بسم الله الرحمن الرحيم

BIRZEIT UNIVERSITY

Faculty of Engineering and Technology

Electrical and Computer Engineering Department

Advanced Digital Systems Design

(ENCS3310)

Verilog Project Report

Arithmetic Unit in Two Stages

Prepared by : Momen Salem **ID : 1200034**

Instructor : Dr. Abdallatif Abuissa

Section : 2 **Date : 22-8-22**

Table of contents

Brief Introduction and Background 1
1. 4x1 Multiplexer (MUX) 1
2. Full Adder (FA) 1
a. Stage 1: ALU With Ripple-Carry Adder 2
b. Stage 2: ALU With Carry-Lookahead Adder 4
Conclusion and future work 6

Table of figures

Figure 1. 4-bit Arithmetic Unit 1
Figure 2. 4x1 M ux Block Diagram and Circuit Diagram 1
Figure 3. Full Adder Block Diagram and Circuit Diagram 2
Figure 4. Worst Case delay 2
Figure 5. Stage 1 Simulation (a) 3
Figure 6. Stage 1 Simulation (b) 3
Figure 7. Before and After Error 3
Figure 8. Carry-Lookahead Adder 4
Figure 9. Carry-Lookahead Adder Circuit diagram 5
Figure 10. Worst Case delay 5
Figure 11. OR in the Mux 4x1 (before Error) 6
Figure 12. XOR in the Mux 4x1 (After Error) 6

Table of tables

Table 1. ALU Operation 2

Brief Introduction and Background

In this project we will construct a simple arithmetic-logic unit (ALU) that will do several operations. The operations are decided by a multiplexer (MUX) and a selection lines, we have two 4-bit input numbers and 1-bit carry in input. In addition, the ALU have 4-bit output and 1-bit carry out as we see in the 1st figure. In the project we must only use the structural method because we will deal with delay of gates and use registers.

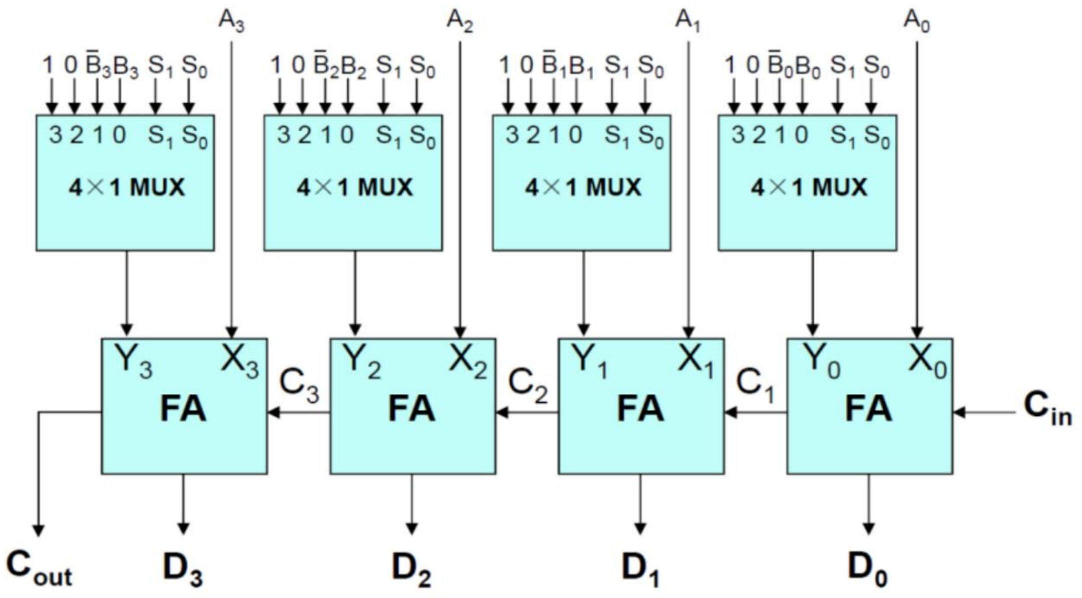


Figure 1. 4-bit Arithmetic Unit

From the figure we have two main components: 1. 4x1 MUX 2. Full Adder

1. 4x1 Multiplexer (MUX)

To construct 4x1 MUX we can derive the equation of output from k-map easily.

The output of 4x1 MUX (if $I_3I_2I_1I_0$ are the input for MUX and S_1S_0 are selection lines)

output → $Y = I_0 \sim S_0 \sim S_1 + I_1\ S_0 \sim S_1 + I_2 \sim S_0\ S_1 + I_3\ S_0\ S_1$. In the bellow figure we conclude the block diagram and block circuit for the 4x1 MUX.

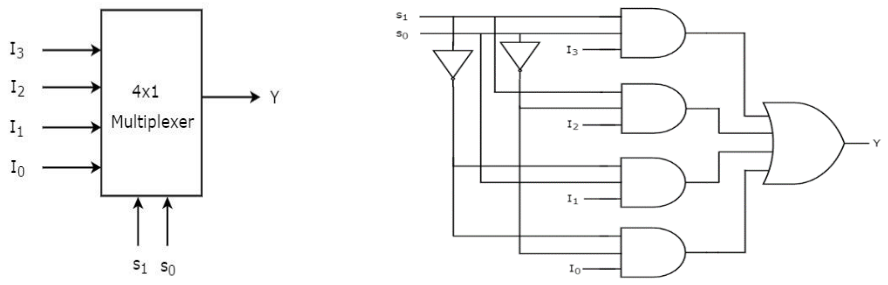


Figure 2. 4x1 M ux Block Diagram and Circuit Diagram

2. Full Adder (FA)

By the same way as we construct the 4x1 MUX we can construct the full adder by finding the equation of the outputs sum and carry out.

If the inputs are C_{in} X Y then the outputs are S → $C_{in} \oplus X \oplus Y$. and $C_{out} = C_{in} X + X\ Y + C_{in}\ Y$.

