Faculty of Engineering & Technology
Electrical & Computer Engineering Department

ENCS3310

Project Report

Prepared by:

Nidal Zabade 1200153

Instructor: Dr. Abdallatif Abuissa

Section: 1

Date: 16/8/2022

Abstract

The aim of this project to design an Arithmetic Unit using Verilog in Active HDL then write complete code for functional verification.

To Use Different types of adders: Ripple Adder, Look-ahead Adder and find the differences using them in the Arithmetic Unit.

Table of Content

Table of Tables & Figures

Table of Figures

Table of Tables

Theory

Multiplexer

The multiplexer or MUX is a digital switch, also called as data selector. It is a Combinational Logic Circuit with more than one input line, one output line and more than one select line. It accepts the binary information from several input lines or sources and depending on the set of select lines, a particular input line is routed onto a single output line.[1]

4x1 Multiplexer

A 4-to-1 multiplexer consists four data input lines as D0 to D3, two select lines as S0 and S1 and a single output line Y. The select lines S0 and S1 select one of the four input lines to connect the output line. The figure below shows the block diagram of a 4-to-1 multiplexer in which, the multiplexer decodes the input through select line.[1]

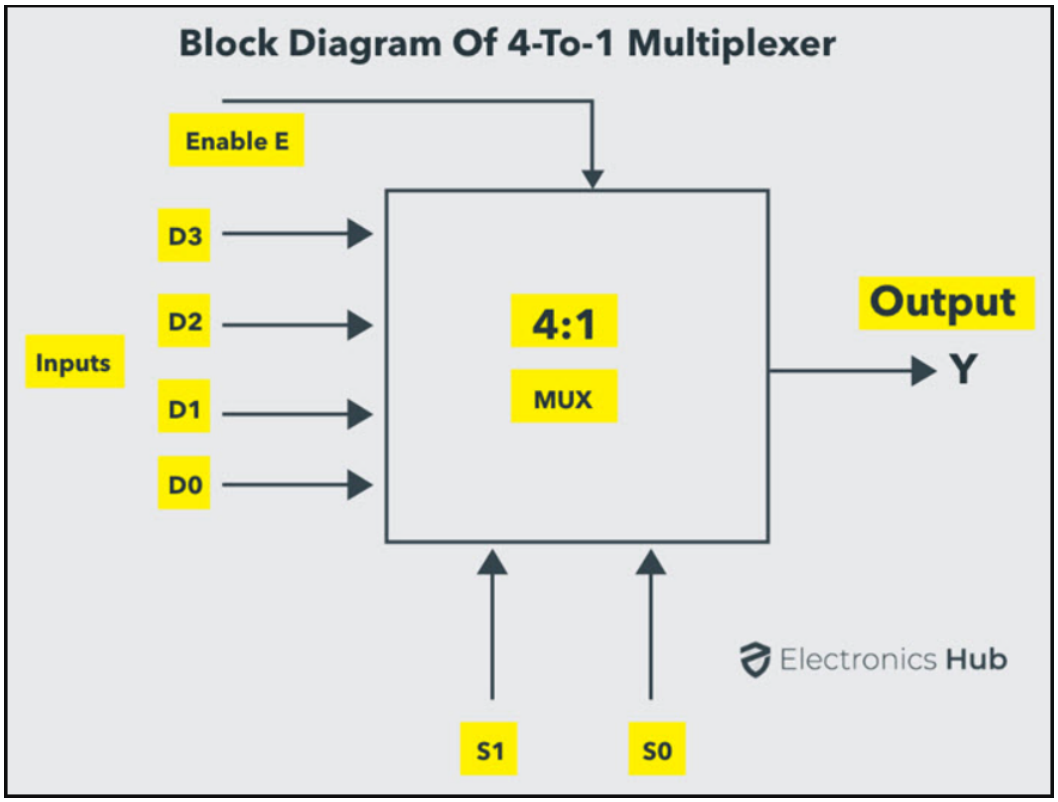

Figure 1.1: 4-to-1 multiplexer

Ripple carry adder

A ripple carry adder is a digital circuit that produces the arithmetic sum of two binary numbers. It can be constructed with full adders connected in cascaded, with the carry output from each full adder connected to the carry input of the next full adder in the chain. Figure below shows the interconnection of four full adder (FA) circuits to provide a 4-bit ripple carry adder. Notice from

the figure that the input is from the right side because the first cell traditionally represents the least significant bit (LSB). Bits a0 and b0 in the figure represent the least significant bits of the numbers to be added. The sum output is represented by the bits s0-s3.[2]

