جامعة بيرزيت

BIRZEIT UNIVERSITY

Faculty Of Engineering and Technology

Electrical And Computer Engineering Department

ADVANCED DIGITAL DESIGN (ENCS3310)

COURSE PROJECT
Report
4-bit Arithmetic Unit

Prepared by: Nicola Abu Shaibeh 1190843

Instructor: Dr. Abdellatif Abu-Issa

Date: 22 / 8 / 2022

1. BRIEF INTRODUCTION AND BACKGROUND

In this project, we learn about the concept and design of a 4-bit Arithmetic Unit, that contains 4 4x1 multiplexers and 4 full adders that will be joined to gather to build it, in addition to implanting it utilizing basic logic gates with specific delays, as well as writing structural Verilog code to describe the full circuit and for functional verification with the purpose of testing out the results of all possible values entering the Arithmetic Unit. This Arithmetic circuit performs seven different arithmetic microoperations using 3 selectors (S0, S1, CIN) to choose the operation as shown in the table below. We utilize 4 multiplexers and 4-bit ripple carry adder to build this circuit

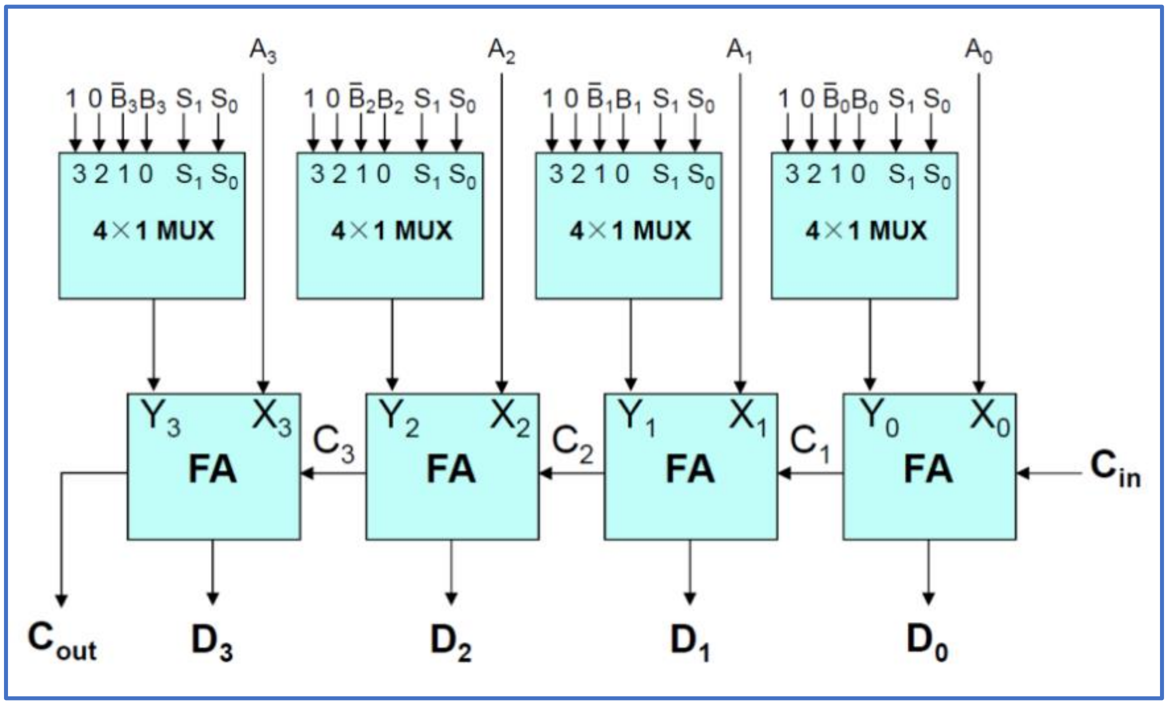


FIGURE 1: 4-BIT ARITHMETIC UNIT

CONTENTS

LIST OF FIGURES

2. DESIGN PHILOSOPHY

• GATES

A logic gate is an idealized model of computation or physical electronic device implementing a Boolean function, a logical operation performed on one or more binary inputs that produce a single binary output. Depending on the context, the term may refer to an ideal logic gate, one that has for instance zero rise time and unlimited fan-out, or it may refer to a non-ideal physical device.

The library element multiplexers and full adders was built with those gates, at given delays.

Gate	Delay
Inverter	3 ns
NAND	5 ns
NOR	5 ns
AND	7 ns
OR	7 ns
XNOR	9 ns
XOR	11 ns

• MULTIPLEXER

Multiplexing is the process of combining one or more signals and transmitting on a single channel. A digital switch also known as a data selector, the multiplexer or MUX is a switch. With several input lines, one output line, the output of the mux will be input b for the full adder

- *In this project, the 4-bit Arithmetic Unit implementation in two stages of complexity, first by using ripple carry adder and second by using carry look ahead on 4-bit groups.*

2.1 STAGE 1

In this stage we will use ripple carry adder, that was constructed from 1-bit full adder which was built from basic gates given. Then we will complete functional verification to demonstrate the ALU.