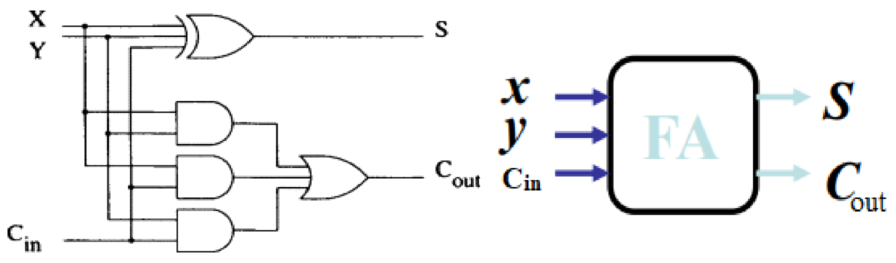


Figure 3. Full Adder Block Diagram and Circuit Diagram

We can conclude the operations for the ALU in the following table:

Table 1. ALU Operation

Op #	**Select**			**Input**	**Output**	**Operation discerption**
	S_1	**S_0**	**C_{in}**	**Y**	**$D = A + Y + C_{in}$**	
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 A**
7	**1**	**1**	**0**	**1**	**D = A – 1**	**Decrement A**
8	**1**	**1**	**1**	**1**	**D = A**	**Transfer A**

So, from the table we have 7 different operations in our ALU (100, 111 are the same effect).

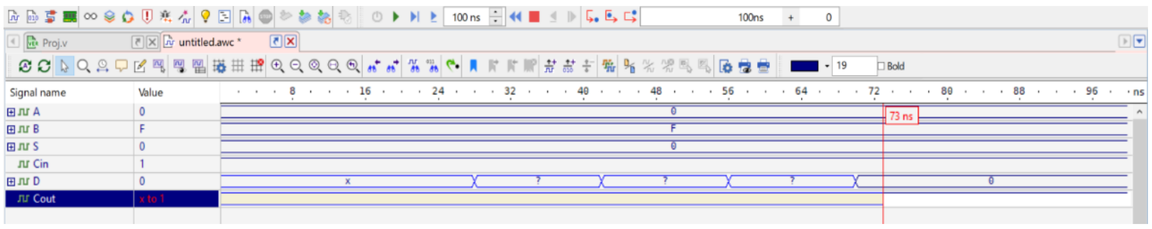
We have two different stages (the stages are just for the full adder circuit) the first one is using the ripple-carry adder which we know and which is slow because the carry in the second full adder will add the wrong carry until the carry from 1st stage change and in our situation because we have delays it will slow the addition operation so we have a latency time for the true value. One of the solutions is to use another fast adder such that carry-lookahead adder because in this full adder the carries don't need to wait for previous carry (the all carries are deal just with the carry in).

a. Stage 1: ALU With Ripple-Carry Adder

In ripple-carry adder the 2nd carry must wait until the first one is calculated properly.

In my result I built the mux and 4-bit full adder (by instantiate full adder four times) and then use the testbench to check if the block work properly (for every block in the project). In addition, I put the 4-bit full adder in the stage 1 so by calculation and testing I see that when A = 4'h0, B = 4'hF, C_{in} = 1'b1, S = 2'b00 this is the worst case in my circuit

So the expected result will be A + B + C_{in} = 0 with carry out = 1 and by simulate it I decide that the clock for register will = 75ns to avoid the glitch of worst case.

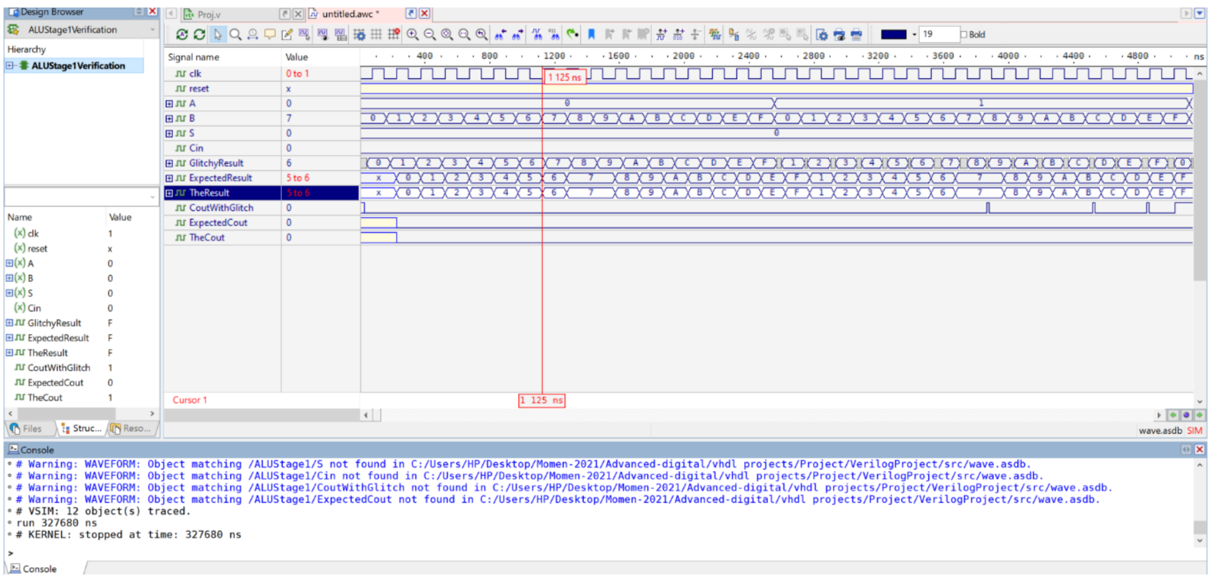


Signal name	Value
A	0
B	F
S	0
Cin	1
D	0
Cout	x to 1

Figure 4. Worst Case delay

After that I built the stage with registers for inputs (A, B, C_{in}, S) and for outputs (D, C_{out}).