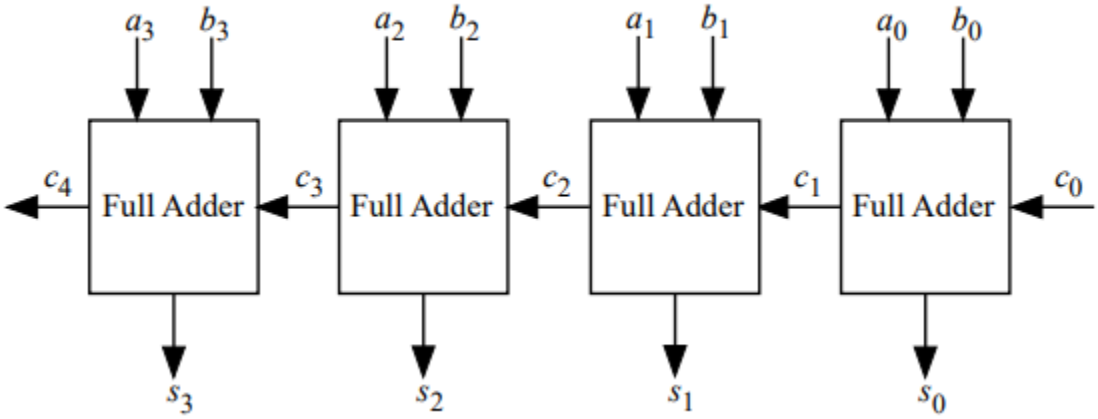

Figure 1.2: Ripple carry adder

Carry look-ahead adder (CLA)

A carry look-ahead adder reduces the propagation delay by introducing hardware that is more complex. In this design, the ripple carry design is suitably transformed such that the carry logic over fixed groups of bits of the adder is reduced to two-level logic.[3]

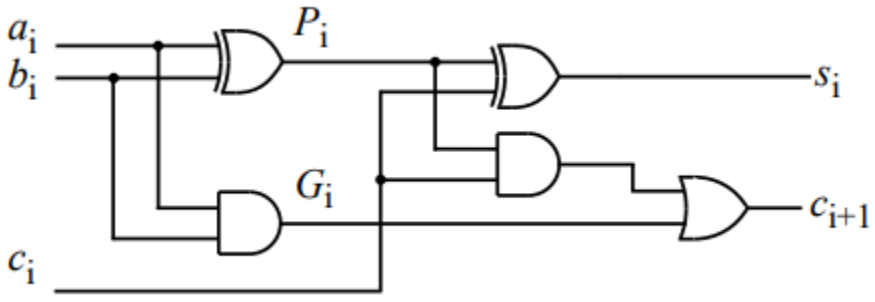

Figure 1.3: carry generate and carry propagate terms

The CLA Built in two important equations which:

$c_{i+1} = G_i + P_i \cdot c_i$

$s_i = P_i \wedge c_i$

Where:

$G_i = a_i \cdot b_i$

$P_i = a_i \wedge b_i$

G_i and P_i are called the carry generate and carry propagate terms, respectively. Notice that the generate and propagate terms only depend on the input bits and thus will be valid after one and 4 two gate delay, respectively. If one uses the above expression to calculate the carry signals, one does not need to wait for the carry to ripple through all the previous stages to find its proper value.

For 4-bit Adder:

$$
\begin{aligned}
c_1 &= G_0 + P_0.c_0 \\
c_2 &= G_1 + P_1.G_0 + P_1.P_0.c_0 \\
c_3 &= G_2 + P_2.G_1 + P_2.P_1.G_0 + P_2.P_1.P_0.c_0 \\
c_4 &= G_3 + P_3.G_2 + P_3.P_2.G_1 + P_3.P_2.P_1.G_0 + P_3.P_2.P_1.P_0.c_0
\end{aligned}
$$

