Faculty of Engineering and Technology
Electrical and Computer Engineering Department

Advanced Digital Systems Design

(ENCS3310)

Project Report

Student's Name: Maha Mali

Student's ID: 1200746

Sec: 2

Instructor: Dr. Abdallatif Abuissa

Date: 13/8/2022

Contents

List Of Figures:

1. Introduction and Background

The aim of this project is to design Arithmetic Unit in two stages structurally by using different type of basic gate with certain delays, in two stages. The first stage it will implemented by using ripple adder and the second stage It will implement by using look ahead adder. After that we want to write a complete code for functional verification.

2. Information About Some Types of Adders

2.1 Ripple Adder

A ripple carry adder is a logic circuit in which the carry-out of each full adder is the carry in of the next full adder. It is called a ripple carry adder because each carry bit gets rippled into the next stage. In a ripple carry adder the sum and carry out bits of any half adder stage is not valid until the carry in of that stage occurs. Propagation delays inside the logic circuitry is the reason behind this. Propagation delay is time elapsed between the application of an input and occurrence of the corresponding output. Consider a NOT gate, When the input is "0" the output will be "1" and vice versa. The time taken for the NOT gate's output to become "0" after the application of logic "1" to the NOT gate's input is the propagation delay here. Similarly, the carry propagation delay is the time elapsed between the application of the carry in signal and the occurrence of the carry out (Cout) signal. Block diagram of a 4-bit ripple carry adder is shown below in figure 1[1].

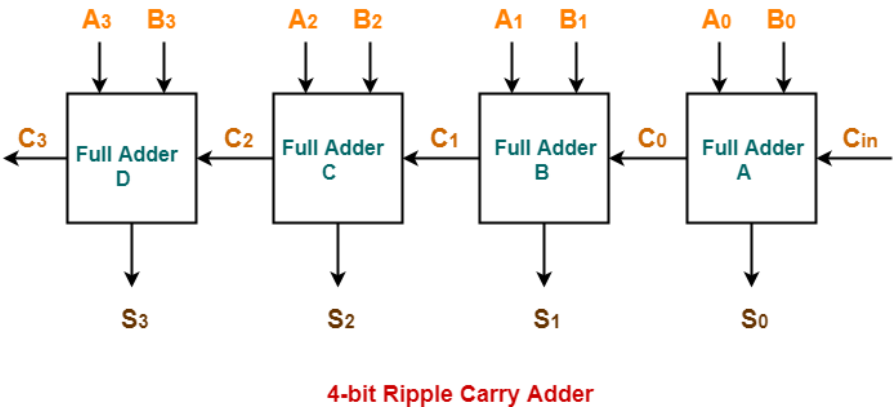


Figure1: Ripple Carry Adder

2.2 Look-ahead Adder

A digital computer must contain circuits which can perform arithmetic operations such as addition, subtraction, multiplication, and division. Among these, addition and subtraction are the basic operations whereas multiplication and division are the repeated addition and subtraction respectively [4].

Carry Look-ahead Adder is the faster adder circuit. It reduces the propagation delay, which occurs during addition, by using more complex hardware circuitry. It is designed by transforming the ripple-carry Adder circuit such that the carry logic of the adder is changed into two-level logic [4].