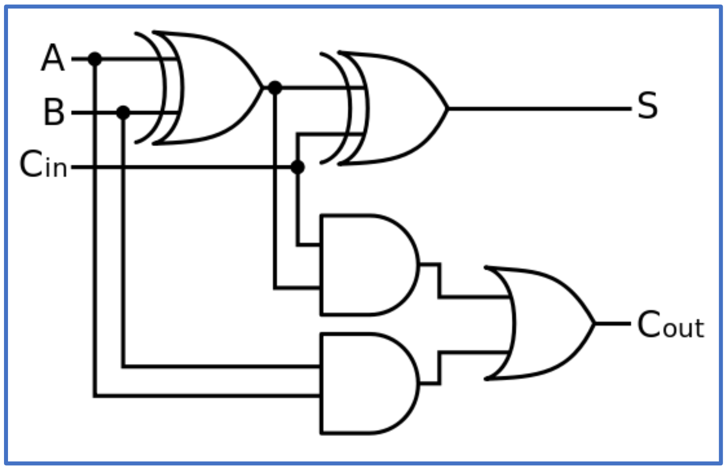


FIGURE 3: 1-BIT FULL ADDER CIRCUIT

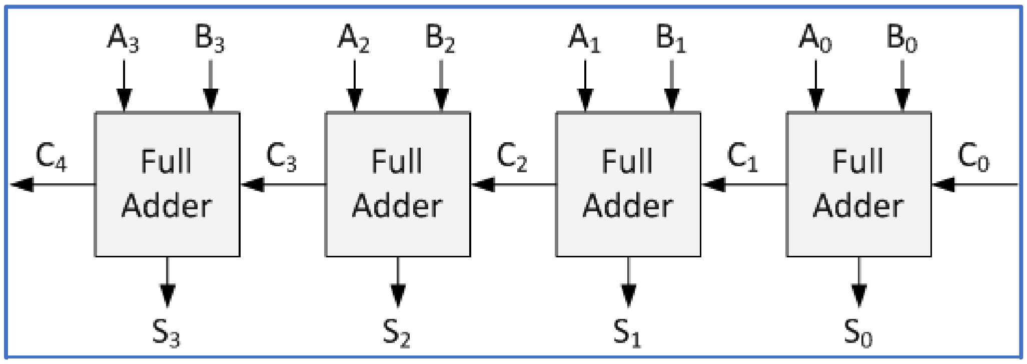
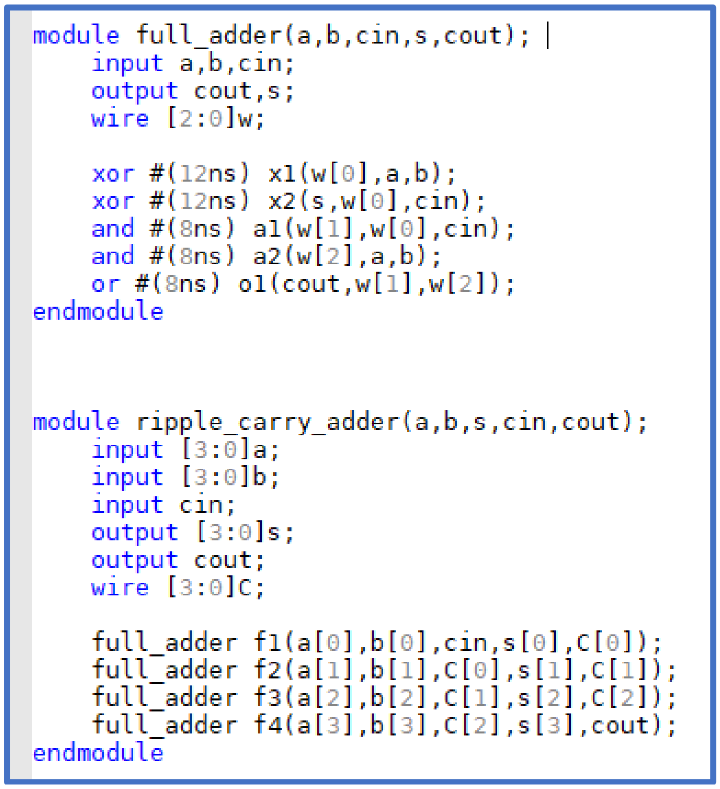


FIGURE 2: RIPPLE CARRY ADDERS

After that we will calculate the maximum latency after the simulation of the circuit to determine the maximum frequency of normal mode clock that can be applied to the flipflop in the circuit, subsequently we should make an error in the circuit and then do a verification that will find the error and make a task that print to console when logical error occur and causes an unexpected output.



```
module full_adder(a,b,cin,s,cout);
    input a,b,cin;
    output cout,s;
    wire [2:0]w;

    xor #(12ns) x1(w[0],a,b);
    xor #(12ns) x2(s,w[0],cin);
    and #(8ns) a1(w[1],w[0],cin);
    and #(8ns) a2(w[2],a,b);
    or #(8ns) o1(cout,w[1],w[2]);
endmodule


module ripple_carry_adder(a,b,s,cin,cout);
    input [3:0]a;
    input [3:0]b;
    input cin;
    output [3:0]s;
    output cout;
    wire [3:0]C;

    full_adder f1(a[0],b[0],cin,s[0],C[0]);
    full_adder f2(a[1],b[1],C[0],s[1],C[1]);
    full_adder f3(a[2],b[2],C[1],s[2],C[2]);
    full_adder f4(a[3],b[3],C[2],s[3],cout);
endmodule
```

FIGURE 4: RIPPLE CARRY ADDER

2.2 STAGE 2

In this stage we will use carry look ahead, that accelerates addition on 4-bit groups. Then we will complete functional verification to demonstrate the ALU.