Therefore, the Block Diagram should be like

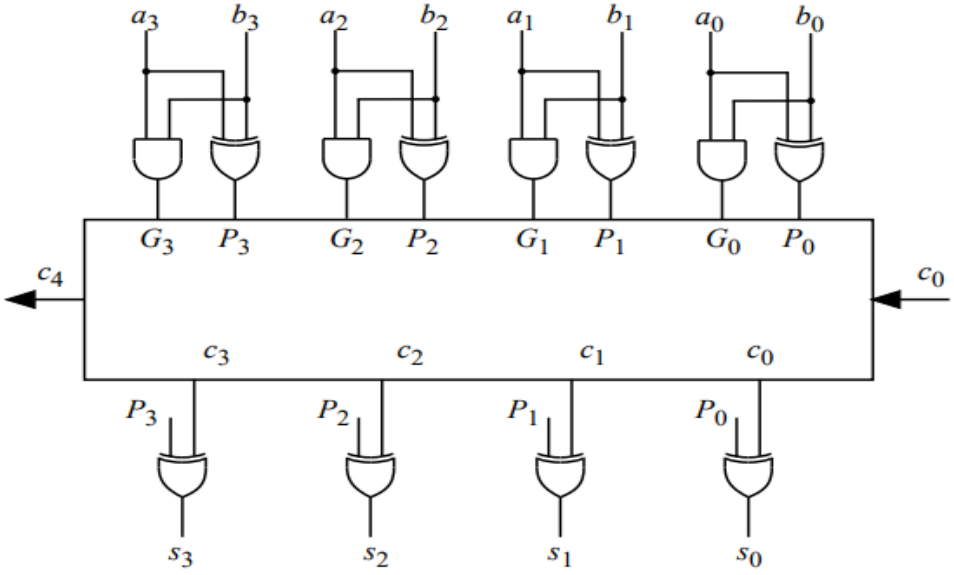

Figure 1.4: carry look-ahead adder

Procedure

The Arithmetic unit works as shown in the following table:

	Select			Input Y	Output D = A + Y + Cin	Microoperations
	S1	S0	Cin			
1	0	0	0	B	D = A + B	Add
2	0	0	1	B	D = A + B + 1	Add with carry
3	0	1	0	B^-	D = A + B^-	Sub. With borrow
4	0	1	1	B^-	D = A + B^- + 1	Sub
5	1	0	0	0	D = A	Transfer A
6	1	0	1	0	D = A + 1	Increment
7	1	1	0	1	D = A - 1	Decrement
8	1	1	1	1	D = A	Transfer A

Table 2.1: Arithmetic unit operations

The Arithmetic Unit is to be built structurally from a library of gates, which contains the following devices:

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

Table 2.2: Gates Delays

Build 4x1 Multiplexer

4x1 multiplexer module takes 4-bit input (B) and 2-bit input Selection. In addition, gives one output (Out). The implementation as the following:

```
module Mux4to1(B, Selection , Out);
    input [3:0]B;
    input [1:0] Selection;
    output Out;
    wire [3:0] f;
    wire [1:0] Selection_Prime;
    not #3ns(Selection_Prime[0],Selection[0]);
    not #3ns(Selection_Prime[1],Selection[1]);
    nand #5ns(f[0], B[0],Selection_Prime[1],Selection_Prime[0]);
    nand #5ns(f[1], B[1],Selection_Prime[1],Selection[0]);
    nand #5ns(f[2], B[2],Selection[1],Selection_Prime[0]);
    nand #5ns(f[3], B[3],Selection[1],Selection[0]);
    nand #5ns(Out,f[0],f[1],f[2],f[3]);
endmodule
```

Figure 2.1: 4x1 multiplexer implementation

Test Bench for 4x1 Multiplexer

To make sure the multiplexer works as desire I built a test bench as the follow:

```
module MuxTest;
    reg [3:0] B_Test;
    reg [1:0] Selection_Test;
    wire Out_Test;
    Mux4to1 Test(B_Test,Selection_Test,Out_Test);
    initial
    |   begin
            B_Test=4'b0110;
            Selection_Test=0;
            repeat(3)
            #30ns Selection_Test=Selection_Test+1;
        end
endmodule
```

Figure 2.2: multiplexer test bench implementation

Discussion and results:

1- Test Benches are more powerful than regular simulations

2- I used nand gates to make the multiplexer more efficient and take less time

3- The mux takes inputs and return the output depends on the selection

Stage 1:

Creating the AU using Multiplexers and several Full Adders (Ripple Adder).

