# ΨΗΦΙΑΚΑ ΚΥΚΛΩΜΑΤΑ & ΣΥΣΤΗΜΑΤΑ(Εργαστήριο)

Ηρακλείδης Αλέξανδρος

1078522

2ο έτος

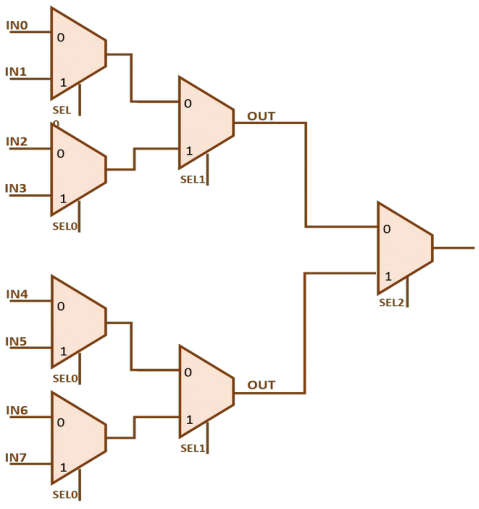
ΕΡΓΑΣΤΗΡΙΟ 2

ΑΣΚΗΣΗ 1

Στην πρώτη άσκηση του παραδοτέου χρειάζεται η σύνταξη κώδικα σε επίπεδο δομής που περιγράφει τη χρήση πολυπλέκτη 8 σε 1 με χρήση πολυπλέκτων 2 σε 1 όπως και το μοντέλο δοκιμής του.

```
module mux81(Y,I,S);
output Y;
input [7:0] I;
input [2:0] S;
wire [6:1] K ;
mux_case vas1(K[1],I[1:0],S[0]);
mux_case vas2(K[2],I[3:2],S[0]);
mux_case vas3(K[3],I[5:4],S[0]);
mux_case vas4(K[4],I[7:6],S[0]);
mux_case vas5(K[5],K[2:1],S[1]);
mux_case vas6(K[6],K[4:3],S[1]);
mux_case vas7(Y,K[6:5],S[2]);
endmodule

module mux_case(Y,I,S);
output reg Y;
input [1:0] I;
input S;
always @ (*)
        casex(S)
                1'b0: Y=I[0];
                1'b1: Y=I[1];
                default : Y = 1'bx;
        endcase
endmodule
```

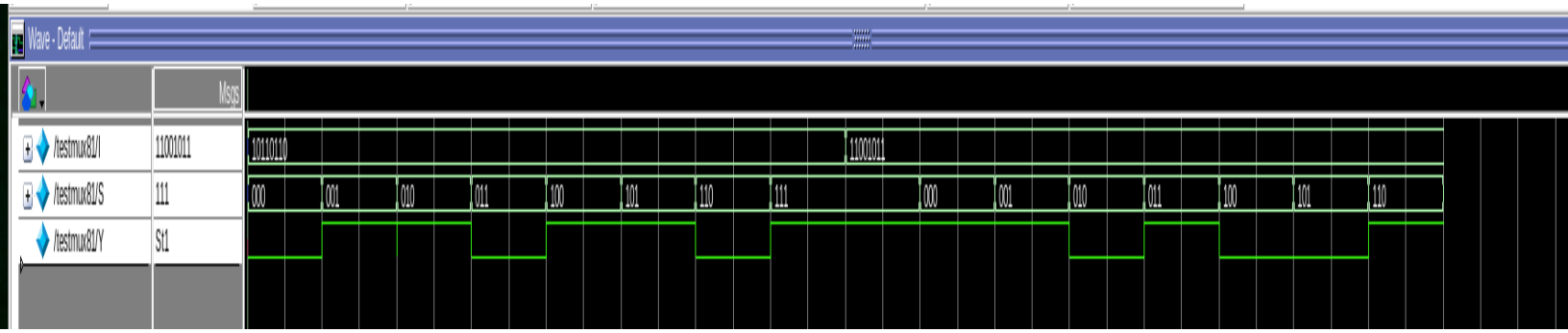
Στα δεξιά φαίνεαι ο πηγαίος κώδικας για τον πολυπλέκτη 8 σε 1 με χρήση πολυπλέκτων 2 σε 1.Πάνω φαίνεται η μονάδα του πολυπλέκτη 8 σε 1 ενώ κάτω βρίσκεται η υπομονάδα του πολυπλέκτη 2 σε 1 η οποία είναι γραμμένη σε επίπεδο συμπεριφοράς. Για τον πολυπλέκτη 8 σε 1 χρησιμοποίησα κώδικα σε επίπεδο δομής όπου δήλωσα είσοδα I μήκους 8 bits, selector S μήκους 3 bits, έξοδο Y, και μεταβλητή K τύπου wire μήκους 6 bit. Δήλωσα μετά 7 instances του πολυπλέκτη 2 σε 1 με ανάλογες εισόδους και εξόδους όπως φαίνεται στο πιο κάτω διάγραμμα όπου δεξία είναι η έξοδος Y.



```
module testmux81;
reg [7:0] I;
reg [2:0] S;
wire Y;
mux81 CUT1(Y,I,S);
initial
begin
I = 8'b10110110;
   S = 3'b000;
#2 S = 3'b001;
#2 S = 3'b010;
#2 S = 3'b011;
#2 S = 3'b100;
#2 S = 3'b101;
#2 S = 3'b110;
#2 S = 3'b111;
#2 I = 8'b11001011;
#2 S = 3'b000;
#2 S = 3'b001;
#2 S = 3'b010;
#2 S = 3'b011;
#2 S = 3'b100;
#2 S = 3'b101;
#2 S = 3'b110;
#2 S = 3'b111;
end
initial $monitor("time=%g,Y=%b,S=%b,I=%b",$time,Y,S,I);
endmodule
```