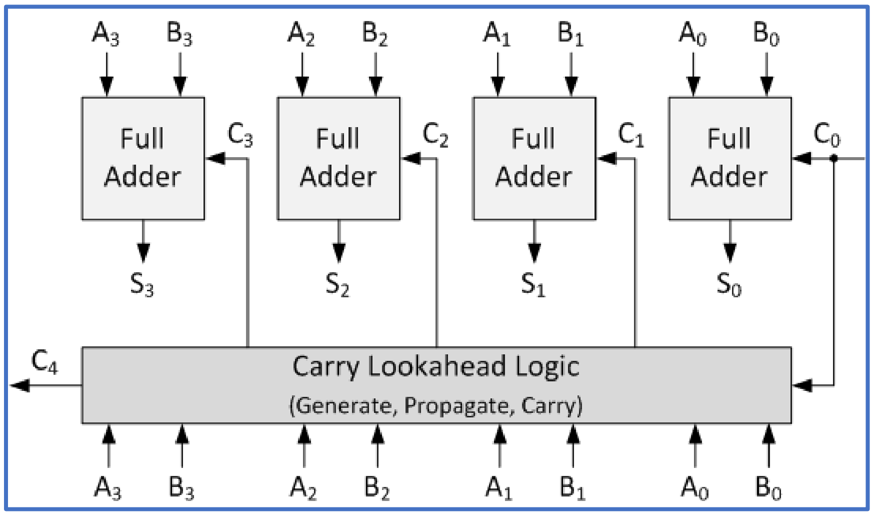
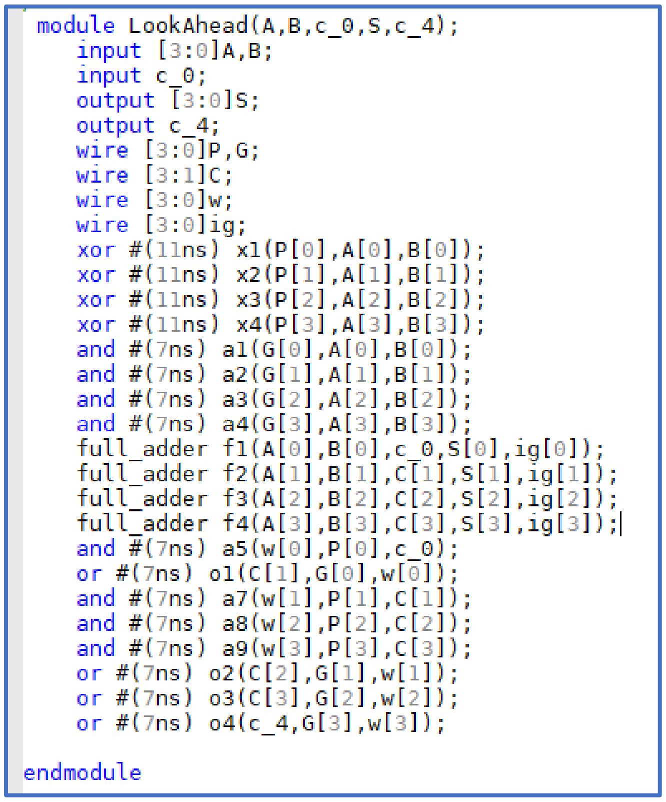


FIGURE 5: PARTIAL FULL ADDER

In normal ripple adders, the carry of the output of each full adder is given as a carry input to the next higher-order state. Henceforth, these adders it is not possible to produce carry and sum outputs of any state without a carry input is available for that state. Subsequently we will calculate the maximum latency after the simulation of the circuit to determine the maximum frequency of normal mode clock that can be applied to the flipflop in the circuit, subsequently we should make an error in the circuit and then do a verification that will find the error and make a task that print to console when logical error occur and causes an unexpected output.

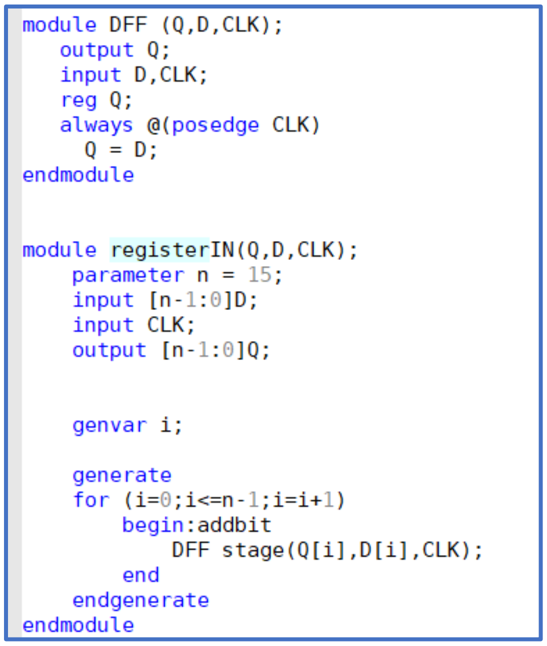


```
 module LookAhead(A,B,c_0,S,c_4);
    input [3:0]A,B;
    input c_0;
    output [3:0]S;
    output c_4;
    wire [3:0]P,G;
    wire [3:1]C;
    wire [3:0]w;
    wire [3:0]ig;
    xor #(11ns) x1(P[0],A[0],B[0]);
    xor #(11ns) x2(P[1],A[1],B[1]);
    xor #(11ns) x3(P[2],A[2],B[2]);
    xor #(11ns) x4(P[3],A[3],B[3]);
    and #(7ns) a1(G[0],A[0],B[0]);
    and #(7ns) a2(G[1],A[1],B[1]);
    and #(7ns) a3(G[2],A[2],B[2]);
    and #(7ns) a4(G[3],A[3],B[3]);
    full_adder f1(A[0],B[0],c_0,S[0],ig[0]);
    full_adder f2(A[1],B[1],C[1],S[1],ig[1]);
    full_adder f3(A[2],B[2],C[2],S[2],ig[2]);
    full_adder f4(A[3],B[3],C[3],S[3],ig[3]);
    and #(7ns) a5(w[0],P[0],c_0);
    or #(7ns) o1(C[1],G[0],w[0]);
    and #(7ns) a7(w[1],P[1],C[1]);
    and #(7ns) a8(w[2],P[2],C[2]);
    and #(7ns) a9(w[3],P[3],C[3]);
    or #(7ns) o2(C[2],G[1],w[1]);
    or #(7ns) o3(C[3],G[2],w[2]);
    or #(7ns) o4(c_4,G[3],w[3]);

endmodule
```

FIGURE 6: LOOK AHEAD CODE

2.3 REGISTERS

We build the registers utilizing d-flipflop to synchronize the input of the ALU, as well as synchronizing the output.



```
module DFF (Q,D,CLK);
   output Q;
   input D,CLK;
   reg Q;
   always @(posedge CLK)
     Q = D;
endmodule


module registerIN(Q,D,CLK);
    parameter n = 15;
    input [n-1:0]D;
    input CLK;
    output [n-1:0]Q;


    genvar i;

    generate
    for (i=0;i<=n-1;i=i+1)
        begin:addbit
            DFF stage(Q[i],D[i],CLK);
        end
    endgenerate
endmodule
```