The AU should look as the follow:

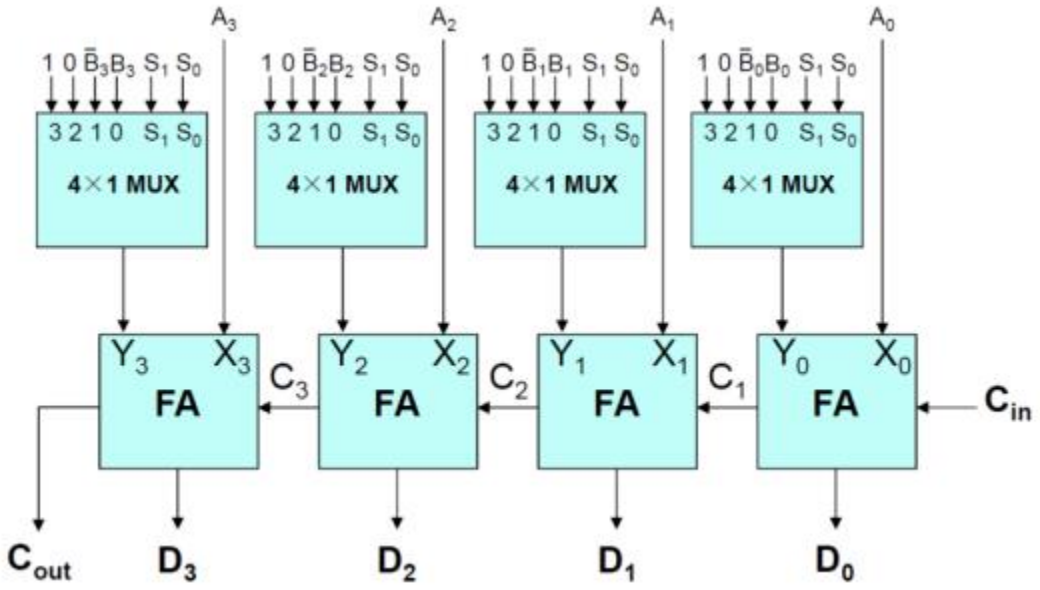

Figure 2.3: Stage 1

Full Adder:

Full Adder module (FA) takes 3 inputs each is one bit (X,Y,Cin) and gives two outputs (Sum, Cout)

Implemented as bellow:

```
module FA(X,Y,Cin,Sum,Cout);
    input X,Y,Cin;
    output Sum,Cout;
    wire w1,w2,w3;

    xor #11ns(Sum,X,Y,Cin);
    nand #5ns(w1,X,Y);
    nand #5ns(w2,X,Cin);
    nand #5ns(w3,Y,Cin);
    nand #5ns(Cout,w1,w2,w3);

endmodule
```

Figure 2.4: Full adder implementation

Full Adder Test Bench

For creating the test bench, I used an implementation from (Dr. Abdallatif Abuissa Slides)

As the follow:

```
module FATest;
    reg X_Test,Y_Test,Cin_Test;
    wire Sum_Test,Cout_Test;

    FA G(X_Test,Y_Test,Cin_Test,Sum_Test,Cout_Test);

    initial
        begin
            {Y_Test,X_Test,Cin_Test}=0;
            repeat(7)
            begin
            #100ns {Y_Test,X_Test,Cin_Test}={Y_Test,X_Test,Cin_Test}+1;
            #30ns $display("A=%b, B=%b, Cin=%b, D=%b, Cout=%b",X_Test,Y_Test,Cin_Test,Sum_Test,Cout_Test);
            end
        end
endmodule
```

Figure 2.5: Full adder test bench implementation

Stage 1 implementation:

```
module AU(A,B,Cin,S,D,Cout);
    input [3:0] A,B;
    input [1:0] S;
    input Cin;
    output [3:0]D;
    output Cout;
    wire [3:0] B_Prime;
    wire [3:1] C;
    wire [3:0] Mux_Output;

    not #3ns(B_Prime[0],B[0]);
    not #3ns(B_Prime[1],B[1]);
    not #3ns(B_Prime[2],B[2]);
    not #3ns(B_Prime[3],B[3]);

    Mux4to1 M1({1'b1,1'b0,B_Prime[0],B[0]},S,Mux_Output[0]);
    Mux4to1 M2({1'b1,1'b0,B_Prime[1],B[1]},S,Mux_Output[1]);
    Mux4to1 M3({1'b1,1'b0,B_Prime[2],B[2]},S,Mux_Output[2]);
    Mux4to1 M4({1'b1,1'b0,B_Prime[3],B[3]},S,Mux_Output[3]);


    FA F1(A[0],Mux_Output[0],Cin,D[0],C[1]);
    FA F2(A[1],Mux_Output[1],C[1],D[1],C[2]);
    FA F3(A[2],Mux_Output[2],C[2],D[2],C[3]);
    FA F4(A[3],Mux_Output[3],C[3],D[3],Cout);


endmodule
```

Figure 2.6: Stage 1 AU implementation

Stage 2:

Creating the AU using Multiplexers and carry look ahead adder.