Εδώ φαίνεται ο κώδικας δοκιμής του πολυπλέκτη 8 σε 1 με χρήση πολυπλέκτων 2 σε 1.

Χρησιμοποίησα 2 τυχαίες τιμές εισόδου I 10110110 και 11001011 και εξέτασα την έξοδο Y για όλες τις πιθανές τιμές των εισόδων S για τις δύο περιπτώσεις.

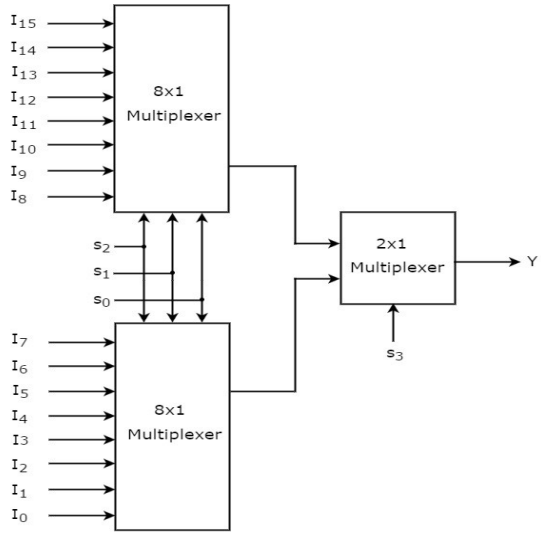



Πιο πάνω φαίνεται το αποτέλεσμα του simulatin του κώδικα δοκιμής.

Παρατηρούμε ότι ανάλογα με τις τιμές των selector η έξοδος Y αντιπροσωπεύει την αντίστοιχη είσοδο I. Για παράδειγμα στην πρώτη τυχαία τιμή I όταν τα S παίρνουν τιμή 110 (δηλαδή 6 σε δεκαδική μορφή η έξοδος είναι μηδέν δηλαδή ίση με το έκτο bit της εισόδου.

```
module mux161(
          input [15:0] I,
          input [3:0] S,
          output Y
          );
wire [1:0] T;
mux81 MUXA(T[0],I[7:0],S[2:0]);
mux81 MUXB(T[1],I[15:8],S[2:0]);
mux_case MUXC(Y,T[1:0],S[3]);
endmodule
module mux81(Y,I,S);
output Y;
input [7:0] I;
input [2:0] S;
wire [6:1] K ;
mux_case MUX1(K[1],I[1:0],S[0]);
mux_case MUX2(K[2],I[3:2],S[0]);
mux_case MUX3(K[3],I[5:4],S[0]);
mux_case MUX4(K[4],I[7:6],S[0]);
mux_case MUX5(K[5],K[2:1],S[1]);
mux_case MUX6(K[6],K[4:3],S[1]);
mux_case MUX7(Y,K[6:5],S[2]);
endmodule

module mux_case(Y,I,S);
output reg Y;
input [1:0] I;
input S;
always @ (*)
          casex(S)
                    1'b0: Y=I[0];
                    1'b1: Y=I[1];
                    default : Y = 1'bx;
          endcase
endmodule
```

Εδώ φαίνεται ένας πολυπλέκτης 16 σε 1 σε επίπεδο συμπεριφοράς. Χρησιμοποίησα μόνο 4 γραμμές κώδικα εφόσον αξιοποίησα την λειτουργία instantiantion της verilog. Ο πολυπλέκτης δομείται απο 2 πολυπλέκτες 8 σε 1 και 1 πολυπλέκτη 2 σε 1 σε διάταξη όπως φαίνεται στο σχήμα. Ο υπόλοιπος κώδικας είναι απλά ενδεικτικά τα δύο άλλα modules που χρησιμοποίησα του πολυπλέκτη 8 σε 1 και 2 σε 1 απο την προηγούμενη άσκηση.




```
module testmux161;
reg [15:0] I;
reg [3:0] S;
wire Y;
mux161 CUT1(I,S,Y);
initial
begin
I = 8'b10110110;
          S = 4'b0000;
#2        S = 3'b0001;
#2        S = 3'b0010;
#2        S = 3'b0011;
#2        S = 3'b0100;
#2        S = 3'b0101;
#2        S = 3'b0110;
#2        S = 3'b0111;
#2        S = 3'b1000;
#2        S = 3'b1001;
#2        S = 3'b1010;
#2        S = 3'b1011;
#2        S = 3'b1100;
#2        S = 3'b1101;
#2        S = 3'b1110;
#2        S = 3'b1111;
end
initial $monitor("time=%g, Y=%b , S=%b , I=%b",$time,Y,S,I);
```

Μια απλή παραλλαγή του προηγούμενου μοντέλου δοκιμής για να φαίνεται η ορθή λειτουργια του πολυπλέκτη 16 σε 1 για μία τιμή του I.