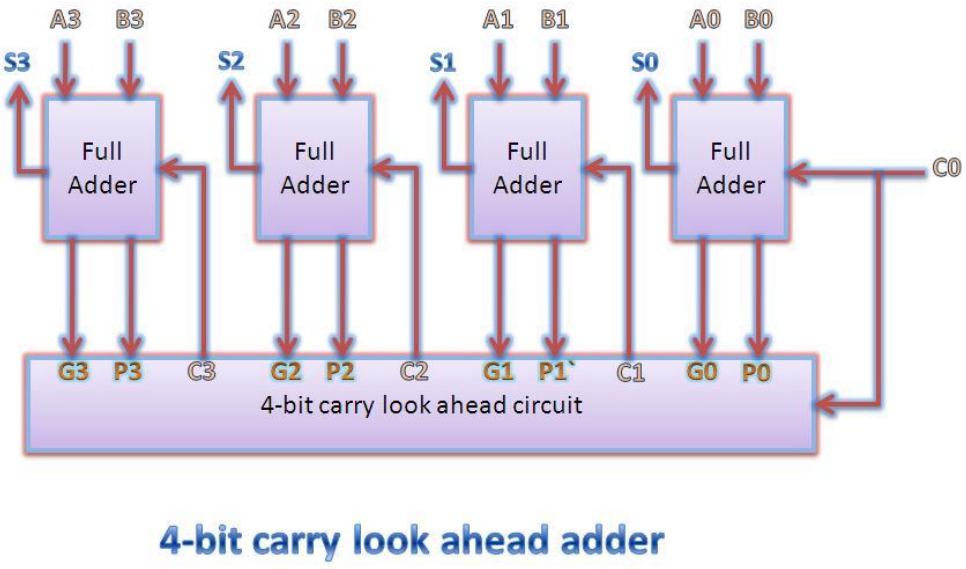


Figure2: Carry Look-ahead Adder

2.3 Arithmetic Unit

An arithmetic logic unit (ALU) is a digital circuit used to perform arithmetic and logic operations. It represents the fundamental building block of the central processing unit (CPU) of a computer. Modern CPUs contain very powerful and complex ALUs. In addition to ALUs, modern CPUs contain a control unit (CU).

Most of the operations of a CPU are performed by one or more ALUs, which load data from input registers. A register is a small amount of storage available as part of a CPU. The control unit tells the ALU what operation to perform on that data, and the ALU stores the result in an output register. The control unit moves the data between these registers, the ALU, and memory [2].

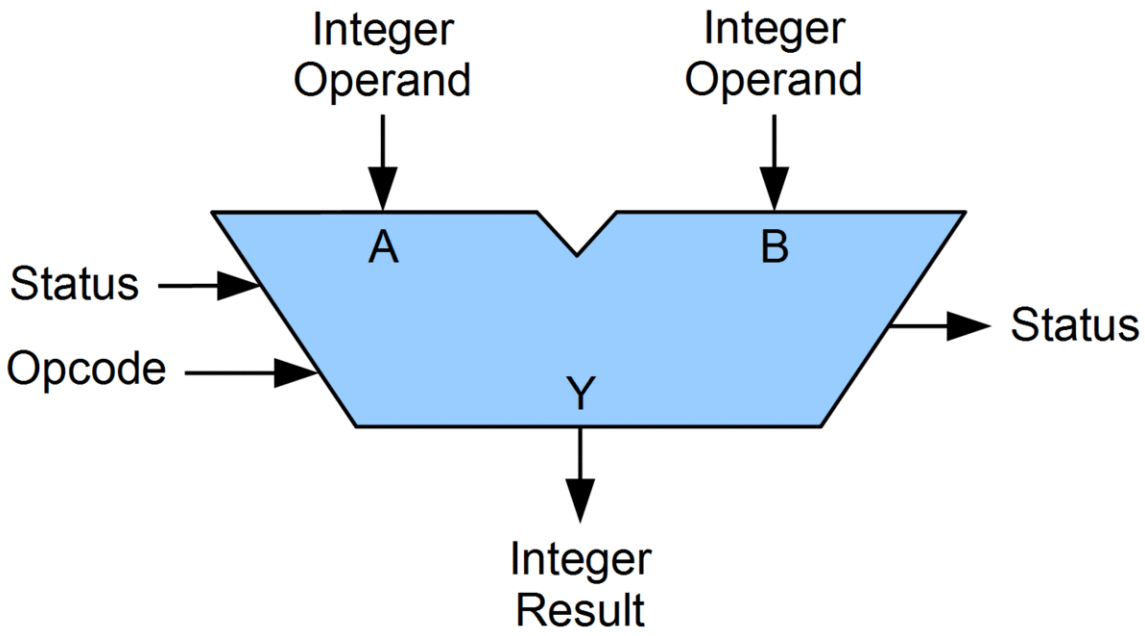


Figure3: Arithmetic Logic Unit

3. Design Process

3.1 Stage 1

- **4X1 MUX**

We implemented the multiplexer 4x1 structurally using basic gates (NOT, AND, OR) and two selection line. As shown in appendix **(**[9.1code 4X1 Mux](#) code**)**. The figure shows the block diagram and truth table for mux 4x1.