Build the carry look ahead adder

For this module, I also used more nand gate instead of and gates to reduce the cost and the delay of the AU as lower as possible and the implementation was as the follow:

```
module CLA(X,Y,Cin,Sum,Cout);
    input [3:0]X,Y;
    input Cin;
    output [3:0] Sum;
    output Cout;
    wire [3:1]C;
    wire [0:3]P;
    wire [0:3]G;
    wire [0:3]G_Prime;
    wire [10:1]Temp;

    not #3ns(G_Prime[0],G[0]);
    not #3ns(G_Prime[1],G[1]);
    not #3ns(G_Prime[2],G[2]);
    not #3ns(G_Prime[3],G[3]);

    xor #11ns(P[0],X[0],Y[0]);
    xor #11ns(P[1],X[1],Y[1]);
    xor #11ns(P[2],X[2],Y[2]);
    xor #11ns(P[3],X[3],Y[3]);

    and #7ns(G[0],X[0],Y[0]);
    and #7ns(G[1],X[1],Y[1]);
    and #7ns(G[2],X[2],Y[2]);
    and #7ns(G[3],X[3],Y[3]);

    nand #5ns(Temp[1],P[0],Cin);
    nand #5ns(C[1],G_Prime[0],Temp[1]);

    nand #5ns(Temp[2] , P[1] , G[0]);
    nand #5ns(Temp[3] , P[1] , P[0] , Cin);
    nand #5ns(C[2] , Temp[2] , Temp[3] , G_Prime[1]);

    nand #5ns(Temp[4] , P[2] , G[1]);
    nand #5ns(Temp[5] , P[2] , P[1] , G[0]);
    nand #5ns(Temp[6] , P[2] , P[1] , P[0],  Cin);
    nand #5ns(C[3] , Temp[4] , Temp[5] , Temp[6] , G_Prime[2]);

    nand #5ns(Temp[7] , P[3] , G[2]);
    nand #5ns(Temp[8] , P[3] , P[2] , G[1]);
    nand #5ns(Temp[9] , P[3] , P[2] , P[1],  G[0]);
    nand #5ns(Temp[10] , P[3] , P[2] , P[1],  P[0],Cin);
    nand #5ns(Cout , Temp[7] , Temp[8] , Temp[9] , Temp[10] , G_Prime[3]);

    xor #11ns(Sum[0] , P[0] , Cin);
    xor #11ns(Sum[1] , P[1] , C[1]);
    xor #11ns(Sum[2] , P[2] , C[2]);
    xor #11ns(Sum[3] , P[3] , C[3]);
endmodule
```

Figure 2.7: Carry Look-Ahead Adder implementation

Stage 2 implementation:

```
module AU2(A,B,Cin,S,D,Cout);
    input [3:0] A,B;
    input [1:0] S;
    input Cin;
    output [3:0]D;
    output Cout;
    wire [3:0] B_Prime;
    wire [3:1] C;
    wire [3:0] Mux_Output;

    not #3ns(B_Prime[0],B[0]);
    not #3ns(B_Prime[1],B[1]);
    not #3ns(B_Prime[2],B[2]);
    not #3ns(B_Prime[3],B[3]);

    Mux4to1 M1({1'b1,1'b0,B_Prime[0],B[0]},S,Mux_Output[0]);
    Mux4to1 M2({1'b1,1'b0,B_Prime[1],B[1]},S,Mux_Output[1]);
    Mux4to1 M3({1'b1,1'b0,B_Prime[2],B[2]},S,Mux_Output[2]);
    Mux4to1 M4({1'b1,1'b0,B_Prime[3],B[3]},S,Mux_Output[3]);


    CLA CLA1(A,Mux_Output,Cin,D,Cout);


endmodule
```

Figure 2.8: Stage 1 AU implementation

Test Generator

The generator has only one input, which is the clock, In addition has five outputs (the carry in, First Number (A), Second Number (B), Selection (S) and the expected answer).

Test Generator do the following things:

1- Loop all possible values (use for loop) or repeat…etc.

