PERIOD;
$display("A = %b, B = %b, Op = %b, Sum = %b, Carry = %b, V = %b", A, B, Op, Sum, Carry, V);

// Test case 5
A = 4'b0000;
B = 4'b1111;
Op = 1;
#PERIOD;
$display("A = %b, B = %b, Op = %b, Sum = %b, Carry = %b, V = %b", A, B, Op, Sum, Carry, V);

// Test case 6
A = 4'b1011;
B = 4'b0100;
Op = 1;
#PERIOD;
$display("A = %b, B = %b, Op = %b, Sum = %b, Carry = %b, V = %b", A, B, Op, Sum, Carry, V);

// Test case 7
A = 4'b1100;
B = 4'b0010;
Op = 1;
#PERIOD;
$display("A = %b, B = %b, Op = %b, Sum = %b, Carry = %b, V = %b", A, B, Op, Sum, Carry, V);

// Test case 8
A = 4'b1011;
B = 4'b1010;
Op = 1;
#PERIOD;
$display("A = %b, B = %b, Op = %b, Sum = %b, Carry = %b, V = %b", A, B, Op, Sum, Carry, V);

// Test case 9
A = 4'b1001;
B = 4'b1111;
```

```
    Op = 1;
    #PERIOD;
    $display("A = %b, B = %b, Op = %b, Sum = %b, Carry = %b, V = %b", A, B, Op, Sum, Carry, V);

    // Test case 10
    A = 4'b1000;
    B = 4'b1010;
    Op = 1;
    #PERIOD;
    $display("A = %b, B = %b, Op = %b, Sum = %b, Carry = %b, V = %b", A, B, Op, Sum, Carry, V);


    $finish;
  end

  always #((PERIOD)/2) clk = ~clk;
endmodule
```

4 4-bit Full Multiplier

4.1 Execution Code

The purpose of this multiplier is to have two 4-bit inputs produce an 8-bit output as a result of multiplication. A series of partial products are accumulated through the code to obtain the final 8-bit product.

- The 'multi_4bit' module initializes the inputs and calculates the partial products. The block of code containing this line: 'wire [3:0]m0;', stores the partial product results for each LSB of 'a' multiplied by 'b'. The sun is stored throughout each stage.
- The block of code containing: 'assign m0 = 4a[0] b[3:0];' calculates the partial product for the specified bit of 'a' and performs bitwise AND on the lower 4bits of 'b'. For 'a[0]' is repeated four times since it is a 4-bit number.
- The last block of code calculates the sums for each product stage. It adds 'm0' to the partial product of 'a[1]' with 'b' shifted one bit to the left. This process is repeated to the next sums. Finally, the final sum is stored in 'p' output.

```
module multi_4bit(a, b, p);
    input [3:0]a,b;
    wire [3:0]m0;
    wire [4:0]m1;
    wire [5:0]m2;
    wire [6:0]m3;
```

```
    wire[7:0]s1,s2,s3;
    output[7:0]p;

    assign m0 = {4{a[0]}} & b[3:0];
    assign m1 = {4{a[1]}} & b[3:0];
    assign m2 = {4{a[2]}} & b[3:0];
    assign m3 = {4{a[3]}} & b[3:0];

    assign s1 = m0 + (m1<<1);
    assign s2 = s1 + (m2<<2);
    assign s3 = s2 + (m3<<3);
    assign p = s3;
endmodule
```

4.2 Test Bench

This testbench for the multiplier, applies different input combinations of A and B and checks the out for validity. Ten different test cases are used with various inputs are used and then the results are printed.

- The testbench module first defines the time period for the clk in the simulation. The input are initialized for A, and B, and the 8-bit output 'Result' stores the product of A and B. The Design Under Test(dut) initiates the 'multi_4bit' module with the inputs A, B, and Result output. The clock is then generated with a half-period delay and 50% duty cycle.

- The test cases block has ten cases that use inputs for A and B. Then, The simulation adds the use of #PERIOD to ensure that each result is shown before the next result is executed. Display statements print out the values and results for each test case. The clock driver is included at the code's end to ensure the clock signal is consistent throughout the simulation.