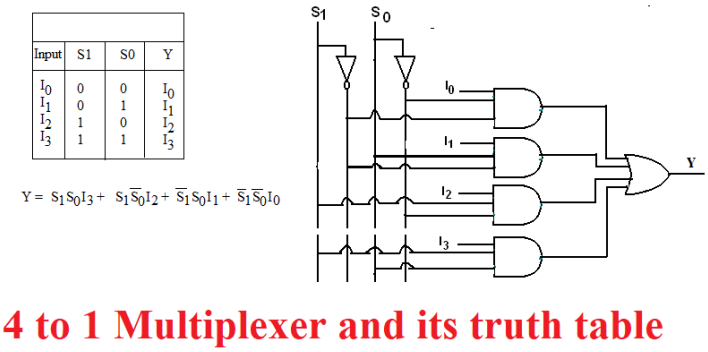


Figure4: Multiplexer 4x1

And the figure shows the simulation of mux4x1.so when the selection line is 00 the output will be the value of B. Then when the selection line is 01 the output will be the complement of B, when the selection line is 10 the output will be 0, when the selection line is 11 the output will be 1.

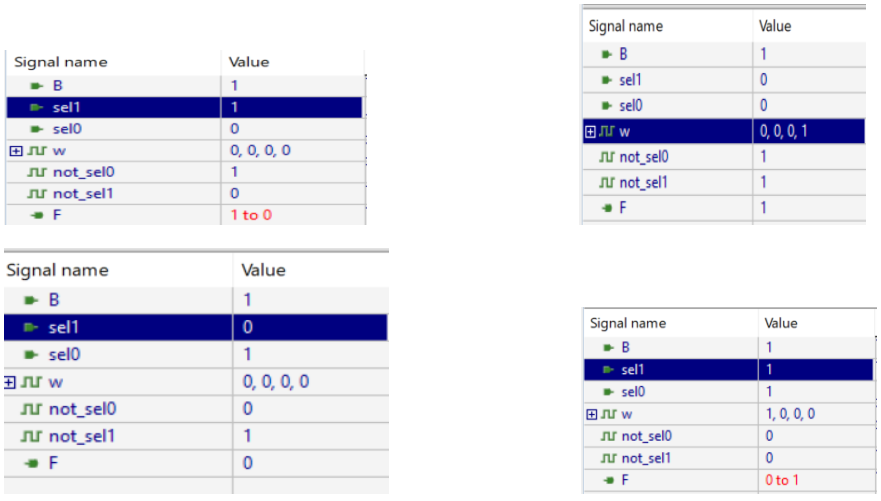
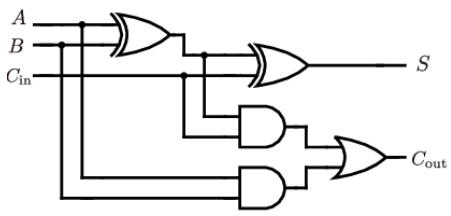


Figure5:Simulation Of Multiplexer 4x1

- **Full Adder**

We implemented the full adder structurally using basic gates (AND, OR, XOR), to use it later in building ripple full adder. As shown in appendix (**9.2code of Full Adder**). The figure shows the block diagram and truth table for full adder.



Inputs			Outputs	
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure6: Full Adder

The figure shows the simulation of full adder

Figure7: Simulation Of *Full Adder*

- 4-bit adder

We implemented the 4-bit adder using the full adder which was implementer previously, As shown in appendix (

9.3 Code 4-bit adder). The figure shows the block diagram for 4-bit adder.

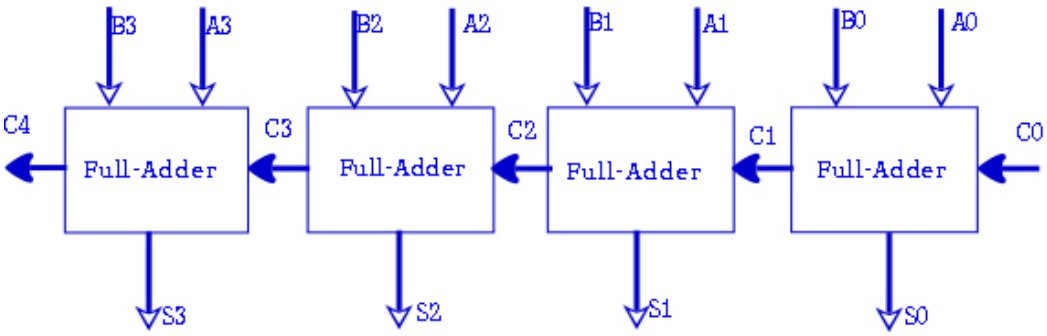


Figure8: 4-bit adder

The figure shows the simulation of 4 bit binary adder.