Then I built the test generator and output or design analyzer and put the modules in the verification and the output are true but I have a lot of problems such that for the C_{out} when the selection lines are 1 or 3 the expected carry is complemented to the carry out from the stage (unit under test). and in the verification, I announced that the good output is faster than the output from the circuit. so, because I want to check if the results are equal in the same clock edge, I put the result that out from **test generator (good result) to the register** and by this I solve the glitch and the delay between the results.



```
• # Warning: WAVEFORM: Object matching /ALUStage1/S not found in C:/Users/HP/Desktop/Momen-2021/Advanced-digital/vhdl projects/Project/VerilogProject/src/wave.asdb.
• # Warning: WAVEFORM: Object matching /ALUStage1/Cin not found in C:/Users/HP/Desktop/Momen-2021/Advanced-digital/vhdl projects/Project/VerilogProject/src/wave.asdb.
• # Warning: WAVEFORM: Object matching /ALUStage1/CoutWithGlitch not found in C:/Users/HP/Desktop/Momen-2021/Advanced-digital/vhdl projects/Project/VerilogProject/src/wave.asdb.
• # Warning: WAVEFORM: Object matching /ALUStage1/ExpectedCout not found in C:/Users/HP/Desktop/Momen-2021/Advanced-digital/vhdl projects/Project/VerilogProject/src/wave.asdb.
• # VSIM: 12 object(s) traced.
• run 327680 ns
• # KERNEL: stopped at time: 327680 ns
>
```

Figure 5. Stage 1 Simulation (a)

As we see the results are at the same clock cycle and the compiler did not print any thing and I simulate the all-possible values from A, B, S, C_{in} are zeros to ones.

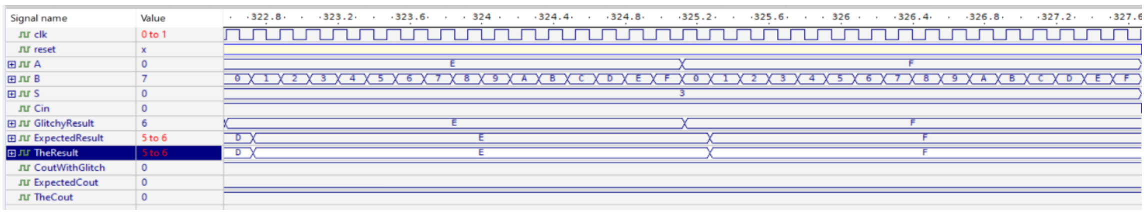
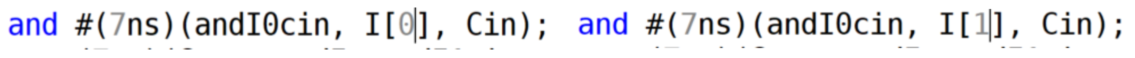


Figure 6. Stage 1 Simulation (b)

So, the result is good except the C_{out} when S = 10 or 11 because the subtraction in digital differs than normal subtraction (the two's complement is the problem for the value of C_{out}).

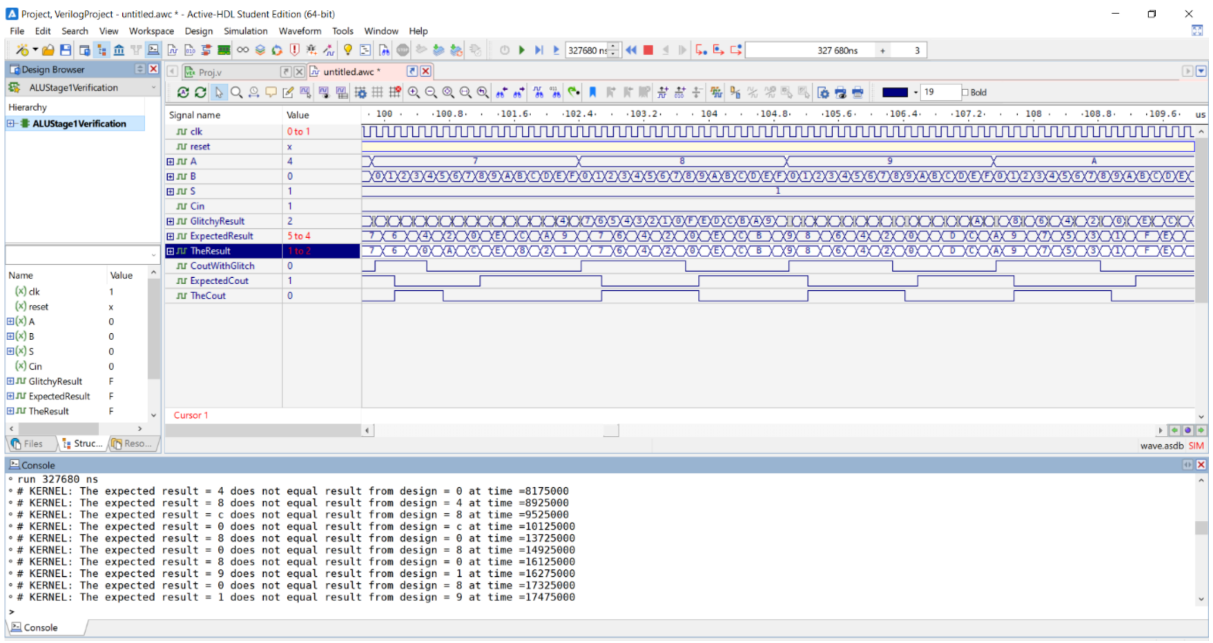
Finally, I put an error and check if the result change or not so I change one input in the full adder as bellow figures (from I[0] to I[1]).



```
and #(7ns)(andI0cin, I[0], Cin);  and #(7ns)(andI0cin, I[1], Cin);
```

Figure 7. Before and After Error

As expected, the result is not good and the compiler display a lot of error messages.



```
run 327680 ns
# KERNEL: The expected result = 4 does not equal result from design = 0 at time =8175000
# KERNEL: The expected result = 8 does not equal result from design = 4 at time =8925000
# KERNEL: The expected result = c does not equal result from design = 8 at time =9525000
# KERNEL: The expected result = 0 does not equal result from design = c at time =10125000
# KERNEL: The expected result = 8 does not equal result from design = 0 at time =13725000
# KERNEL: The expected result = 0 does not equal result from design = 8 at time =14925000
# KERNEL: The expected result = 8 does not equal result from design = 0 at time =16125000
# KERNEL: The expected result = 9 does not equal result from design = 1 at time =16275000
# KERNEL: The expected result = 0 does not equal result from design = 8 at time =17325000
# KERNEL: The expected result = 1 does not equal result from design = 9 at time =17475000
>
```

b. Stage 2: ALU With Carry-Lookahead Adder

In this stage we want to fast the operation in full adder by using carry-lookahead adder because in this adder the carry is a function of the C_{in} so the carry waits less than in the ripple-carry adder.