2- Produce the behavioural output using direct operations

The implementation as the following:

```
module Generator(CLK,A,B,Cin,S,Ans);
     input CLK;
     output reg [3:0] A,B;
     output reg Cin;
     output reg [1:0] S;
     output reg [4:0]Ans;
     integer counter=0;
     integer E=1;
     always @(posedge CLK)
          if (E)
               begin
                    {S,Cin,A,B}=counter;
                    counter=counter+1;
                    case ({S,Cin})
                         0: Ans=A+B;
                         1: Ans=A+B+1'b1;
                         2: Ans=A+{1'b0,~B};
                         3: Ans=A+{1'b0,~B}+1'b1;
                         4: Ans=A;
                         5: Ans=A+1'b1;
                         6: Ans=A+4'b1111;
                         7: Ans={1'b1,A};
                    endcase
                    if(counter==2**11)
                         E=0;
               end

endmodule
```

Figure 2.9: Test Generator implementation

Analyzer

The analyzer takes three inputs (Clock, the expected answer and the module answer) and do not give any outputs.

The analyzer checks if the there is an error or not in the module answer by compare it with the expected answer from the test generator.

The implementation as the following:

```
module Analayzer(CLK,A,B,Mode,AUAns,GenAns);
     input CLK;
     input [3:0] A,B;
     input [2:0] Mode;
     input [4:0] AUAns, GenAns;
     always @(negedge CLK)
          if(AUAns[4:0] != GenAns[4:0])
               $display ("A=%b, B=%b, Mode=%b, AUAns=%b, GenAns=%b",A,B,Mode,AUAns,GenAns);
endmodule
```

Figure 2.10: Analyzer implementation

Build the first Verification with AU1:

Verification contains three primary components the Test Generator, the module and The Analyzer

The test generator generate the inputs for the module and the expected value for the analyzer then wait a specific delay until the module compute it answer then give it to analyzer to check if the answer from the module is correct (same as expected).

The implementation:

```
module Stage1_Test;
    reg CLK=0;
    reg [3:0] A,B;
    reg Cin;
    reg [1:0] S;
    reg [4:0] Ans;
    wire [3:0]Sum;
    wire Cout;
    Generator G(CLK,A,B,Cin,S,Ans);
    AU Au(A,B,Cin,S,Sum,Cout);
    Analayzer Anz(CLK,A,B,{S,Cin},{Cout,Sum},Ans);
    always
        begin
        #100ns CLK=~CLK;
        end
    initial #1000us $finish;
endmodule
```

Figure 2.11: Verification for stage 1 implementation

Build the second Verification with AU2:

Same as the previous the only different is the module we used is AU2.

```
module Stage2_Test;
    reg CLK=0;
    reg [3:0] A,B;
    reg Cin;
    reg [1:0] S;
    reg [4:0] Ans;
    wire [3:0]Sum;
    wire Cout;
    Generator G(CLK,A,B,Cin,S,Ans);
    AU2 Au2(A,B,Cin,S,Sum,Cout);
    Analayzer Anz(CLK,A,B,{S,Cin},{Cout,Sum},Ans);
    always
        begin
        #100ns CLK=~CLK;
        end
    initial #1000us $finish;
endmodule
```

Figure 2.12: Verification for stage 2 implementation

Conclusion

1- I achieve the aim of this project to design an Arithmetic Unit using Verilog in Active HDL then write complete code for functional verification.

2- Understood how Ripple and Look-ahead Adders work and how to reduce the delay of both using as mush nand gates.

3- Using universal gates such nand and nor to implement other gates may cost less and reduce the delay

4- The results obtained from the analyzer and the test generator agree with the modules AU, AU2 that means the design was built correctly.

5- The maximum delay for the AU 1 is 54ns (13ns Mux+42ns Ripple Adder), The maximum delay for the AU 2 is 47ns (13ns Mux+34ns Carry Look-ahead Adder),

Feedback

I think this project given by Dr. Abdallatif Abuissa is Very suitable for covering all basic and advanced concepts of using Verilog language to implement hardware components.