Signal name	Value
A	0
B	1
cary_in	0
caray	0
caray[2]	0
caray[1]	0
caray[0]	0
sum	1
cary_out	0

Figure9: Simulation *4-bit adder*

- **Stage 1 Ripple Carry Adder**

After completion of building full adder,4-bit adder, 4x2 Mux, we built the stage 1 Arithmetic Unit by using ripple carry adder. Before we build stage1 we write the code for the system (Arithmetic Unit) as shown in appendix (**9.4 code for Arithmetic** *Unit*). From here we can easily modify the design as shown in appendix (**9.5 Code of Stage 1 with ripple carry** *adder*). In this stage I calculate the maximum latency for ripple carry adder. Is the sum of the gate delay which build the full adder as shown in figure10. so, we add the gate delay (11+11+7+7+7=43ns) then the total cycles taken in the full adder is 43 ns the Ripple carry adder will take (4*43) ns that's equal to 172ns.And for mux we add the gate delay (11+11+7+7+7=43ns) then the total cycles taken in the full adder (2*3+5*7=41 ns) the Ripple carry adder will take (4*41=164) ns. Finally, the maximum latency equal (164+172) =336ns.

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

Figure10:Gate Delay

3.2 Stage 2

3.2.1 stage2 Carrey Look Ahead Adder

This stage is very similar with stage 1 but instead of using ripple carry adder we use Carrey look ahead adder as shown in appendix (**9.6 Stage2 Code of Carrey Look Ahead** *Adder*). In this stage I calculate the maximum latency for Carrey look ahead adder which is the sum of gate delay(11*4+7*4+7*10+7*4+11*4)=214ns

4. Test Generator

The test generator is a code module that uses hardware description languages (HDL) to describe the stimulus to a logic design and check whether the design outputs match its specification or not.

In our project in the test generator, the input of test generator clk. And the output D which goes to result analyzer, and the outputs: a, b, select1, select0, carray_in they are input to the arithmetic unit as shown in figure 10. Also, we write all condition to check all possible value as shown in the code in appendix (***9.7 Test generator***).

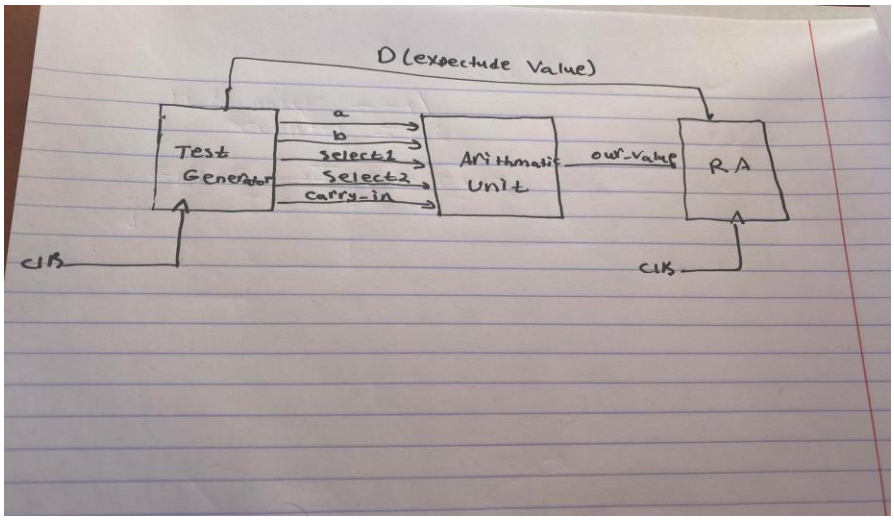


Figure11:Test Generator

At first, we define the input of the Test generator which is clk then we define the output as shown in figure 10, then inside the initial block we put the value of output 0 then repeat 31 time each time increment the output by 1 to check all the possible value. Also, Inside the always we put the test cases. But when I try to simulation the test generator the value is zero, I try to solve this problem but the value still zero as shown in figure 11.