Πιο κάτω φαίνεται το αποτέλεσμα του simulation.

# ΑΣΚΗΣΗ 2

-Στην δεύτερη άσκηση μας ζητείται να συντάξουμε μοντέλο σε επίπεδο συμπεριφοράς ενός κωδικοποιητή προτεραιότητας 4 σε 2 και το αντοίστιχο μοντέλο δοκιμής έτσι ώστε να μπορούμε να επιβεβαιώσουμε την σωστή λειτουργία του.

```
module pcoder(D3,D2,D1,D0,Y1,Y0,Z);

input D3,D2,D1,D0;
output reg Y1,Y0,Z;

always @(*)
          case ({D3,D2,D1,D0})
                    4'b0000: {Y1,Y0,Z} = 3'b001;
                    4'b0001: {Y1,Y0,Z} = 3'b000;
                    4'b0010: {Y1,Y0,Z} = 3'b010;
                    4'b0011: {Y1,Y0,Z} = 3'b010;
                    4'b0100: {Y1,Y0,Z} = 3'b100;
                    4'b0101: {Y1,Y0,Z} = 3'b100;
                    4'b0110: {Y1,Y0,Z} = 3'b100;
                    4'b0111: {Y1,Y0,Z} = 3'b100;
                    4'b1000: {Y1,Y0,Z} = 3'b110;
                    4'b1001: {Y1,Y0,Z} = 3'b110;
                    4'b1010: {Y1,Y0,Z} = 3'b110;
                    4'b1011: {Y1,Y0,Z} = 3'b110;
                    4'b1100: {Y1,Y0,Z} = 3'b110;
                    4'b1101: {Y1,Y0,Z} = 3'b110;
                    4'b1110: {Y1,Y0,Z} = 3'b110;
                    4'b1111: {Y1,Y0,Z} = 3'b110;
          endcase
endmodule
```

Πηγαίος κώδικας που χρησιμοποίησα.

4 εισόδοι D0-D3

3 εξόδοι Y1 Y0 Z

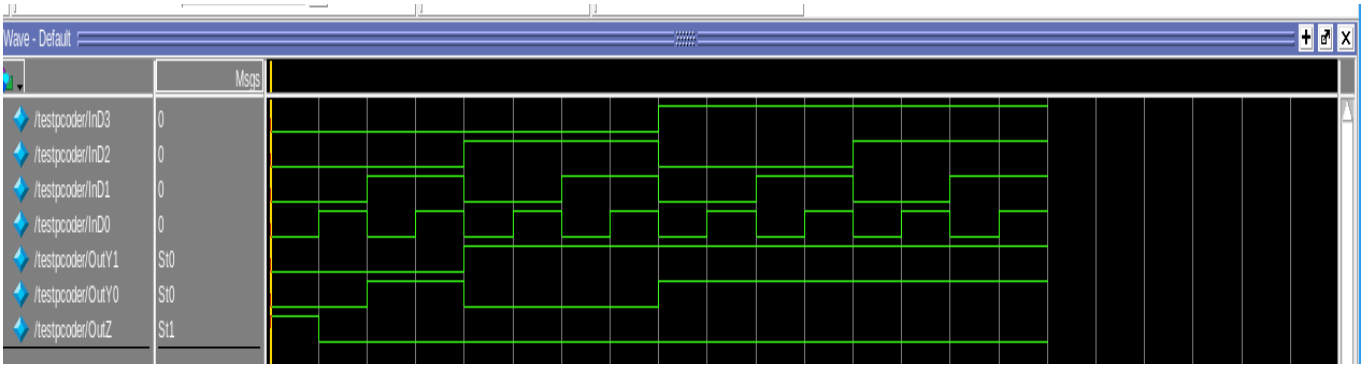
Δήλωσα για κάθε συνδυασμό εισόδων την κατάλληλη έξοδο που θα πρέπει να έχει η κάθε έξοδος με την μέθοδο case.

```
module testpcoder;
  reg InD3,InD2,InD1,InD0;
  wire OutY1,OutY0,OutZ;

  pcoder CUT1(InD3,InD2,InD1,InD0,OutY1,OutY0,OutZ);

  initial
  begin
          {InD3,InD2,InD1,InD0}  = 4'b0000;
#10       {InD3,InD2,InD1,InD0}  = 4'b0001;
#10       {InD3,InD2,InD1,InD0}  = 4'b0010;
#10       {InD3,InD2,InD1,InD0}  = 4'b0011;
#10       {InD3,InD2,InD1,InD0}  = 4'b0100;
#10       {InD3,InD2,InD1,InD0}  = 4'b0101;
#10       {InD3,InD2,InD1,InD0}  = 4'b0110;
#10       {InD3,InD2,InD1,InD0}  = 4'b0111;
#10       {InD3,InD2,InD1,InD0}  = 4'b1000;
#10       {InD3,InD2,InD1,InD0}  = 4'b1001;
#10       {InD3,InD2,InD1,InD0}  = 4'b1010;
#10       {InD3,InD2,InD1,InD0}  = 4'b1011;
#10       {InD3,InD2,InD1,InD0}  = 4'b1100;
#10       {InD3,InD2,InD1,InD0}  = 4'b1101;
#10       {InD3,InD2,InD1,InD0}  = 4'b1110;
#10       {InD3,InD2,InD1,InD0}  = 4'b1111;
#10        $stop;
end

initial $monitor($time ,  , "InD3 = %b, InD2=%b,InD1=%b,InD0=%b,
        OutY1=%b,OutY0=%b,OutZ=%b",InD3,InD2,InD1,InD0,OutY1,OutY0,OutZ);
endmodule
```