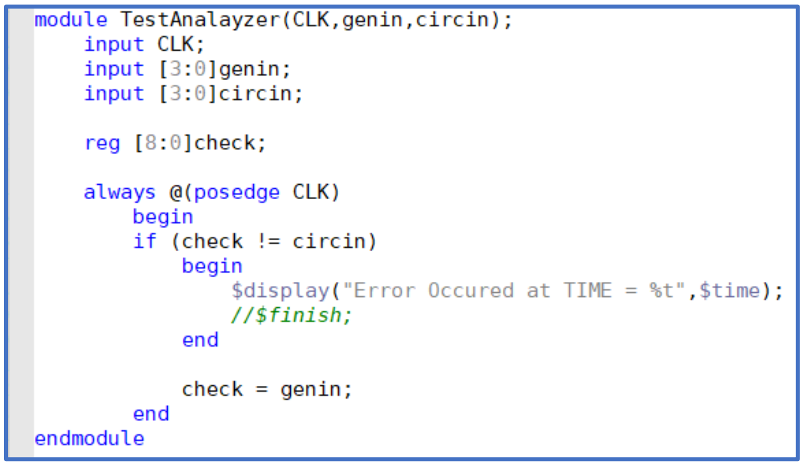
FIGURE 7: REGISTER

2.4 BUILT-IN SELF-TEST

Until now we have assumed that testing of the logic circuits in each stage is done by externally applying the test inputs and comparing the results with the expected behavior of the circuit. This requires connecting external equipment to the circuit under test.

• TEST ANALYZER

Test analyzer ensures the output of the test generator same as the output of the circuit and raise an error if the output in not the same.

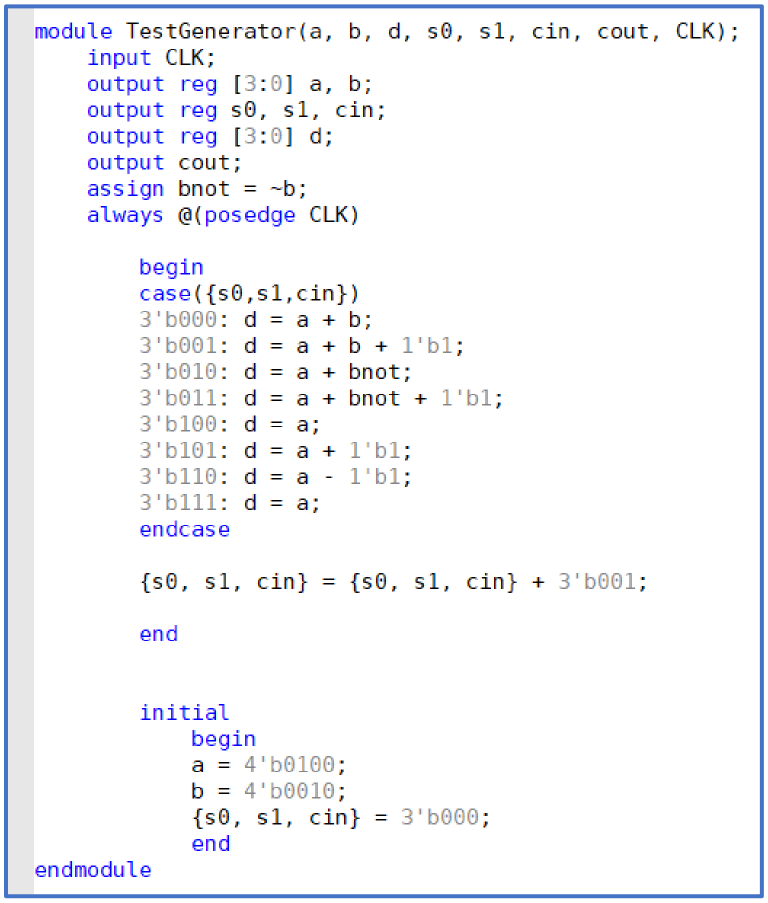


```
module TestAnalayzer(CLK,genin,circin);
    input CLK;
    input [3:0]genin;
    input [3:0]circin;

    reg [8:0]check;

    always @(posedge CLK)
        begin
        if (check != circin)
            begin
                $display("Error Occured at TIME = %t",$time);
                //$finish;
            end

            check = genin;
        end
endmodule
```

FIGURE 8: TEST ANALYZER CODE

• TEST GENERATOR

This generator builds on behavioral logic of the ALU micro-operations, to test the all operation on A and B, also it sends output to test analyzer and to the main circuit to compare them.



```
module TestGenerator(a, b, d, s0, s1, cin, cout, CLK);
    input CLK;
    output reg [3:0] a, b;
    output reg s0, s1, cin;
    output reg [3:0] d;
    output cout;
    assign bnot = ~b;
    always @(posedge CLK)

        begin
        case({s0,s1,cin})
        3'b000: d = a + b;
        3'b001: d = a + b + 1'b1;
        3'b010: d = a + bnot;
        3'b011: d = a + bnot + 1'b1;
        3'b100: d = a;
        3'b101: d = a + 1'b1;
        3'b110: d = a - 1'b1;
        3'b111: d = a;
        endcase

        {s0, s1, cin} = {s0, s1, cin} + 3'b001;

        end


        initial
            begin
            a = 4'b0100;
            b = 4'b0010;
            {s0, s1, cin} = 3'b000;
            end
endmodule
```

FIGURE 9: TEST GENERATOR

3. RESULTS

- In this section, simulation output of the Arithmetic Unit and the maximum delay are provided.

3.1. STAGE 1

After testing the ALU utilizing the ripple adder, I calculate the maximum latency possible time to prevent the delay problems, its (114) nanoseconds, thus the maximum frequency of normal-mode clock is 8.77MHz.