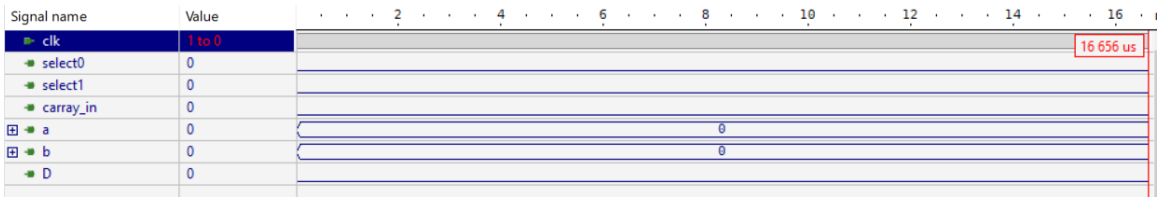


Figure 12:Test Generator Simulation

5. Result Analyze

The result analyzer checks the output from our system and compare the output with the right value in test generator when the clock input rises, if the output equals the value in test generator print on console(success) by using the task $display, and if not equal print (Error) by using the task $display, as shown in appendix (**9.8 Result** *Analyze*).

6. Verification

Verification is the process of ensuring that a given hardware design works as expected or not. Chip design is a very extensive and time-consuming process and costs millions to fabricate. Functional defects in the design if caught at an earlier stage in the design process will help save costs. If a bug is found later on in the design flow, then all of the design steps have to be repeated again which will use up more resources, money and time. If the entire design flow has to be repeated, then it's called a respin of the chip.[5].

So, in our design in the verification we built the whole design with test generator, result analyze, and the stage we want. So, the verification for stage1 as shown in appendix (**9.9.1 verification for stage** 1). And, the verification for stage2 as shown in appendix (**9.9.1 verification for stage** 2**)**

7. Conclusion and future works

In this project I almost managed to design Arithmetic Unit in two stages. First stage using ripple carry adder, and second stage using carry look ahead adder.

I faced many challenges in this project, especially when building the module for test generator. Because I did not know how the Test generator works, so I searched for test generator a lot through references on the internet. But still. I had a problem the module for Test generator, the simulation always gets the result 0. I tried to solve this problem but I couldn't solve it.

It is important to think about the development of the design in the future, to solve any defect in the design, also to make it include all possible cases because it is possible that there are certain cases which the system does not fit.

8. References

[1] https://www.circuitstoday.com/ripple-carry-adder. Accessed on 7-8-2022 at 1:23PM.

[2] https://study.com/academy/lesson/arithmetic-logic-unit-alu-definition-design-function.html . Accessed on 7-08-2022 at 2:14PM.

[3] https://electrosofts.com/verilog/mux.html .Accessed on 8-8-2022 at 5:34PM.

[4] Carry Look-ahead Adder - Circuit Diagram, Applications & Advantages (elprocus.com). Accessed on 8-8-2022 at 7:12PM.

[5] https://www.chipverify.com/systemverilog/systemverilog-tutorial . Accessed on 10-8-2022 at 1:12PM.

[6]https://ritaj.birzeit.edu/bzumsgs/attach/2145138/Fundamentals_of_Digital_Logic_with_Veril.pdf . Accessed on 12-8-2022 at 11:19AM.