Το testbench για το κωδικοποιητή προτεραιότητας.

Χρησιμοποίησα αυτόματη αντιστοίχηση θηρών

και έπειτα ανάθεσα αυξανόμενους σε συνάρτηση με το χρόνο δυαδικούς συνδυασμούς των εισόδων.




Simulation του testbench για τον κωδικοποιήτη προτεραίοτητας μας επιβεβαιώνει την ορθή λειτουργία του. Η έξοδος Zeros παίρνει τιμή μόνο όταν όλες οι εισόδοι είναι μηδέν ενώ οι δύο εξόδοι Y0 , Y1 αναπαριστούν με διψήφια δυαδική τιμή τη θέση του MSB(Most Significant Bit) με είσοδος που έχει τιμή 1.π.χ. Όταν η είσοδος είναι 1010 τα Y0, Y1 είναι αντίστοιχα 1,1 δείχνοντας ότι η είσοδος D3 είναι η πιό σημαντική ενεργή είσοδος.

## ΑΣΚΗΣΗ 3

Σε αυτή την άσκηση ζητείται μοντέλο σε επίπεδο συμπεριφοράς και μοντέλο σε επίπεδο ροής δεδομένων ενός κωδικοποιητή 8 σε 3 και τα αντίστοιχα testbench για να ελέγξουμε την σωστή λειτουργία των δύο μοντέλων.

```
module boolencd83(D0,D1,D2,D3,D4,D5,D6,D7,Y2,Y1,Y0);

input D0,D1,D2,D3,D4,D5,D6,D7;
output Y2,Y1,Y0;

assign Y2 = D7|D6|D5|D4;
assign Y1 = D7|D6|D3|D2;
assign Y0 = D7|D5|D3|D1;
endmodule
```

Σε επίπεδο ροής δεδομένων φαίνεται ο πηγάιος κώδικας στα αριστερά.

8 Εισόδοι D0-D7

3 Εξόδοι Y2,Y1,Y0

3 λογικές συναρτήσεις, μία για κάθε έξοδο. π.χ Y2 = D7 | D6 | D5 | D4

Η έξοδος Y2 παίρνει τιμή θετική όταν μία απο τις πύλες D7,D6,D5,D4 έχει τιμή 1. Με το ίδιο σκεπτικό έκανα και τις άλλες δύο λογικές συναρτήσεις.

```
module behenc83 (D,Y2,Y1,Y0);
input [0:7] D;
output reg Y2,Y1,Y0;
always @(D)
begin
        case(D)
                8'b10000000:{Y0,Y1,Y2} = 3'b000;
                8'b01000000:{Y0,Y1,Y2} = 3'b001;
                8'b00100000:{Y0,Y1,Y2} = 3'b010;
                8'b00010000:{Y0,Y1,Y2} = 3'b011;
                8'b00001000:{Y0,Y1,Y2} = 3'b100;
                8'b00000100:{Y0,Y1,Y2} = 3'b101;
                8'b00000010:{Y0,Y1,Y2} = 3'b110;
                8'b00000001:{Y0,Y1,Y2} = 3'b111;
                default : {Y0,Y1,Y2} = 3'b111;
        endcase
end
endmodule
```