$C_{i+1} = X_iY_i + X_iC_i + Y_iC_i$, take ci as a common factor → $C_{i+1} = X_iY_i + C_i (X_i + Y_i)$

Let $G_i = X_iY_i$ be the generate function, $P_i = X_i + Y_i$ be the propagate function

By this equation we can find any C_{out} with respect to the C_{in} and some (and) and (or) gates.

This will make the addition faster than ripple-carry adder.

In our ALU we have 4 C so let $C_0 = C_{in}$.

By definition we can find the three C by the equation of the carry-lookahead adder.

For example, $C_1 = G_0 + P_0C_0$, $C_2 = G_1 + G_0P_1 + P_1P_0C_0$ ….. and so on until C_4 which the last carry in the 4-bit full adder. By this addition combinational circuit, I speedup the adder as bellow figure.

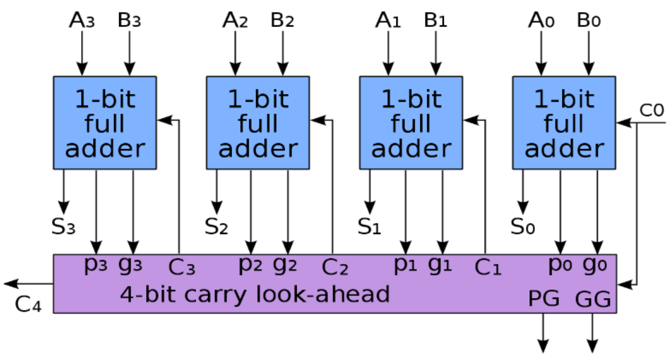


Figure 8. Carry-Lookahead Adder

This figure is circuit diagram for out four-bit carry-lookahead adder.

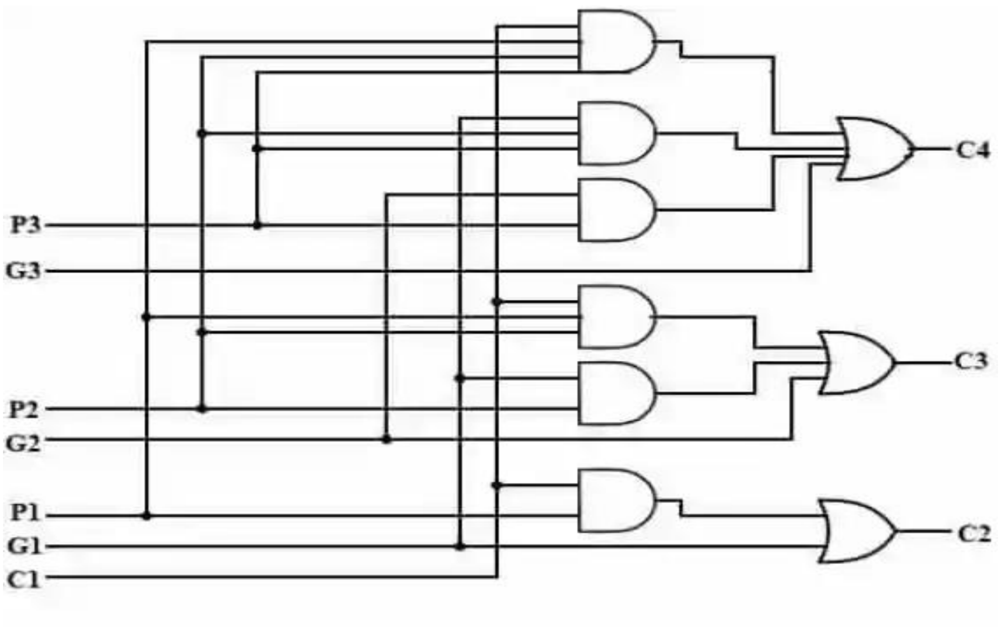


Figure 9. Carry-Lookahead Adder Circuit diagram

As we expected the carry-lookahead adder is faster than ripple-carry adder and to find the worst case as I did for the ripple-carry adder.

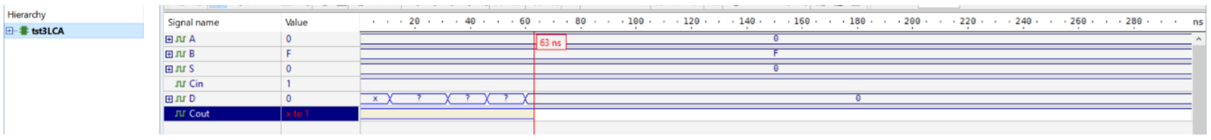
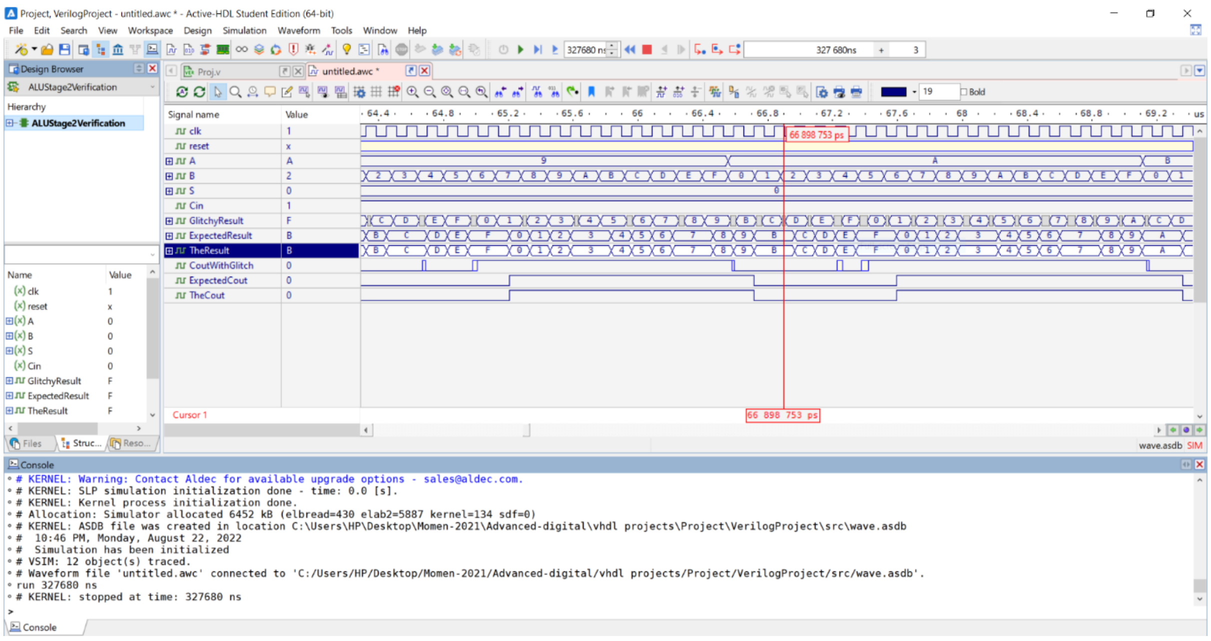