9.Appendix

9.1code 4X1 Mux code

```
// Name:Maha Maher Mali
 // ID:1200746
// mux4x1
 module mux4x1( B,sel0,sel1,F);
        input B,sel0,sel1;
        output  F;
        wire not_sel0,not_sel1;
        wire w[3:0];
        not      #3 G1(not_sel0,sel0); // first invertor
        not #3 G2(not_sel1,sel1);// second  invertor
        and #7 G3(w[0],not_sel0,not_sel1,B);
        and      #7 G4(w[1],sel0,not_sel1,~B);
        and      #7 G5(w[2],not_sel0,sel0,0);
        and      #7 G6(w[3],sel0,sel1,1);
        or       #7 G7(F,w[0],w[1],w[2],w[3]);
 endmodule
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

9.2code of Full Adder

```
// Name:Maha Maher Mali
 // ID:1200746
// full-adder
 module FA(A,B,cary_in,sum,cary_out);
input A,B,cary_in;
output sum,cary_out;
wire w1,w2,w3;
xor#11 G1(w1,A,B);
xor #11 G2(sum,w1,cary_in);
and #7 G3(w2,A,B);
and #7 G4(w3,w1,cary_in);
or  #7 G5(cary_out,w2,w3);
endmodule
////////////////////////////////////////////////////////////////////////////////////////////////////
```

9.3 Code 4-bit adder

```
// Name:Maha Maher Mali
 // ID:1200746
//          4 bit  binary adder
module Four_Bit_Adder(A,B,cary_in,sum,cary_out);
input [3:0]A,B;
input cary_in;
output [3:0]sum;
output cary_out;
wire [2:0]caray;
FA      Block1(cary_in,A[0],B[0],sum[0],caray[0]);
FA      Block2(caray[0],A[1],B[1],sum[1],caray[1]);
FA      Block3(caray[1],A[2],B[2],sum[2],caray[2]);
FA      Block4(caray[2],A[3],B[3],sum[3],cary_out);
endmodule
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

9.4 code for Arithmetic Unit

```
// Name:Maha Maher Mali
// ID:1200746
// Arithmtic unit
module Arithmetic_Unit(A,s1,s0,cary_in,d,carray_out);
        input [3:0]A;
        input s1,s0,cary_in;
        output [3:0] d;
        output carray_out;
        wire [2:0]caray;
        wire [3:0]w;

        mux4x1 m1(s1,s0,,w[0]);
        FA f1(A[0],w[0],cary_in,d[0],caray[0]);


        mux4x1 m2(s1,s0,,w[1]);
        FA f2(A[1],w[1],caray[0],d[1],caray[1]);

        mux4x1 m3(s1,s0,,w[2]);
        FA f3(A[2],w[2],caray[1],d[2],caray[2]);

        mux4x1 m4(s1,s0,,w[3]);
        FA f4(A[3],w[3],caray[2],d[2],caray[2]);
        endmodule
////////////////////////////////////////////////////////////////////////////////////////////////
```

Dear DrAbdallatif Abuissa,

please find the attached file for the homework

thank you

9.5 Code of Stage 1 with ripple carry adder

```
module Arithmetic_Unit(A,s1,s0,cary_in,d,carray_out);
        input [3:0]A;
        input s1,s0,cary_in;
        output [3:0] d;
        output carray_out;
        wire [2:0]caray;
        wire [3:0]w;

        mux4x1 m1(s1,s0,,w[0]);
        FA f1(A[0],w[0],cary_in,d[0],caray[0]);

        mux4x1 m2(s1,s0,,w[1]);
        FA f2(A[1],w[1],caray[0],d[1],caray[1]);

        mux4x1 m3(s1,s0,,w[2]);
        FA f3(A[2],w[2],caray[1],d[2],caray[2]);

        mux4x1 m4(s1,s0,,w[3]);
        FA f4(A[3],w[3],caray[2],d[2],caray[2]);
        endmodule
```