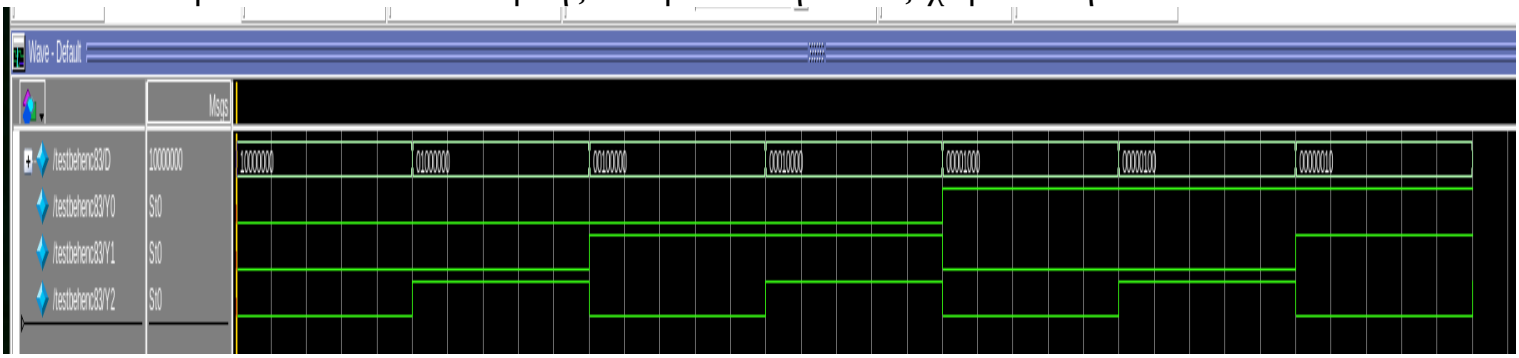
Σε επίπεδο συμπεριφοράς χρησιμοποίησα την ίδια τεχνική case με την άσκηση 2 όπου αντιστοίχησα τις εισόδους D με τις ανάλογες τιμές εξόδου.

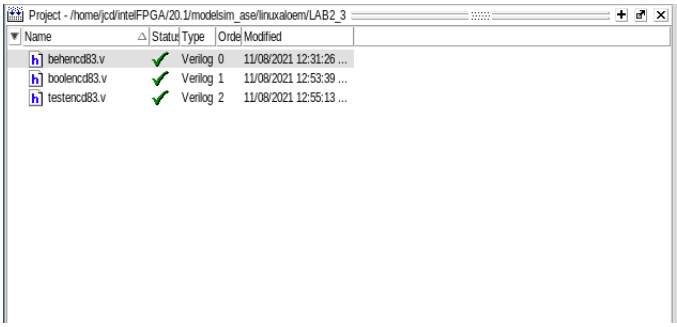
Τα δύο μοντέλα δοκίμης που χρησιμοποιήσα. Το μοντέλο για το επίπεδο συμπεριφοράς το έγραψα πιό μετα για αυτό αξιοποίο πιο efficient κώδικα. Στην ουσία όμως τα δυο μοντέλα έχουν την ίδια λειτουργία απλά το ένα είναι γραμμένο με πιό λίγες γραμμές κώδικα.

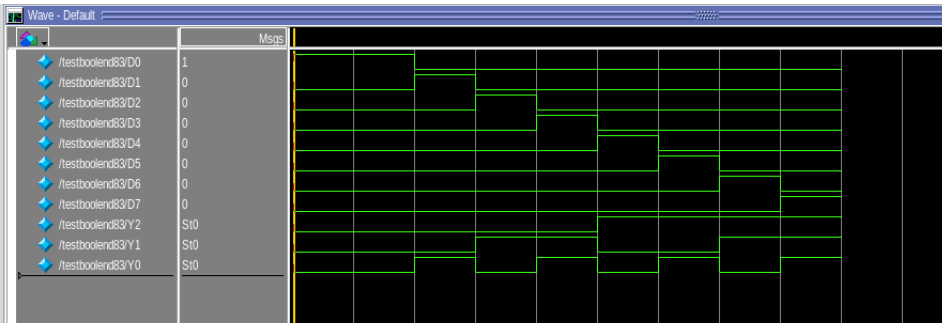
```
module testbehenc83;
reg [0:7] D;
wire Y0,Y1,Y2;

behenc83 CUT1(D,Y2,Y1,Y0);

initial
begin
        D = 8'b10000000;
#10     D = 8'b01000000;
#10     D = 8'b00100000;
#10     D = 8'b00010000;
#10     D = 8'b00001000;
#10     D = 8'b00000100;
#10     D = 8'b00000010;
#10     D = 8'b00000001;
end
endmodule
```

```
module testboolencd83;
reg D0,D1,D2,D3,D4,D5,D6,D7;
wire Y2,Y1,Y0;

boolencd83 CUT1(D0,D1,D2,D3,D4,D5,D6,D7,Y2,Y1,Y0);

initial
begin
D0 = 1;
D1 = 0;
D2 = 0;
D3 = 0;
D4 = 0;
D5 = 0;
D6 = 0;
D7 = 0;
#100;
#100;D0 = 0;D1 = 1;D2 = 0;D3 = 0;D4 = 0;D5 = 0;D6 = 0;D7 = 0;
#100;D0 = 0;D1 = 0;D2 = 1;D3 = 0;D4 = 0;D5 = 0;D6 = 0;D7 = 0;
#100;D0 = 0;D1 = 0;D2 = 0;D3 = 1;D4 = 0;D5 = 0;D6 = 0;D7 = 0;
#100;D0 = 0;D1 = 0;D2 = 0;D3 = 0;D4 = 1;D5 = 0;D6 = 0;D7 = 0;
#100;D0 = 0;D1 = 0;D2 = 0;D3 = 0;D4 = 0;D5 = 1;D6 = 0;D7 = 0;
#100;D0 = 0;D1 = 0;D2 = 0;D3 = 0;D4 = 0;D5 = 0;D6 = 1;D7 = 0;
#100;D0 = 0;D1 = 0;D2 = 0;D3 = 0;D4 = 0;D5 = 0;D6 = 0;D7 = 1;
#100 $stop;
end
initial $monitor($time , , "D0 = %b,D1=%b,D2=%b,D3=%b,D4=%b,D5=%b,D6=%b,D7=%b,Y2=%b,Y1=%b,Y0=%b",D0,D1,D2,D3,D4,D5,D6,D7,Y2,Y1,Y0);

endmodule
```