Also, as we can see from figure below, I made A and B as a constant A=4 and B=2, and changing the selectors (s0, s1, cin) to test all the microoperations on the ALU.As we can see all the result are correct and stable as needed.

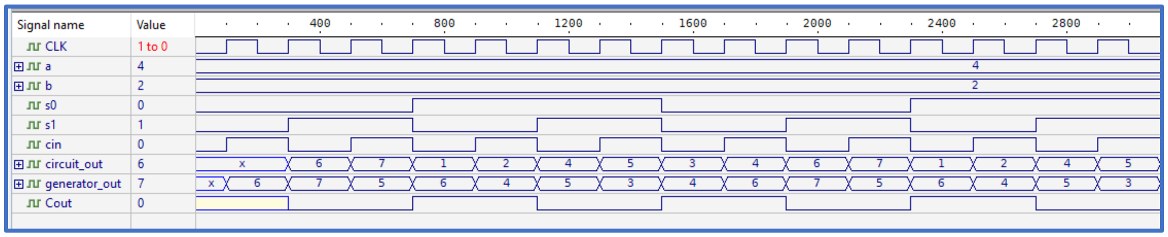
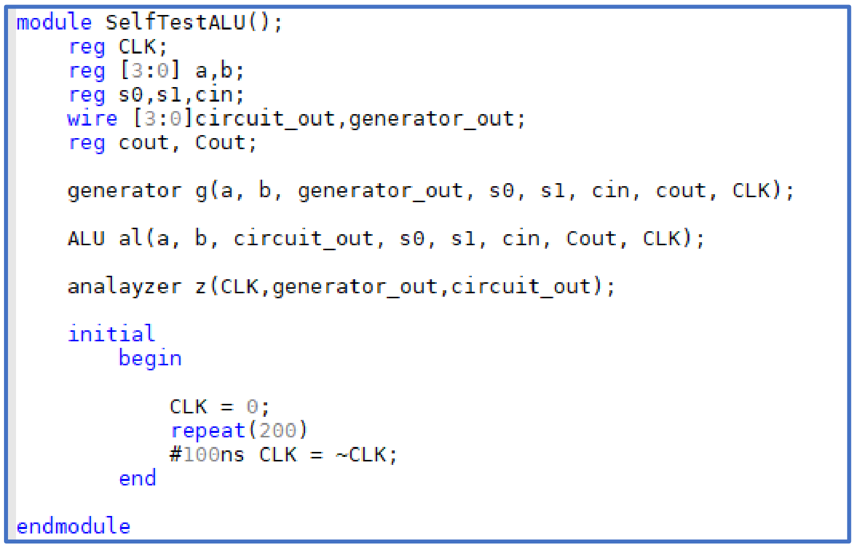


FIGURE 10: RESULT STAGE 1 WAVE FORM



```
module SelfTestALU();
    reg CLK;
    reg [3:0] a,b;
    reg s0,s1,cin;
    wire [3:0]circuit_out,generator_out;
    reg cout, Cout;

    generator g(a, b, generator_out, s0, s1, cin, cout, CLK);

    ALU al(a, b, circuit_out, s0, s1, cin, Cout, CLK);

    analayzer z(CLK,generator_out,circuit_out);

    initial
        begin

            CLK = 0;
            repeat(200)
            #100ns CLK = ~CLK;
        end

endmodule
```

FIGURE 11: RESULT STAGE 1 SELF-TEST ALU CODE

RISING AN ERROR

As we can see below in the wave form, by introducing an error the result will be wrong, and the system will discover it and rise an error.

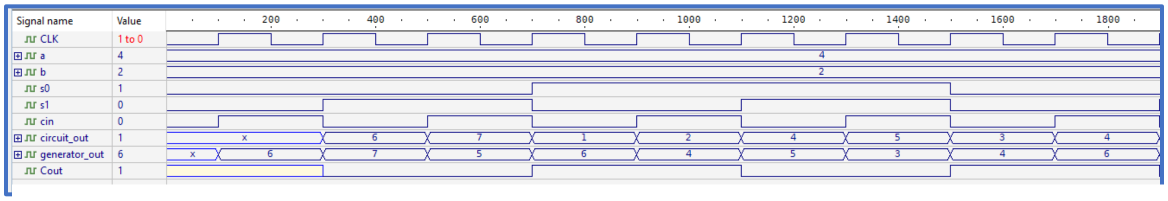
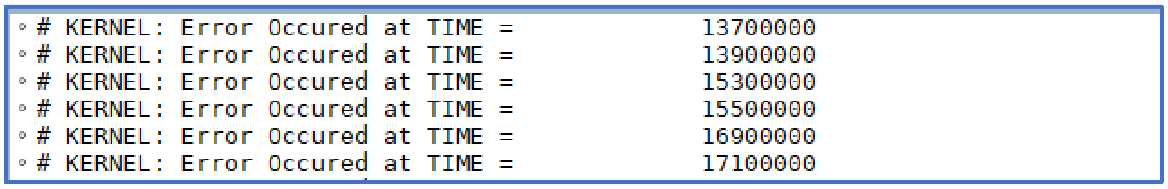


FIGURE 12: RESULT STAGE 1 ERROR WAVE



```
# KERNEL: Error Occured at TIME =          13700000
# KERNEL: Error Occured at TIME =          13900000
# KERNEL: Error Occured at TIME =          15300000
# KERNEL: Error Occured at TIME =          15500000
# KERNEL: Error Occured at TIME =          16900000
# KERNEL: Error Occured at TIME =          17100000
```

FIGURE 13: RESULT STAGE 1 ERROR OCCURED

3.2. STAGE 2

After testing the ALU utilizing the accelerator addition by using carry look ahead on 4-bit groups. I calculate the maximum latency possible time to prevent the delay problems, its (164) nanoseconds, thus the maximum frequency of normal-mode clock is 6.09MHz.