```
module testbench;.
  parameter PERIOD = 10; // Time period for clock (in simulation ticks)
  // Inputs
  reg [3:0] A;
  reg [3:0] B;
  // Outputs
  wire [7:0] Result;

  multi_4bit dut (.a(A),.b(B),.p(Result));
  // Clock generation
  reg clk = 0;
  always #((PERIOD)/2) clk = ~clk;
```

```
// Test stimulus
initial begin
  $display("Testbench");

  // Test case 1
  A = 4'b0011;
  B = 4'b0010;
  #PERIOD;
  $display("A = %b, B = %b, Result = %b", A, B, Result);

  // Test case 2
  A = 4'b1010;
  B = 4'b0110;
  #PERIOD;
  $display("A = %b, B = %b, Result = %b", A, B, Result);

  // Test case 3
  A = 4'b1111;
  B = 4'b1010;
  #PERIOD;
  $display("A = %b, B = %b, Result = %b", A, B, Result);

  // Test case 4
  A = 4'b1010;
  B = 4'b0101;
  #PERIOD;
  $display("A = %b, B = %b, Result = %b", A, B, Result);

  // Test case 5
  A = 4'b0000;
  B = 4'b1111;
  #PERIOD;
  $display("A = %b, B = %b, Result = %b", A, B, Result);

  // Test case 6
  A = 4'b1011;
  B = 4'b0100;
  #PERIOD;
  $display("A = %b, B = %b, Result = %b", A, B, Result);

  // Test case 7
  A = 4'b1100;
  B = 4'b0010;
  #PERIOD;
  $display("A = %b, B = %b, Result = %b", A, B, Result);

  // Test case 8
  A = 4'b1011;
  B = 4'b1010;
  #PERIOD;
```

```
    $display("A = %b, B = %b, Result = %b", A, B, Result);

    // Test case 9
    A = 4'b1001;
    B = 4'b1111;
    #PERIOD;
    $display("A = %b, B = %b, Result = %b", A, B, Result);

    // Test case 10
    A = 4'b1000;
    B = 4'b1010;
    #PERIOD;
    $display("A = %b, B = %b, Result = %b", A, B, Result);