Πιο κάτω φαίνονται τα δύο αποτελέσματα του simulation των testbench. Το πάνω είναι σε επίπεδο συμπεριφοράς όπου χρησιμοποίησα εισόδους με πολλαπλά ψηφία ενώ στο κάτω μοντέλο στο επίπεδο ροης δεδομένων δήλωσα ξεχωριστά την κάθε είσοδο.




# ΑΣΚΗΣΗ 4

Αρχικά ζητείται να σχεδιάσουμε μοντέλο verilog ενός αθροιστή carry look ahead (CLA) μήκος τεσσάρων bit. Ο λόγος που χρησιμοποιούμε ένα CLA και όχι Ripple Carry Adder είναι γιατί μείωνεται το propagation delay σε ακολουθιακό κύκλωμα διότι ο επόμενος (n) CLA δεν χρειάζεται να περιμένει για το carry του προηγούμενου CLA (n-1) για να υπολογίσει το αποτέλεσμα.

```
module cla4(
    output [3:0] S,
    output Cout,
    input [3:0] A,B,
    input Cin
    );
    wire [3:0] G,P,C;
    assign G = A & B;
    assign P = A ^ B;
    assign C[0] = Cin;
    assign C[1] = G[0] | (P[0] & C[0]);
    assign C[2] = G[1] | (P[1] & G[0]) | (P[1] & P[0] & C[0]);
    assign C[3] = G[2] | (P[2] & G[1]) | (P[2] & P[1] & G[0]) |            (P[2] & P[1] & P[0] & C[0]);
    assign Cout = 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]);
    assign S = P ^ C;
endmodule
```

Πιό πάνω είναι ο πηγαίος κώδικας του αθροιστή carry look ahead μήκος 4 bit σε επίπεδο συμπεριφοράς. Δήλωσα δύο εξόδους και τρείς εισόδους. Οι τρείς εισόδοι είναι οι δύο 4 bit αριθμοί που θα αθροιστούν A B, και το carry Cin ενός bit . Οι δύο εξόδοι είναι το S όπου αντιπροσωπεύει το sum μήκος 4 bit και το carry out Cout. Επίσης δήλωσα μεταβλητές τύπου wire. G είναι ο carry generator και P ο carry propagator. Απο τις λογικές συναρτήσεις που δίνονται στο αρχείο αναφοράς για C1,C2,C3,C4 = Cout, υπολογίζω τα carry και τελικά απο την λογική συνάρτηση S = P ^ C

(P XOR C) βρίσκω το sum των δύο αριθμών.

```
module select8(
        output [7:0] S,
        output Cout,
        input [7:0] A,B,
        input Cin
        );
        wire C0,C1,CADD1,CADD2,CADD3;
        wire [3:0] S0,S1,A_1,B_1,A_2,B_2,SL,SH;
        assign C0 = 1'b0;
        assign C1 = 1'b1;
        assign A_1={A[3],A[2],A[1],A[0]};
        assign B_1={B[3],B[2],B[1],B[0]};
        assign A_2={A[7],A[6],A[5],A[4]};
        assign B_2={B[7],B[6],B[5],B[4]};

        cla4 ADD1(SL,CADD1,A_1,B_1,Cin);
        cla4 ADD2(S0,CADD2,A_2,B_2,C0);
        cla4 ADD3(S1,CADD3,A_2,B_2,C1);
        mux21 MUX(S0,S1,CADD1,SH);

        assign S = {SH,SL};
        assign Cout = CADD1?CADD3:CADD2;
        endmodule

module cla4(
    output [3:0] S,
    output Cout,
    input [3:0] A,B,
    input Cin
    );
    wire [3:0] G,P,C;
    assign G = A & B;
    assign P = A ^ B;
    assign C[0] = Cin;
    assign C[1] = G[0] | (P[0] & C[0]);
    assign C[2] = G[1] | (P[1] & G[0]) | (P[1] & P[0] & C[0]);
    assign C[3] = G[2] | (P[2] & G[1]) | (P[2] & P[1] & G[0]) |                (P[2] & P[1] & P[0] & C[0]);
    assign Cout = 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]);
    assign S = P ^ C;
endmodule

module mux21(I0,I1,S,OUT);
input[3:0]I0,I1;
input S;
output[3:0]OUT;
assign OUT=S?I1:I0;
endmodule
```