Figure 10. Worst Case delay

The result may have an error because I expect that the delay is faster and faster. So I choose the clock = 65 ns

The result in verification is good as bellow figure.



```
# KERNEL: Warning: Contact Aldec for available upgrade options - sales@aldec.com.
# KERNEL: SLP simulation initialization done - time: 0.0 [s].
# KERNEL: Kernel process initialization done.
# Allocation: Simulator allocated 6452 kB (elbread=430 elab2=5887 kernel=134 sdf=0)
# KERNEL: ASDB file was created in location C:\Users\HP\Desktop\Momen-2021\Advanced-digital\vhdl projects\Project\VerilogProject\src\wave.asdb
#  10:46 PM, Monday, August 22, 2022
#  Simulation has been initialized
# VSIM: 12 object(s) traced.
# Waveform file 'untitled.awc' connected to 'C:/Users/HP/Desktop/Momen-2021/Advanced-digital/vhdl projects/Project/VerilogProject/src/wave.asdb'.
run 327680 ns
# KERNEL: stopped at time: 327680 ns
>
```

There is no warning lines in the console.

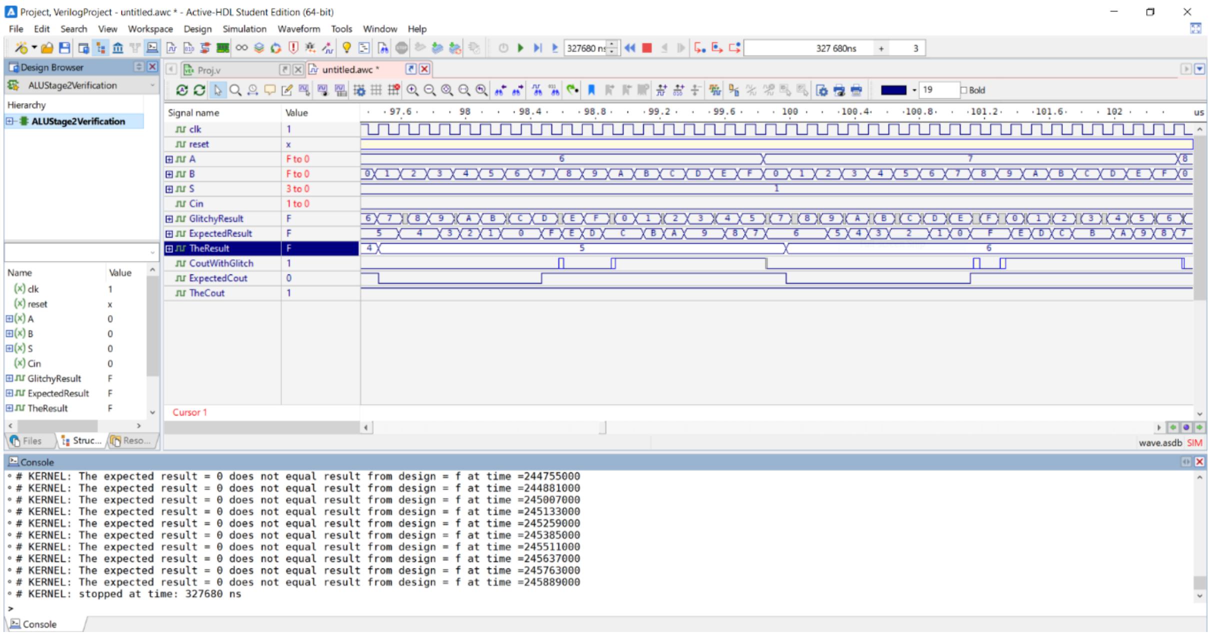
If we put an error as bellow figure.

```
or #(7ns)(Y, and_wire[0], and_wire[1], and_wire[2], and_wire[3]);
```

Figure 11. OR in the Mux 4x1 (before Error)

```
xor #(7ns)(Y, and_wire[0], and_wire[1], and_wire[2], and_wire[3]);
```

Figure 12. XOR in the Mux 4x1 (After Error)



```
# KERNEL: The expected result = 0 does not equal result from design = f at time =244755000
# KERNEL: The expected result = 0 does not equal result from design = f at time =244881000
# KERNEL: The expected result = 0 does not equal result from design = f at time =245007000
# KERNEL: The expected result = 0 does not equal result from design = f at time =245133000
# KERNEL: The expected result = 0 does not equal result from design = f at time =245259000
# KERNEL: The expected result = 0 does not equal result from design = f at time =245385000
# KERNEL: The expected result = 0 does not equal result from design = f at time =245511000
# KERNEL: The expected result = 0 does not equal result from design = f at time =245637000
# KERNEL: The expected result = 0 does not equal result from design = f at time =245763000
# KERNEL: The expected result = 0 does not equal result from design = f at time =245889000
# KERNEL: stopped at time: 327680 ns
>
```

A lot of errors appear.

Conclusion and future work

In conclusion I see the difference between the two adders and I know how to use registers properly for both inputs and outputs also how to check the circuits by test generator and design analyzer in the verification module. For future work I want to know how I can deal with carry out when we subtract (test the carry out properly) and how if I want to make the ALU bigger (can have more than 4-bit inputs).