    $finish; // End the simulation
  end

  // Clock driver
  always #((PERIOD)/2) clk = ~clk;
endmodule
```

5 Circuit Diagrams

The circuit diagrams provide a good and simplified visualization of what the Verilog code will execute.

5.1 Full Adder

Full Adders include logic gates AND, OR, and XOR.

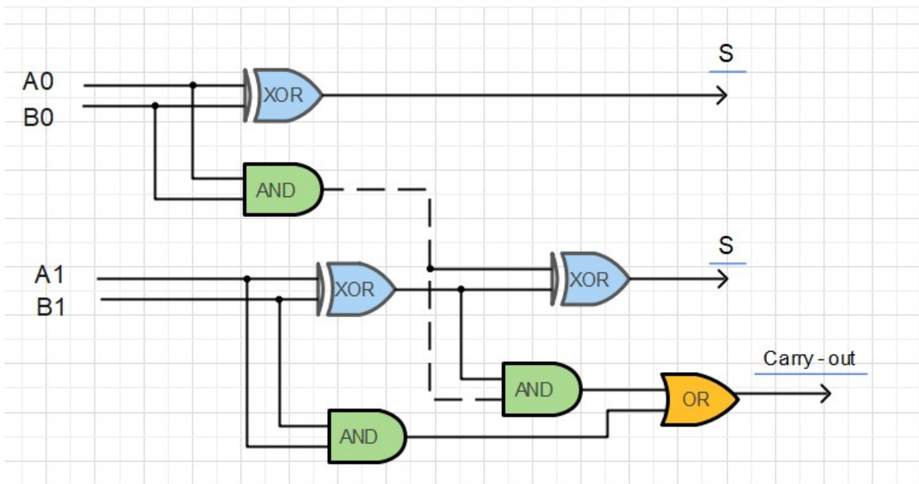

Fig. 1: The Sleepy Cafe created this diagram using Edraw Max free software.

Truth Table	A	B	Cin	Sum	Cout
1	0011	0010	0	0101	0
2	1111	0001	1	0001	1
3	0110	1001	0	1111	0
4	1010	0101	0	1111	0
5	0000	1111	0	1111	0
6	1011	0100	0	1111	0
7	1100	0010	0	1110	0
8	1011	1010	0	0101	1
9	1001	1111	0	1000	1
10	1000	1010	0	0010	1

5.2 Full Subtractor

The subtractor diagram was created by including negation of bits.

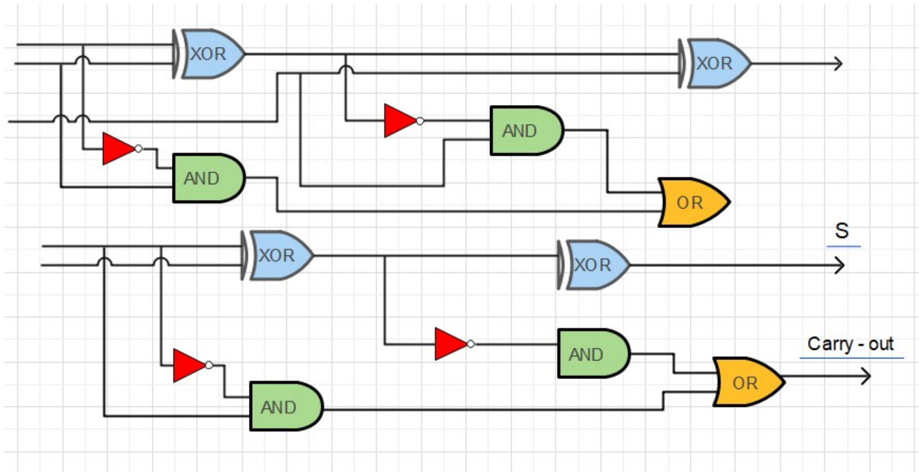

Fig. 2: Caption

Truth Table	A	B	Op	Sum	Carry	V
1	0011	0010	0	0101	0	0
2	0110	0001	1	0101	0	0
3	1111	0001	0	0000	1	0
4	1010	1001	1	0001	0	0
5	1111	1010	1	0101	0	0
6	1010	0101	1	0101	0	1
7	0000	1111	1	0001	1	0
8	1011	0100	1	0111	0	1
9	1100	0010	1	1010	0	0