References

1-https://www.electronicshub.org/multiplexerandmultiplexing/

2-https://www.ece.uvic.ca/~fayez/courses/ceng465/lab_465/project1/adders.pdf

3-https://www.geeksforgeeks.org/carry-look-ahead-adder/

Appendix

The following code is attached with all test benches

```
module Mux4to1(B, Selection          , Out);
        input [3:0]B;
        input [1:0] Selection;
        output Out;
        wire [3:0] f;
        wire [1:0] Selection_Prime;
        not #3ns(Selection_Prime[0],Selection[0]);
        not #3ns(Selection_Prime[1],Selection[1]);
        nand #5ns(f[0], B[0],Selection_Prime[1],Selection_Prime[0]);
        nand #5ns(f[1], B[1],Selection_Prime[1],Selection[0]);
        nand #5ns(f[2], B[2],Selection[1],Selection_Prime[0]);
        nand #5ns(f[3], B[3],Selection[1],Selection[0]);
        nand #5ns(Out,f[0],f[1],f[2],f[3]);
endmodule


module MuxTest;
        reg [3:0] B_Test;
        reg [1:0] Selection_Test;
        wire Out_Test;
        Mux4to1 Test(B_Test,Selection_Test,Out_Test);
        initial
                begin
                        B_Test=4'b0110;
                        Selection_Test=0;
                        repeat(3)
                        #30ns Selection_Test=Selection_Test+1;
                end
endmodule


module FA(X,Y,Cin,Sum,Cout);
        input X,Y,Cin;
        output Sum,Cout;
        wire w1,w2,w3;

        xor #11ns(Sum,X,Y,Cin);
        nand #5ns(w1,X,Y);
        nand #5ns(w2,X,Cin);
        nand #5ns(w3,Y,Cin);
        nand #5ns(Cout,w1,w2,w3);

endmodule


module FATest;
        reg X_Test,Y_Test,Cin_Test;
        wire Sum_Test,Cout_Test;
```

```
	FA G(X_Test,Y_Test,Cin_Test,Sum_Test,Cout_Test);

	initial
		begin
			{Y_Test,X_Test,Cin_Test}=0;
			repeat(7)
			begin
			#100ns {Y_Test,X_Test,Cin_Test}={Y_Test,X_Test,Cin_Test}+1;
			#30ns $display("A=%b, B=%b, Cin=%b, D=%b,
Cout=%b",X_Test,Y_Test,Cin_Test,Sum_Test,Cout_Test);
			end
		end
endmodule


module AU(A,B,Cin,S,D,Cout);
	input [3:0] A,B;
	input [1:0] S;
	input Cin;
	output [3:0]D;
	output Cout;
	wire [3:0] B_Prime;
	wire [3:1] C;
	wire [3:0] Mux_Output;

	not #3ns(B_Prime[0],B[0]);
	not #3ns(B_Prime[1],B[1]);
	not #3ns(B_Prime[2],B[2]);
	not #3ns(B_Prime[3],B[3]);

	Mux4to1 M1({1'b1,1'b0,B_Prime[0],B[0]},S,Mux_Output[0]);
	Mux4to1 M2({1'b1,1'b0,B_Prime[1],B[1]},S,Mux_Output[1]);
	Mux4to1 M3({1'b1,1'b0,B_Prime[2],B[2]},S,Mux_Output[2]);
	Mux4to1 M4({1'b1,1'b0,B_Prime[3],B[3]},S,Mux_Output[3]);


	FA F1(A[0],Mux_Output[0],Cin,D[0],C[1]);
	FA F2(A[1],Mux_Output[1],C[1],D[1],C[2]);
	FA F3(A[2],Mux_Output[2],C[2],D[2],C[3]);
	FA F4(A[3],Mux_Output[3],C[3],D[3],Cout);


endmodule




module Generator(CLK,A,B,Cin,S,Ans);
	input CLK;
	output reg [3:0] A,B;
	output reg Cin;
	output reg [1:0] S;
	output reg [4:0]Ans;
	integer counter=0;
	integer E=1;
	always @(posedge CLK)
		if (E)
```

```
                begin
                        {S,Cin,A,B}=counter;
                        counter=counter+1;
                        case ({S,Cin})
                                0: Ans=A+B;
                                1: Ans=A+B+1'b1;
                                2: Ans=A+{1'b0,~B};
                                3: Ans=A+{1'b0,~B}+1'b1;
                                4: Ans=A;
                                5: Ans=A+1'b1;
                                6: Ans=A+4'b1111;
                                7: Ans={1'b1,A};
                        endcase
                        if(counter==2**11)
                                E=0;