Appendix

Project Code

```
module MUX4x1(I, S, Y);//I for input and S for selection line Y for output
	input [3:0]I;//there are four input for mux4x1
	input [1:0]S;//there are two selecion line to select from input
	output Y;
	wire [3:0]and_wire;//wire for the output of the and gate
	wire [1:0]not_wire;//wire to write delay for inverter
	not #(3ns)(not_wire[0], S[0]);
	not #(3ns)(not_wire[1], S[1]);
	and #(7ns)(and_wire[0], not_wire[1], not_wire[0], I[0]);
	and #(7ns)(and_wire[1], not_wire[1], S[0], I[1]);
	and #(7ns)(and_wire[2], S[1], not_wire[0], I[2]);
	and #(7ns)(and_wire[3], S[1], S[0], I[3]);
	or #(7ns)(Y, and_wire[0], and_wire[1], and_wire[2], and_wire[3]);
endmodule
module testbenchMux;//testbench for mux
	reg [3:0]I;
	reg [1:0]S;
	wire Y;
	MUX4x1 stage(I, S, Y);
	initial
		begin
			{I, S} = 6'b000_000;
			repeat(63)
			#10ns {I, S} = {I, S} + 1;
		end
endmodule

module FullAdder(I, Cin, S, Cout);
	input [1:0]I;
```

```
        input Cin;
        output S, Cout;
        wire andI, andI1cin, andI0cin;
        xor #(11ns)(S, I[1], I[0], Cin);//the output sum
        and #(7ns)(andI, I[0], I[1]);
        and #(7ns)(andI1cin, I[1], Cin);
        and #(7ns)(andI0cin, I[0], Cin);
        or #(7ns)(Cout, andI, andI1cin, andI0cin);//the carry out
endmodule
module testbenchFA;//testbench for FA
        reg [1:0]I;
        reg Cin;
        wire S, Cout;
        FullAdder stage(I, Cin, S, Cout);
        initial
                begin
                        {I, Cin} = 3'b000;
                        repeat(7)
                        #10ns {I, Cin} = {I, Cin} + 1;
                end
endmodule

module FourBitFullAdder(X, Y, Cin, D, Cout);
        input [3:0]X, Y;
        input Cin;
        output [3:0]D;
        output Cout;
        wire [4:0]C;
        assign C[0] = Cin, Cout = C[4];
        genvar i;
        generate
        for(i = 0 ; i < 4 ; i = i + 1)
```

```
			begin:IFA//IFA -> mean instantiate full adder for four times
				FullAdder Stage({Y[i], X[i]}, C[i], D[i], C[i+1]);
			end
			endgenerate
endmodule
module testbench4bitfulladder;//testbench for 4-bit FA
		reg [3:0]X, Y;
		reg Cin;
		wire [3:0]D;
		wire Cout;
		FourBitFullAdder stage(X, Y, Cin, D, Cout);
		initial
			begin
				{X, Y, Cin} = 9'b0000_00000;
				repeat(511)
				#75ns {X, Y, Cin} = {X, Y, Cin} + 1;
			end
endmodule

module Stage1RippleCarryAdderWithoutClock(A, B, S, Cin, D, Cout);//without clock for the
glitchy result (without using registers)
		input [3:0]A, B;
		input [1:0]S;
		input Cin;
		output [3:0]D;
		output Cout;
		wire [3:0]NotB;
		wire [3:0]MuxO;//MuxO mean the output of the mux
		not #(3ns)(NotB[0], B[0]);
		not #(3ns)(NotB[1], B[1]);
		not #(3ns)(NotB[2], B[2]);
		not #(3ns)(NotB[3], B[3]);
		genvar i;
```

```
        generate
        for(i = 0 ; i < 4 ; i = i + 1)
                begin :buildALU
                        MUX4x1 MUX({1'b1, 1'b0, NotB[i], B[i]}, S, MuxO[i]);
                end
                endgenerate
                FourBitFullAdder SystemFA(A,  MuxO, Cin, D, Cout);
endmodule
module Stage1RippleCarryAdderWithClock(A, B, S, Cin, DA, CoutA, clk, reset);//the reuslt
without glitchy(Proper result)
        input [3:0]A, B;
        wire [3:0]AC, BC;
        input [1:0]S;
        wire [1:0]SC;
        input Cin, clk, reset;
        wire CinC;
        wire [3:0]D;
        output reg [3:0] DA;
        output reg CoutA;
        wire Cout;
        wire [3:0]NotB;
        wire [3:0]MuxO;//MuxO mean the output of the mux
        not #(3ns)(NotB[0], BC[0]);
        not #(3ns)(NotB[1], BC[1]);
        not #(3ns)(NotB[2], BC[2]);
        not #(3ns)(NotB[3], BC[3]);
        Register RA(A, clk, reset, AC);         //registers for inputs
        Register RB(B, clk, reset, BC);
        Register #(.n(2))RS(S, clk, reset, SC);
        Register RCin(Cin, clk, reset, CinC);
        defparam RCin.n = 1;
        FourBitFullAdder SystemFA(AC, MuxO, CinC, D, Cout);//four bit full adder
        Register RD(D, clk, reset, DA);//registers for outputs
```

```
            Register #(.n(1))RCout(Cout, clk, reset, CoutA);
            genvar i;
            generate
            for(i = 0 ; i < 4 ; i = i + 1)
                        begin :buildALU
                                    MUX4x1 MUX({1'b1, 1'b0, NotB[i], BC[i]}, SC, MuxO[i]);
                        end
                        endgenerate
endmodule
module testbenchS1withoutclock;//testbench for Ripple-carry adder (Glitchy Result)
            reg [3:0]A, B;
            reg [1:0]S;
            reg Cin;
            wire [3:0]D;
            wire Cout;
            Stage1RippleCarryAdderWithoutClock stage(A, B, S, Cin, D, Cout);
            initial
                        begin
                                    {A, B, S, Cin} = 11'b00000000000;
                                    repeat(2048)
                                    #75ns {A, B, S, Cin} = {A, B, S, Cin} + 1'b1;
                        end
endmodule
module testbenchS1withclock;//testbench for Ripple-carry adder (Not Glitchy Result)
            reg [3:0]A, B;
            reg [1:0]S;
            reg Cin, clock, reset;
            wire [3:0]D;
            wire Cout;
            Stage1RippleCarryAdderWithClock stage(A, B, S, Cin, D, Cout , clock, reset);
            initial