Also, as we can see from figure below, I made A and B as a constant A=4 and B=2, and changing the selectors (s0, s1, cin) to test all the microoperations on the ALU.

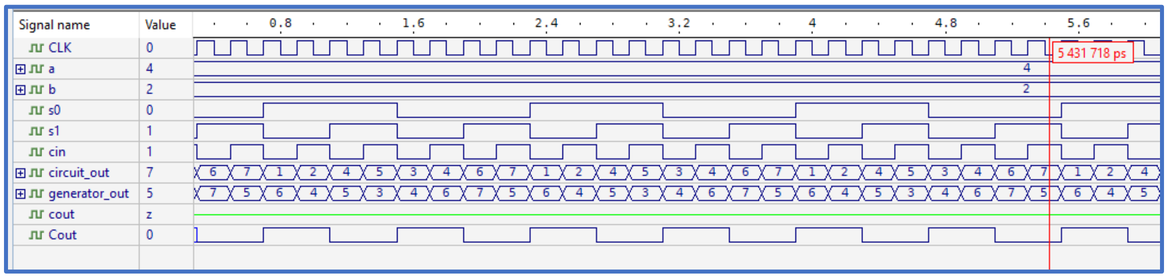
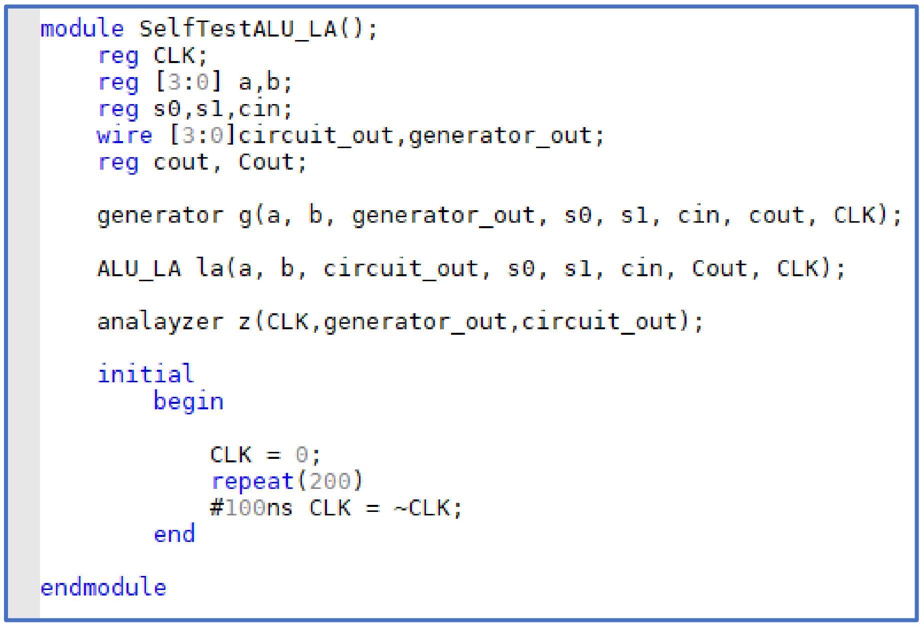


FIGURE 14: RESULT STAGE 2 WAVE FORM

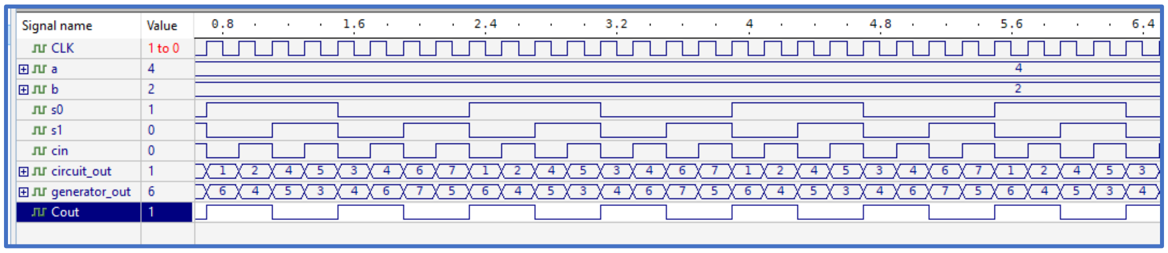


```verilog
module SelfTestALU_LA();
    reg CLK;
    reg [3:0] a,b;
    reg s0,s1,cin;
    wire [3:0]circuit_out,generator_out;
    reg cout, Cout;

    generator g(a, b, generator_out, s0, s1, cin, cout, CLK);

    ALU_LA la(a, b, circuit_out, s0, s1, cin, Cout, CLK);

    analayzer z(CLK,generator_out,circuit_out);

    initial
        begin

            CLK = 0;
            repeat(200)
            #100ns CLK = ~CLK;
        end

endmodule
```

FIGURE 15: RESULT STAGE 2 SELF-TEST ALU CODE

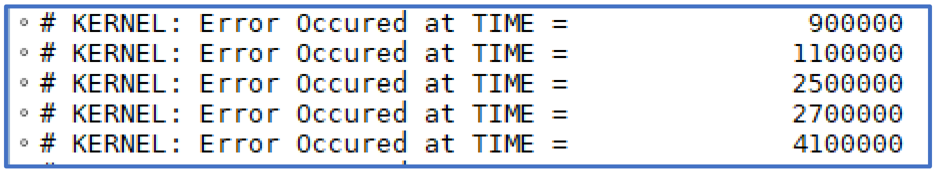
RISING AN ERROR

As we can see below in the wave form, by introducing an error the result will be wrong, and the system will discover it and rise an error.