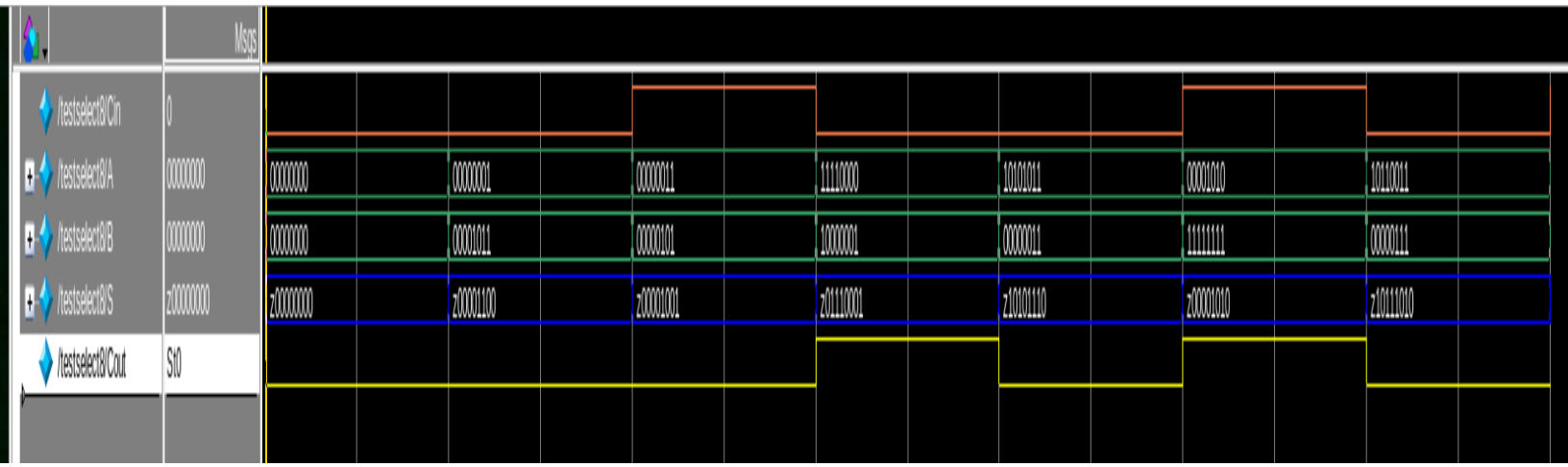
Εδώ φαίνεται ο κώδικας για το module select8. Χρησιμοποίησα τρία modules cla4 και δύο πολυπλέκτες 2 σε 1. Ο πρώτος cla4 υπολογίζει το sum και το carry των τεσσάρων λιγότερο σημαντικών ψηφίων απο τους δύο αριθμούς. Τα δύο άλλα παίρνουν δύο διαφορετικές περιπτώσεις,το ένα προσθέτει τα τέσσερα περισσότερο σημαντικά ψηφία αν θεωρήσει οτι το carry input (Cin) είναι 0 kai το δεύτερο cla4 παίρνει την περίπτωση όπου το carry input ισούται με 1. Έπειτα ο πολυπλέκτης παίρνει ως εισόδους τα δύο sum των τελευτάιων cla4 και τιμή selector το carry που έχει υπολογίσει ο πρώτος cla4. Αυτό γίνεται για να μην χαθεί χρόνος περιμένοντας να υπολογιστεί το carry των τεσσαρων LSB για να υπολογιστεί το sum των MSB.