                        begin
```

```
                clock = 0;
                {S, Cin, A, B} = 11'b00000000000;
                repeat(2048)
                #160ns {S, Cin, A, B} = {S, Cin, A, B} + 1'b1;
            end
            always #75ns clock = ~clock;
endmodule

module D_FlipFlop(D, clk, Q, reset);//Dff
        input D, clk, reset;
        output reg Q;
        always @(posedge clk, reset)
            if(reset)
                Q <= 0'b0;
            else if(clk)
                Q <= D;
endmodule
module Register(D, clk, reset, Q);//Registers with input reset to the level(when 1 output = 0)
        parameter n = 4;
        input [n-1:0]D;
        input clk, reset;
        output reg [n-1:0]Q;
        always @(posedge clk, reset)
            if(reset)
                Q <= 0;
            else if(clk)
                Q <= D;
endmodule
module TestGeneraterForALU(AT, BT, ST, CinT, ResultR, CoutR, clk, reset);
        reg [3:0]A, B;
        reg [1:0]S;
        reg Cin;
```

```
    output [3:0]AT, BT;
    output [1:0]ST;
    output CinT;
    reg [3:0]ExpectedD;
    reg ExpectedCout;
    output reg [3:0]ResultR;
    output reg CoutR;
    Register Result_R(ExpectedD, clk, reset, ResultR);//i put this result in the register
because the input are in the registers so the outputs must be in registers
    Register #(.n(1))Cout_R(ExpectedCout, clk, reset, CoutR);
    input clk;
    input reset;
    assign AT = A,
            BT = B,
            ST = S,
            CinT = Cin;
    initial
        begin
            {S, Cin, A, B} = 11'b00000000000;
            repeat(2048)
            #(160ns) {S, Cin, A, B} = {S, Cin, A, B} + 1'b1;
        end
    always @(posedge clk, reset) //the good (expected result)
        case({S, Cin})
            3'b000 : {ExpectedCout, ExpectedD} = A + B;
            3'b001 : {ExpectedCout, ExpectedD} = A + B + 1;
            3'b010 : {ExpectedCout, ExpectedD} = A + ~B;
            3'b011 : {ExpectedCout, ExpectedD} = A + ~B + 1;
            3'b100 : {ExpectedCout, ExpectedD} = A;
            3'b101 : {ExpectedCout, ExpectedD} = A + 1;
            3'b110 : {ExpectedCout, ExpectedD} = A - 1;
            3'b111 : {ExpectedCout, ExpectedD} = A;
            endcase
```

```
endmodule

module designAnalyzer(S, ExpectedResult, ExpectedCout, DesignResult, DesignCout, clk,
reset);

        input [3:0]ExpectedResult, DesignResult;

        input S, ExpectedCout,DesignCout, clk, reset;

        always @(posedge clk, reset)

                begin

                        if(ExpectedResult != DesignResult)

                                        $display ("The expected result = %h does not equal
result from design = %h at time =%0d", ExpectedResult, DesignResult, $time);

                                if (S != 2'b01 && S != 2'b11)//when S = 1 or 3 cout is not good
and Complemeted

                                        begin

                                                if(ExpectedCout != DesignCout)//there is an
small error when goes to S = 1 or 3 we can avoid it in the future

                                                        $display ("The good (expected)
carry out = %b does not equal carry out from design = %b at time =%0d", ExpectedCout,
DesignCout, $time);

                                        end

                end

endmodule

module ALUStage1Verification;

        reg clk;

        reg reset;

        reg [3:0]A, B;

        reg [1:0]S;

        reg Cin;

                initial

                clk = 0;

        always #75ns clk = ~clk;

        wire [3:0]GlitchyResult;//without register

        wire [3:0]ExpectedResult;

        wire [3:0]TheResult;//the result with register

        wire CoutWithGlitch;
```

```
	wire ExpectedCout;

	wire TheCout;

	TestGeneraterForALU TGS1(A, B, S, Cin, ExpectedResult, ExpectedCout, clk, 
reset);

	Stage1RippleCarryAdderWithoutClock S1UT(A, B, S, Cin, GlitchyResult, 
CoutWithGlitch);//UT -> mean under test

	Stage1RippleCarryAdderWithClock S1UTWithRegister(A, B, S, Cin, TheResult, 
TheCout, clk, reset);

	designAnalyzer S1A(S, ExpectedResult, ExpectedCout, TheResult, TheCout, clk, 
reset);

endmodule

module CarryLookaheadAdder(A, B, Cin, Cout);

	input [3:0]A, B;

	input Cin;

	output [4:0]Cout;

	assign Cout[0] = Cin;

	wire [3:0]G, P;//propagate and generate functions

	wire [3:0]PandC;//wire for and between ci-1 and pi

	wire C2;

	wire [1:0]C3;

	wire [2:0]C4;

	genvar i;

	generate

	for(i = 0 ; i < 4 ; i = i + 1)

		begin:CLA

			and #(7ns) genand(G[i], A[i], B[i]);

			or #(7ns) proor(P[i], A[i], B[i]);

		end

	endgenerate

	and #(7ns) pandc(PandC[0], P[0], Cout[0]);

	or #(7ns) resor1(Cout[1], G[0], PandC[0]);

	and #(7ns) pandg2(C2, G[0], P[1]);

	and #(7ns) pandc2(PandC[1], P[0], P[1], Cout[0]);

	or #(7ns) resor2(Cout[2], G[1], C2, PandC[1]);
```

```
        and #(7ns) pandg3a(C3[0], G[1], P[2]);
        and #(7ns) pandg3b(C3[1], P[2], P[1], G[0]);
        and #(7ns) pandc3(PandC[2], P[2], P[1], P[0], Cout[0]);
        or #(7ns) resor3(Cout[3], G[2], C3[0], C3[1], PandC[2]);
        and #(7ns) pandg4a(C4[0], G[2], P[3]);
        and #(7ns) pandg4b(C4[1], P[3], P[2], G[1]);
        and #(7ns) pandg4c(C4[2], P[3], P[2], P[1], G[0]);
        and #(7ns) pandc4(PandC[3], P[3], P[2], P[1], P[0], Cout[0]);