Signal name	Value
CLK	1 to 0
a	4
b	2
s0	1
s1	0
cin	0
circuit_out	1
generator_out	6
Cout	1

FIGURE 16: RESULT STAGE 2 ERROR WAVE



```
# KERNEL: Error Occured at TIME =          900000
# KERNEL: Error Occured at TIME =         1100000
# KERNEL: Error Occured at TIME =         2500000
# KERNEL: Error Occured at TIME =         2700000
# KERNEL: Error Occured at TIME =         4100000
```

FIGURE 17: RESULT STAGE 2 ERROR OCCURED

4. CONCLUSION AND FUTURE WORKS

In conclusion we learned how to structurally make a 4-bit Arithmetic Unit that can implement deferent microoperation and how to check and verify if our work is done accurately.

The two stages presented that there is always more than a way to implement circuit that gave the same result, yet each implementation has its own advantages and disadvantages, in time or the number of needed gates needed, and some can be considered better than others judging based on results.

This project helped me develop new skills in Verilog hardware description language that make me implement very complex circuits than before, in addition to taking advantage of its capability to run in parallel to make things go faster.

APPENDIX

CODE

```
`timescale 1ns / 1ps
/*
******************************************************************************
*************************************************** */


/*
                    Nicola Abu Shaibeh 1190843
                    4 bit Arithmetic Unit
                    12/08/2022
                    The task is to design an Arithmetic Unit

*/

/*
******************************************************************************
*************************************************** */

module mux (out, a, b, c, d, s0, s1);
          output out;
          input a, b, c, d, s0, s1;
          wire sobar, s1bar, T1, T2, T3, T4;



          not #(3ns) (s0bar, s0), (s1bar, s1);
          and #(7ns) (T1, a, s0bar, s1bar), (T2, b, s0bar, s1),(T3, c, s0, s1bar), (T4, d, s0, s1);
          or #(7ns) (out, T1, T2, T3, T4);

endmodule

/*
******************************************************************************
*************************************************** */
```

```
module full_adder(a,b,cin,s,cout); // Full adder module for ripple carry 4 bit adder
        input a,b,cin;
        output cout,s;
        wire [2:0]w;

        xor #(12ns) x1(w[0],a,b);
        xor #(12ns) x2(s,w[0],cin);
        and #(8ns) a1(w[1],w[0],cin);
        and #(8ns) a2(w[2],a,b);
        or #(8ns) o1(cout,w[1],w[2]);
endmodule



module ripple_carry_adder(a,b,s,cin,cout);
        input [3:0]a;
        input [3:0]b;
        input cin;
        output [3:0]s;
        output cout;
        wire [3:0]C;

        full_adder f1(a[0],b[0],cin,s[0],C[0]);
        full_adder f2(a[1],b[1],C[0],s[1],C[1]);
        full_adder f3(a[2],b[2],C[1],s[2],C[2]);
        full_adder f4(a[3],b[3],C[2],s[3],cout);
endmodule

/*
********************************************************************************
**************************************************** */
```

```
module DFF (Q,D,CLK);
  output Q;
  input D,CLK;
  reg Q;
  always @(posedge CLK)
    Q = D;
endmodule



module registerIN(Q,D,CLK);
         parameter n = 15;
         input [n-1:0]D;
         input CLK;
         output [n-1:0]Q;



         genvar i;


         generate
         for (i=0;i<=n-1;i=i+1)
                  begin:addbit
                           DFF stage(Q[i],D[i],CLK);
                  end
         endgenerate
endmodule

module registerOUT(Q,D,CLK);
         parameter n = 5;
         input [n-1:0]D;
         input CLK;
         output [n-1:0]Q;
```

```
        genvar i;

        generate
        for (i=0;i<=n-1;i=i+1)
                begin:addbit
                        DFF stage(Q[i],D[i],CLK);
                end
        endgenerate
endmodule



/*
****************************************************************************
************************************************** */

module ALU(a, b, d, s0, s1, cin, cout, CLK);
        input [3:0] a, b;
        input s0, s1, cin, CLK;
        output [3:0] d;
        output cout;
        wire [3:0] A, B, Bnot, D;
        wire S0, S1, CIN, COUT;
        wire [3:0] c;
        reg [3:0] bnot;
        assign bnot = ~b;
        assign zero = 1'b0;
        assign one = 1'b1;

        // 0-3a 4-7b, 8-11!b 12s0 13s1 14cin
        registerIN ri({A,B,Bnot,S0,S1,CIN},{a, b, bnot, s0, s1, cin},CLK);

        mux m0(c[0], B[0], Bnot[0], zero, one, S0, S1);
```

```
            mux m1(c[1], B[1], Bnot[1], zero, one, S0, S1);
            mux m2(c[2], B[2], Bnot[2], zero, one, S0, S1);
            mux m3(c[3], B[3], Bnot[3], zero, one, S0, S1);
            ripple_carry_adder fr(A[3:0],c[3:0],D[3:0],CIN,COUT);

            registerOUT #(.n(18)) ro({cout, d},{COUT, D},CLK);

endmodule




module Stimulus() ;
            reg [3:0] a,b;
            reg s0,s1,cin, CLK;
            wire [3:0] d;
            wire cout;
            ALU my(.a(a),.b(b),.d(d),.s0(s0),.s1(s1),.cin(cin), .cout(cout), .CLK(CLK));

            initial
                        begin
                         CLK=0;
                        repeat(2000)
                        #100ns CLK = ~CLK;
                        $finish;

            end

            initial
            begin
                        a = 4'b1010;
                        b = 4'b0101;

                        {s0, s1, cin} = 3'b000;
```

```
                    #100ns;
                    repeat(7)
                    #200ns {s0, s1, cin} = {s0, s1, cin} + 3'b001;



                    $display("s0 = %b , s1 = %b , cin = %b \n",s0,s1,cin);
                    $display("ANSWER = %b %b %b %b\n",d[3],d[2],d[1],d[0]);

        end
endmodule

/*