                end

endmodule



module Analayzer(CLK,A,B,Mode,AUAns,GenAns);
        input CLK;
        input [3:0] A,B;
        input [2:0] Mode;
        input [4:0] AUAns, GenAns;
        always @(negedge CLK)
                if(AUAns[4:0] != GenAns[4:0])
                        $display ("A=%b, B=%b, Mode=%b, AUAns=%b,
GenAns=%b",A,B,Mode,AUAns,GenAns);
endmodule




module Stage1_Test;
        reg CLK=0;
        reg [3:0] A,B;
        reg Cin;
        reg [1:0] S;
        reg [4:0] Ans;
        wire [3:0]Sum;
        wire Cout;
        Generator G(CLK,A,B,Cin,S,Ans);
        AU Au(A,B,Cin,S,Sum,Cout);
        Analayzer Anz(CLK,A,B,{S,Cin},{Cout,Sum},Ans);
        always
                begin
                #100ns CLK=~CLK;
                end
        initial #1000us $finish;
endmodule


module CLA(X,Y,Cin,Sum,Cout);
        input [3:0]X,Y;
        input Cin;
        output [3:0] Sum;
        output Cout;
```

```
        wire [3:1]C;
        wire [0:3]P;
        wire [0:3]G;
        wire [0:3]G_Prime;
        wire [10:1]Temp;

        not #3ns(G_Prime[0],G[0]);
        not #3ns(G_Prime[1],G[1]);
        not #3ns(G_Prime[2],G[2]);
        not #3ns(G_Prime[3],G[3]);

        xor #11ns(P[0],X[0],Y[0]);
        xor #11ns(P[1],X[1],Y[1]);
        xor #11ns(P[2],X[2],Y[2]);
        xor #11ns(P[3],X[3],Y[3]);

        and #7ns(G[0],X[0],Y[0]);
        and #7ns(G[1],X[1],Y[1]);
        and #7ns(G[2],X[2],Y[2]);
        and #7ns(G[3],X[3],Y[3]);

        nand #5ns(Temp[1],P[0],Cin);
        nand #5ns(C[1],G_Prime[0],Temp[1]);

        nand #5ns(Temp[2] , P[1] , G[0]);
        nand #5ns(Temp[3] , P[1] , P[0] , Cin);
        nand #5ns(C[2] , Temp[2] , Temp[3] , G_Prime[1]);

        nand #5ns(Temp[4] , P[2] , G[1]);
        nand #5ns(Temp[5] , P[2] , P[1] , G[0]);
        nand #5ns(Temp[6] , P[2] , P[1] , P[0],  Cin);
        nand #5ns(C[3] , Temp[4] , Temp[5] , Temp[6] , G_Prime[2]);

        nand #5ns(Temp[7] , P[3] , G[2]);
        nand #5ns(Temp[8] , P[3] , P[2] , G[1]);
        nand #5ns(Temp[9] , P[3] , P[2] , P[1],  G[0]);
        nand #5ns(Temp[10] , P[3] , P[2] , P[1],  P[0],Cin);
        nand #5ns(Cout , Temp[7] , Temp[8] , Temp[9] , Temp[10] , G_Prime[3]);

        xor #11ns(Sum[0] , P[0] , Cin);
   xor #11ns(Sum[1] , P[1] , C[1]);
   xor #11ns(Sum[2] , P[2] , C[2]);
   xor #11ns(Sum[3] , P[3] , C[3]);
endmodule



module AU2(A,B,Cin,S,D,Cout);
        input [3:0] A,B;
        input [1:0] S;
        input Cin;
        output [3:0]D;
        output Cout;
        wire [3:0] B_Prime;
        wire [3:1] C;
        wire [3:0] Mux_Output;

        not #3ns(B_Prime[0],B[0]);
        not #3ns(B_Prime[1],B[1]);
```

```
        not #3ns(B_Prime[2],B[2]);
        not #3ns(B_Prime[3],B[3]);

        Mux4to1 M1({1'b1,1'b0,B_Prime[0],B[0]},S,Mux_Output[0]);
        Mux4to1 M2({1'b1,1'b0,B_Prime[1],B[1]},S,Mux_Output[1]);
        Mux4to1 M3({1'b1,1'b0,B_Prime[2],B[2]},S,Mux_Output[2]);
        Mux4to1 M4({1'b1,1'b0,B_Prime[3],B[3]},S,Mux_Output[3]);


        CLA CLA1(A,Mux_Output,Cin,D,Cout);


endmodule



module Stage2_Test;
        reg CLK=0;
        reg [3:0] A,B;
        reg Cin;
        reg [1:0] S;
        reg [4:0] Ans;
        wire [3:0]Sum;
        wire Cout;
        Generator G(CLK,A,B,Cin,S,Ans);
        AU2 Au2(A,B,Cin,S,Sum,Cout);
        Analayzer Anz(CLK,A,B,{S,Cin},{Cout,Sum},Ans);
        always
                begin
                #100ns CLK=~CLK;
                end
        initial #1000us $finish;
endmodule
```