        or #(7ns) resor4(Cout[4], G[3], C4[0], C4[1], C4[2], PandC[3]);
endmodule
module testbenchCLA;
        reg [3:0]A, B;
        reg Cin;
        wire [4:0]Cout;
        CarryLookaheadAdder CLA(A, B, Cin, Cout);
        initial
                begin
                        {A, B, Cin} = 9'b0000_00000;
                        repeat(512)
                        #(40ns) {A, B, Cin} = {A, B, Cin} + 1;
                end
endmodule
module FourBitCarryLookAheadFullAdder(X, Y, Cin, D, Cout);
        input [3:0]X, Y;
        input Cin;
        wire [4:0]Cpre;
        assign Cpre[0] = Cin, Cout = Cpre[4];
        output [3:0]D;
        output Cout;
        CarryLookaheadAdder CLA(X, Y, Cin, Cpre);
        genvar i;
        generate
        for(i = 0 ; i < 4 ; i = i + 1)
                begin:IFA//IFA -> mean instantiate full adder for four times
                        FullAdder Stage({Y[i], X[i]}, Cpre[i], D[i], Cpre[i+1]);
                end
                endgenerate
endmodule
module testbench4bitCLA;//testbench for 4-bit CLA ADDER
        reg [3:0]X, Y;
        reg Cin;
        wire [3:0]D;
        wire Cout;
        FourBitCarryLookAheadFullAdder FLAA(X, Y, Cin, D, Cout);
        initial
                begin
                        {X, Y, Cin} = 9'b0000_11111;
                        repeat(511)
                        #50ns {X, Y, Cin} = {X, Y, Cin} + 1;
                end
endmodule
module Stage2CarryLookAheadAdderWithoutClock(A, B, S, Cin, D, Cout);
        input [3:0]A, B;
        input [1:0]S;
        input Cin;
        output [3:0]D;
        output Cout;
        wire [3:0]NotB;
        wire [3:0]MuxO;//MuxO mean the output of the mux
        not #(3ns)(NotB[0], B[0]);
        not #(3ns)(NotB[1], B[1]);
        not #(3ns)(NotB[2], B[2]);
        not #(3ns)(NotB[3], B[3]);
        genvar i;
        generate
        for(i = 0 ; i < 4 ; i = i + 1)
                begin :buildALU
                        MUX4x1 MUX({1'b1, 1'b0, NotB[i], B[i]}, S, MuxO[i]);
                end
                endgenerate
                FourBitCarryLookAheadFullAdder FLAA(A, B, Cin, D, Cout);
endmodule
module Stage2CarryLookAheadAdderWithClock(A, B, S, Cin, DA, CoutA, clk, reset);
        input [3:0]A, B;
        wire [3:0]AC, BC;
        input [1:0]S;
        wire [1:0]SC;
        input Cin, clk, reset;
        wire CinC;
        wire [3:0]D;
        output reg [3:0] DA;
        output reg CoutA;
        wire Cout;
        wire [3:0]NotB;
        wire [3:0]MuxO;//MuxO mean the output of the mux
        not #(3ns)(NotB[0], BC[0]);
        not #(3ns)(NotB[1], BC[1]);
        not #(3ns)(NotB[2], BC[2]);
        not #(3ns)(NotB[3], BC[3]);
        wire [3:0]MUXR;
        Register RA(A, clk, reset, AC);
        Register RB(B, clk, reset, BC);
        Register #(.n(2))RS(S, clk, reset, SC);
        Register RCin(Cin, clk, reset, CinC);
        defparam RCin.n = 1;
        FourBitCarryLookAheadFullAdder SystemFA(AC, MuxO, CinC, D, Cout);
        Register RD(D, clk, reset, DA);
        Register #(.n(1))RCout(Cout, clk, reset, CoutA);
        genvar i;
        generate
        for(i = 0 ; i < 4 ; i = i + 1)
                begin :buildALU
                        MUX4x1 MUX({1'b1, 1'b0, NotB[i], BC[i]}, SC, MuxO[i]);//SC***
                end
                endgenerate
endmodule
module testbenchCLAwithoutclock;//testbench for CLA
        reg [3:0]A, B;
        reg [1:0]S;
        reg Cin;
        wire [3:0]D;
        wire Cout;
        Stage2CarryLookAheadAdderWithoutClock S2(A, B, S, Cin, D, Cout);
        initial
                begin
                        {S, Cin,A, B} = 11'b00100001111;
                        repeat(2048)
                        #80ns {S, Cin,A, B} = {S, Cin,A, B} + 1'b1;
                end
endmodule
module testbenchCLAwithclock;//testbench for Ripple-carry adder
        reg [3:0]A, B;
        reg [1:0]S;
        reg Cin, clock, reset;
        wire [3:0]D;
        wire Cout;
        always #63ns clock = ~clock;
        Stage2CarryLookAheadAdderWithClock Stage(A, B, S, Cin, D, Cout, clock, reset);
        initial
                begin
                        clock = 0;
                        {S, Cin, A, B} = 11'b00000000000;
                        repeat(2048)
                        #126ns {S, Cin, A, B} = {S, Cin, A, B} + 1'b1;
                end

endmodule//test bench for the stage with input clocked
module ALUStage2Verification;
        reg clk;
        reg reset;
        reg [3:0]A, B;
        reg [1:0]S;
        reg Cin;
        initial
                clk = 0;
        always #63ns clk = ~clk;
        wire [3:0]GlitchyResult;//without register
        wire [3:0]ExpectedResult;
        wire [3:0]TheResult;//D with register
        wire CoutWithGlitch;
        wire ExpectedCout;
        wire TheCout;
        TestGeneraterForALU TGS1(A, B, S, Cin, ExpectedResult, ExpectedCout, clk,
reset);
        Stage2CarryLookAheadAdderWithoutClock S2UT(A, B, S, Cin, GlitchyResult,
CoutWithGlitch);//UT -> mean under test
        Stage2CarryLookAheadAdderWithClock S2UTWithRegister(A, B, S, Cin,
TheResult, TheCout, clk, reset);
        designAnalyzer S1A(S, ExpectedResult, ExpectedCout, TheResult, TheCout, clk,
reset);
endmodule
```