9.6 Stage2 Code of Carrey Look Ahead Adder

```
// Name:Maha Maher Mali
        // ID:1200746
        //carry look ahead

  module Carry_Look_Ahead_Adder(a,b,carry_in,carry_out,sum);

        input carry_in;
        input[3:0]a,b;

        output [3:0]sum;
        output carry_out;

        wire [3:0]w,u;
        wire [9:0]e;
        wire [4:0]q;

xor#(11ns)(w[0],a[0],b[0]);
xor#(11ns)(w[1],a[1],b[1]);
xor#(11ns)(w[2],a[2],b[2]);
xor#(11ns)(w[3],a[3],b[3]);

and#(7ns)(u[0],a[0],b[0]);
```

```
and#(7ns)(u[1],a[1],b[1]);
and#(7ns)(u[2],a[2],b[2]);
and#(7ns)(u[3],a[3],b[3]);

assign q[0]=carry_in;

and #(7ns)(e[0],w[0],carry_in);
and #(7ns)(e[1],w[1],u[0]);
and #(7ns)(e[2],w[1],w[0],carry_in);
and #(7ns)(e[3],w[2],u[1]);
and #(7ns)(e[4],w[2],w[1],u[0]);
and #(7ns)(e[5],w[1],w[1],w[0],carry_in);
and #(7ns)(e[6],w[3],u[2]);
and #(7ns)(e[7],w[3],w[2],u[1]);
and #(7ns)(e[8],w[3],w[2],w[1],u[0]);
and #(7ns)(e[9],w[3],w[2],w[1],w[0],carry_in);

or #(7ns)(q[1],u[0],e[0]);
or #(7ns)(q[2],u[1],e[1],e[2]);
or #(7ns)(q[3],u[2],e[3],e[4],e[5]);
or #(7ns)(q[4],u[3],e[6],e[7],e[8],e[9]);

xor #(11ns)(sum[0],w[0],q[0]);
xor #(11ns)(sum[1],w[1],q[1]);
xor #(11ns)(sum[2],w[2],q[2]);
xor #(11ns)(sum[3],w[3],q[3]);

assign carry_out=q[4];
```

endmodule

9.7 Test generator

```
module the_test_generator(clk,a,b,select0,select1,carray_in, D);
        input clk;

        output reg  [3:0] a,b;
        output  reg select0,select1;
        output reg carray_in;
        output reg D;
        initial @(posedge clk)
                                begin
                                        {a,b,carray_in,select0,select1}=0;
                                        repeat (31)

        {a,b,carray_in,select0,select1}={a,b,carray_in,select0,select1}+1;
                                        end

        always @(posedge clk)
                begin


                if(select1==0 & select0==0 &carray_in==0 ) // add
                                D=a+b;

                else if (select1==0 & select0==0 & carray_in==1)//          add with carry
                                D= a+b+1;
```

```
else if (select1==0 & select0==1 & carray_in==0)// subtract with borrow
                    D= a+~b;

else if (select1==0 & select0==1 & carray_in==1)// subtract
                    D= a+~b+1;

else if (select1==1 & select0==0 & carray_in==0)// transfer a
                    D= a;


else if (select1==1 & select0==0 & carray_in==1) //         incremnt
                    D= a+1;

else if (select1==1 & select0==1 & carray_in==0 )          //          decremnt
                    D= a-1;

else if (select1==1 & select0==1 & carray_in==1 )          //  transfer a
                    D=a;

end
endmodule
```

9.8 Result Analyze

```
module Result_Analyze(result_sys,correct_result,clock);
        input [3:0]result_sys;     // the result from system
        input [3:0] correct_result;          // the correct result
        input clock;
    always @ (posedge clock)  begin

        if(result_sys[3:0]==correct_result[3:0])
                begin
                        $display("sucsess",$time);
                        $finish  ;
                end

        else
                begin
                 $display ("error",$time);
                 $finish;
        end
        end
        endmodule
```

9.9 verification

9.9.1 verification for stage 1

```
module varification_forsystem_Stage1(c_in,a,b,s0,s1,correct_value,system_value,clk);
        output [3:0]a,b;
        output s0,s1;
        input c_in,clk;
        output correct_value[3:0];
        input system_value[3:0];
        Test_Generator t1(clk,a,b,s0,s1,c_in,correct_value[3:0]);
        Arithmetic_Unit t2(a,b,s0,s1,c_in,system_value[3:0]);
        Result_Analyze t3(system_value[3:0],correct_value[3:0],clk);
Endmodule
////////////////////////////////////////////////
```

9.9.1 verification for stage 2

```
module varification_forsystem_Stage2(c_in,a,b,s0,s1,correct_value,system_value,clk);
        output [3:0]a,b;
        output s0,s1;
        input c_in,clk;
        output correct_value[3:0];
```

```
        input system_value[3:0];
        Test_Generator u1(clk,a,b,s0,s1,c_in,correct_value[3:0]);
        Carry_Look_Ahead_Adder u2 (a,b,s0,s1,c_in,system_value[3:0]);
        Result_Analyze u3(system_value[3:0],correct_value[3:0],clk);
endmodule
```