10	1011	1010	1	0001	0	0
11	1001	1111	1	1010	1	0
12	1000	1010	1	1110	1	0

5.3 Full Multiplier

This multiplier includes the directions of the Sums, Carry bits, and resulting products of A and B inputs. Full Adders include logic gates AND, OR, and XOR.

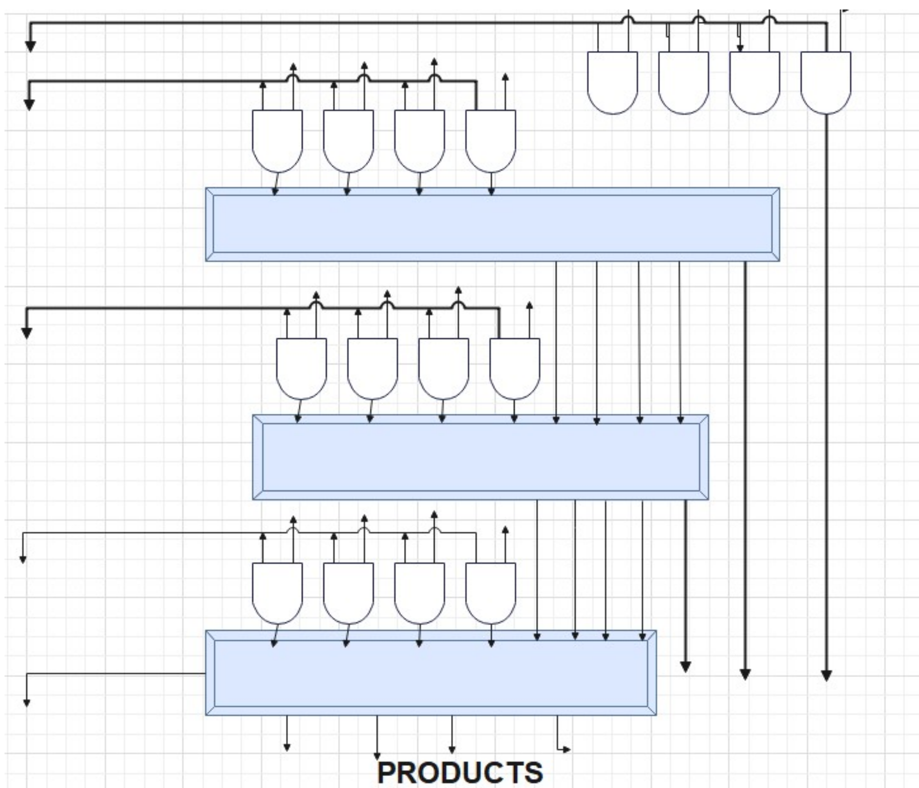

Fig. 3: The Sleepy Cafe created this diagram using Edraw Max free software.

Multi-Test	**A**	**B**	**Result**
1	0011	0010	00000110
2	1010	0110	00111100
3	1111	1010	10010110
4	1010	0101	00110010
5	0000	1111	00000000
6	1011	0100	00101100
7	1100	0010	00011000
8	1011	1010	01101110
9	1001	1111	10000111
10	1000	1010	01010000

6 Test bench and Waveform Responses

The waveforms seen in these screenshots are the visual representations of adder, subtractor, and multiplier components written in Verilog. The images were taken in Fedora Linux and show the output of the Value Change Dump (.vcd) files generated from the code using Verliog and VVP.

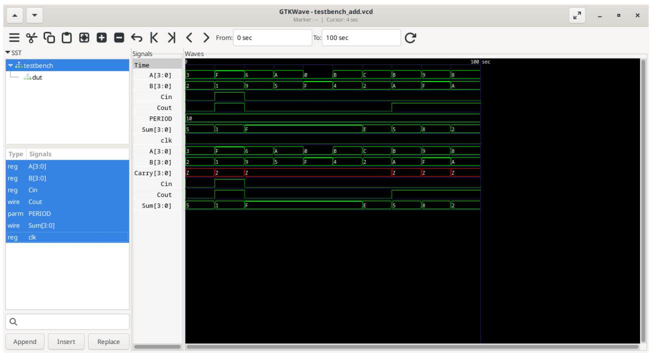

Fig. 4: 4-bit Addder Waveform

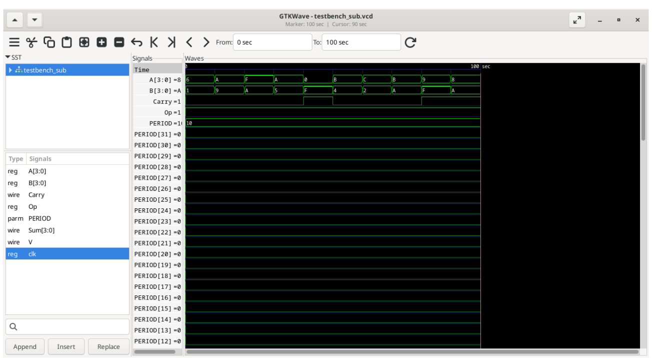