*******************************************************************************
****************************************************** */
 module LookAhead(A,B,c_0,S,c_4);
        input [3:0]A,B;
        input c_0;
        output [3:0]S;
        output c_4;
        wire [3:0]P,G;
        wire [3:1]C;
        wire [3:0]w;
        wire [3:0]ig;
        //propagation and generator implementation
        xor #(11ns) x1(P[0],A[0],B[0]);
        xor #(11ns) x2(P[1],A[1],B[1]);
        xor #(11ns) x3(P[2],A[2],B[2]);
        xor #(11ns) x4(P[3],A[3],B[3]);

        and #(7ns) a1(G[0],A[0],B[0]);
        and #(7ns) a2(G[1],A[1],B[1]);
        and #(7ns) a3(G[2],A[2],B[2]);
        and #(7ns) a4(G[3],A[3],B[3]);
        //fulladder to get sum with carry out ignorant
        full_adder f1(A[0],B[0],c_0,S[0],ig[0]);
        full_adder f2(A[1],B[1],C[1],S[1],ig[1]);
        full_adder f3(A[2],B[2],C[2],S[2],ig[2]);
        full_adder f4(A[3],B[3],C[3],S[3],ig[3]);

        //carries implementation
        and #(7ns) a5(w[0],P[0],c_0);
        or #(7ns) o1(C[1],G[0],w[0]);

        and #(7ns) a7(w[1],P[1],C[1]);
        and #(7ns) a8(w[2],P[2],C[2]);
        and #(7ns) a9(w[3],P[3],C[3]);

        or #(7ns) o2(C[2],G[1],w[1]);
        or #(7ns) o3(C[3],G[2],w[2]);
        or #(7ns) o4(c_4,G[3],w[3]);

endmodule

module ALU_LA(a, b, d, s0, s1, cin, cout, CLK);
        input [3:0] a, b;
        input s0, s1, cin, CLK;
        output [3:0] d;
        output cout;
        wire [3:0] A, B, Bnot, D;
        wire S0, S1, CIN, COUT;
        wire [3:0] c;
        reg [3:0] bnot;
        assign bnot = ~b;
        assign zero = 1'b0;
        assign one = 1'b1;
        // 0-3a 4-7b, 8-11!b 12s0 13s1 14cin
        registerIN ri({A,B,Bnot,S0,S1,CIN},{a, b, bnot, s0, s1, cin},CLK);

        mux m0(c[0], B[0], Bnot[0], zero, one, S0, S1);
        mux m1(c[1], B[1], Bnot[1], zero, one, S0, S1);
        mux m2(c[2], B[2], Bnot[2], zero, one, S0, S1);
        mux m3(c[3], B[3], Bnot[3], zero, one, S0, S1);

        LookAhead la(A[3:0],c[3:0],CIN,D[3:0],COUT);
        registerOUT #(.n(18)) ro({cout, d},{COUT, D},CLK);

endmodule

module Stimulus_LA() ;
        reg [3:0] a,b;
        reg s0,s1,cin, CLK;
        wire [3:0] d;
        wire cout;
        ALULA myla(.a(a),.b(b),.d(d),.s0(s0),.s1(s1),.cin(cin), .cout(cout), .CLK(CLK));

        initial
                begin
                 CLK=0;
                repeat(2000)
                #100ns CLK = ~CLK;
                $finish;

        end

        initial
        begin
                a = 4'b1010;
                b = 4'b0101;
                {s0, s1, cin} = 3'b000;
                #100ns;
                repeat(7)
                #200ns {s0, s1, cin} = {s0, s1, cin} + 3'b001;



                $display("s0 = %b , s1 = %b , cin = %b \n",s0,s1,cin);
                $display("ANSWER = %b %b %b %b\n",d[3],d[2],d[1],d[0]);

        end
endmodule




/*
*******************************************************************************
************************************************** */


module TestGenerator(a, b, d, s0, s1, cin, cout, CLK);
        input CLK;
        output reg [3:0] a, b;
        output reg s0, s1, cin;
        output reg [3:0] d;
        output cout;
        assign bnot = ~b;
        always @(posedge CLK)

                begin
                case({s0,s1,cin})
                3'b000: d = a + b;
                3'b001: d = a + b + 1'b1;
                3'b010: d = a + bnot;
                3'b011: d = a + bnot + 1'b1;
```

```
                    3'b100: d = a;
                    3'b101: d = a + 1'b1;
                    3'b110: d = a - 1'b1;
                    3'b111: d = a;
                    endcase

                    {s0, s1, cin} = {s0, s1, cin} + 3'b001;

                    end



                    initial
                              begin
                              a = 4'b0100;
                              b = 4'b0010;
                              {s0, s1, cin} = 3'b000;
                              end
endmodule



module TestAnalayzer(CLK,genin,circin);
          input CLK;
          input [3:0]genin;
          input [3:0]circin;

          reg [8:0]check;

          always @(posedge CLK)
                    begin
                    if (check != circin)
                              begin
                                        $display("Error Occured at TIME = %t",$time);
                                        //$finish;
                              end

                              check = genin;
                    end
endmodule



module SelfTestALU();
          reg CLK;
          reg [3:0] a,b;
          reg s0,s1,cin;
          wire [3:0]circuit_out,generator_out;
          reg cout, Cout;

          generator g(a, b, generator_out, s0, s1, cin, cout, CLK);

          ALU al(a, b, circuit_out, s0, s1, cin, Cout, CLK);

          analayzer z(CLK,generator_out,circuit_out);

          initial
                    begin

                              CLK = 0;
                              repeat(200)
                              #100ns CLK = ~CLK;
                    end

endmodule

module SelfTestALU_LA();
          reg CLK;
          reg [3:0] a,b;
          reg s0,s1,cin;
          wire [3:0]circuit_out,generator_out;
          reg cout, Cout;

          generator g(a, b, generator_out, s0, s1, cin, cout, CLK);

          ALU_LA la(a, b, circuit_out, s0, s1, cin, Cout, CLK);

          analayzer z(CLK,generator_out,circuit_out);

          initial
                    begin

                              CLK = 0;
                              repeat(200)
                              #100ns CLK = ~CLK;
                    end

endmodule

/*
*****************************************************************************************
********************************************************** */
```