Fig. 5: 4-bit Subtraction Waveform p.1

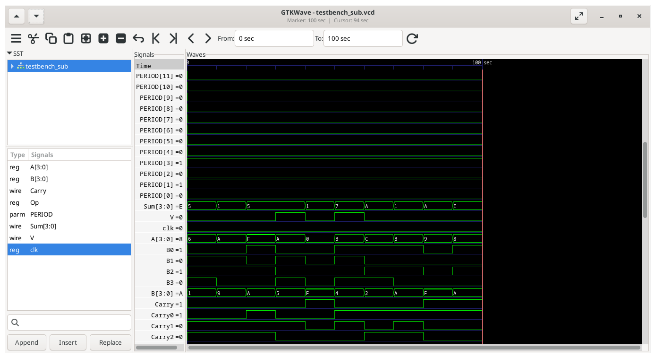

Fig. 6: 4-bit Subtraction Waveform p.2

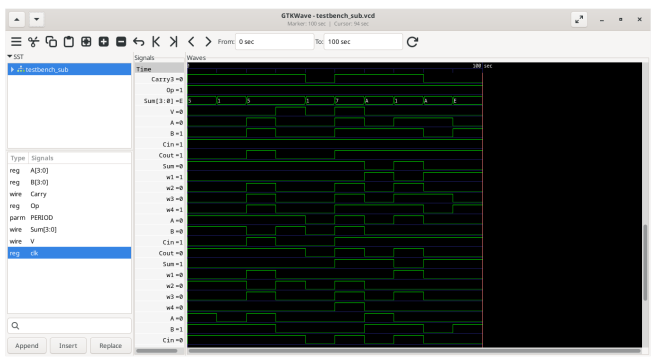

Fig. 7: 4-bit Subtraction Waveform p.3

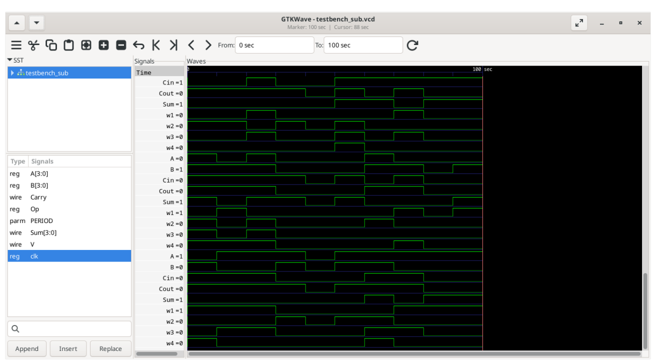

Fig. 8: 4-bit Subtraction Waveform p.4

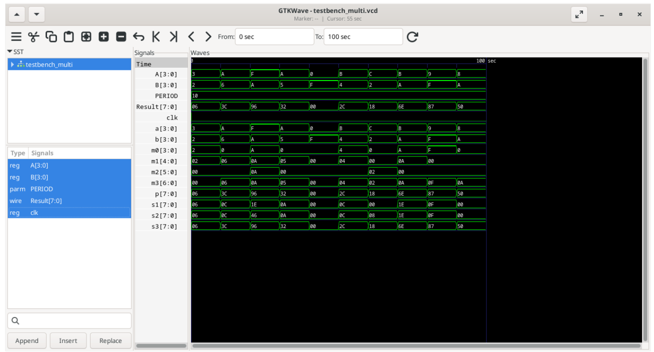

Fig. 9: 4-bit Multiplication Waveform

7 Conclusion

The biggest part of the coding was thoroughly making sure that our modules worked correctly. We also had trouble displaying the results, so at first it was difficult to know if our code was working properly. Having more experience using verilog and LaTEX we were able to find solutions to our issues much faster. Having the circuit diagrams and the wavelengths really helped us verify the consistency of the codes design.

8 References

1. https://web.mit.edu/6.111/www/f2017/handouts/L03$_4$.*pdf*

1. Digital Logic EE 2420 TXSTATE course materials - David Tarnoff,Computer Organization and Design Fundamentals, first revised edition, 2007.