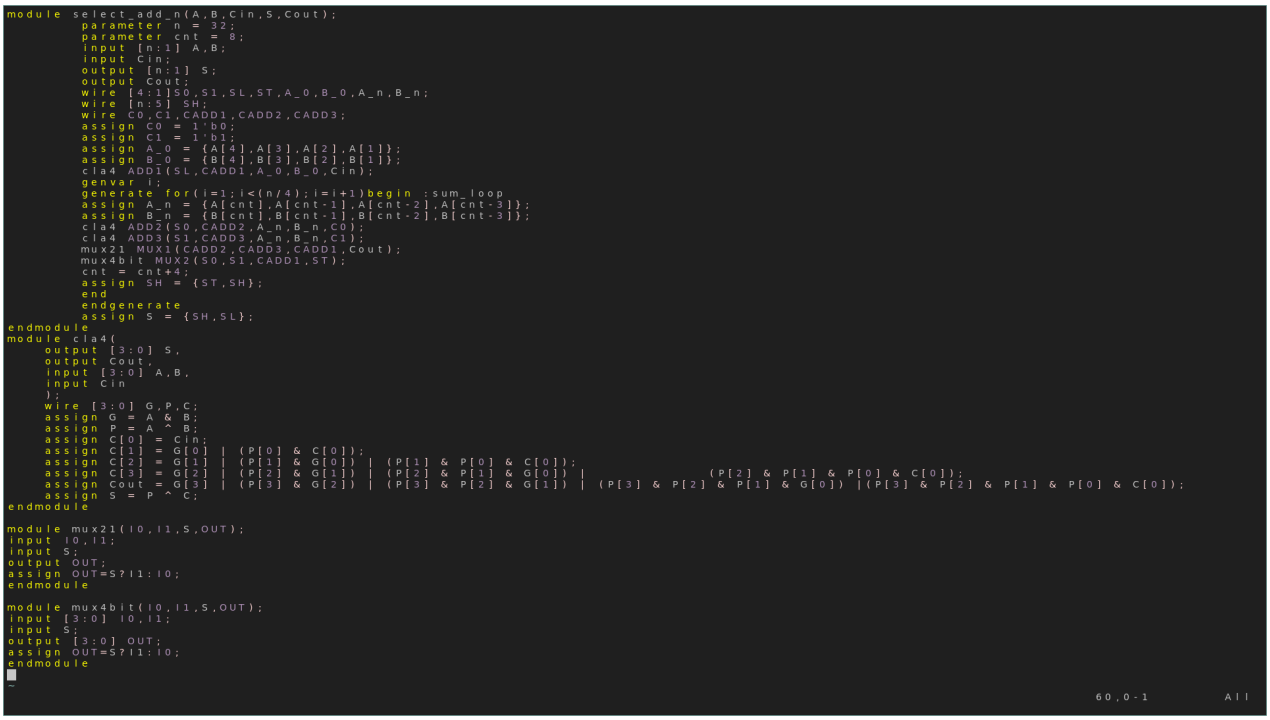
Τέλος ενώνουμε τα δυο sum high και sum low για να βρούμε το συνολικό sum και το υπόλοιπο Carry out υπολογίζεται απο την λογική συνάρτηση Cout = CADD1(Carry 1ου cla4)?CADD3(αντιστοίχως 3ου):CADD2(2ου).

```
module testselect8;
reg Cin;
reg [7:0] A,B;
wire [8:0] S;

select8 CUT1(S,Cout,A,B,Cin);
initial
begin
        A=0; B=0; Cin=0;
        #10
        A=8'b1; B=11; Cin=1'b0;
        #10
        A=8'b11; B=8'b101; Cin=1'b1;
        #10
        A=8'b11110000; B=8'b10000001; Cin=1'b0;
        #10
        A=8'hAB; B=8'h03; Cin=1'b0;
        #10
        A=8'h0A; B=8'hFF; Cin=1'b1;
        #10
        A=8'hB3; B=8'h07; Cin = 1'b0;
        #10
        A=8'b1; B = 8'b11; Cin = 1'b1;
end
endmodule
```

Το μοντέλο δοκιμής του module select8 όπου αρχικά μηδενίζω τα input και μετά τους δίνω δοκιμαστικές τιμές για να εξετάσω το αποτέλεσμα που μου δίνει όσο αφορά το sum και Carry out.

Πιο κάτω βλέπουμε το simulation του testbench και τις διαφορετικές τιμές που δίνουν οι εξόδοι ανάλογα με τα inputs.





```
module select_add_n(A,B,Cin,S,Cout);
        parameter n = 32;
        parameter cnt = 8;
        input [n:1] A,B;
        input Cin;
        output [n:1] S;
        output Cout;
        wire [4:1]S0,S1,SL,ST,A_0,B_0,A_n,B_n;
        wire [n:5] SH;
        wire C0,C1,CADD1,CADD2,CADD3;
        assign C0 = 1'b0;
        assign C1 = 1'b1;
        assign A_0 = {A[4],A[3],A[2],A[1]};
        assign B_0 = {B[4],B[3],B[2],B[1]};
        cla4 ADD1(SL,CADD1,A_0,B_0,Cin);
        genvar i;
        generate for(i=1;i<(n/4);i=i+1)begin :sum_loop
        assign A_n = {A[cnt],A[cnt-1],A[cnt-2],A[cnt-3]};
        assign B_n = {B[cnt],B[cnt-1],B[cnt-2],B[cnt-3]};
        cla4 ADD2(S0,CADD2,A_n,B_n,C0);
        cla4 ADD3(S1,CADD3,A_n,B_n,C1);
        mux21 MUX1(CADD2,CADD3,CADD1,Cout);
        mux4bit MUX2(S0,S1,CADD1,ST);
        cnt = cnt+4;
        assign SH = {ST,SH};
        end
        endgenerate
        assign S = {SH,SL};
endmodule
module cla4(
    output [3:0] S,
    output Cout,
    input [3:0] A,B,
    input Cin
    );
    wire [3:0] G,P,C;
    assign G = A & B;
    assign P = A ^ B;
    assign C[0] = Cin;
    assign C[1] = G[0] | (P[0] & C[0]);
    assign C[2] = G[1] | (P[1] & G[0]) | (P[1] & P[0] & C[0]);
    assign C[3] = G[2] | (P[2] & G[1]) | (P[2] & P[1] & G[0]) |                (P[2] & P[1] & P[0] & C[0]);
    assign Cout = 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]);
    assign S = P ^ C;
endmodule

module mux21(I0,I1,S,OUT);
input I0,I1;
input S;
output OUT;
assign OUT=S?I1:I0;
endmodule

module mux4bit(I0,I1,S,OUT);
input [3:0] I0,I1;
input S;
output [3:0] OUT;
assign OUT=S?I1:I0;
endmodule
~
                                                                    60,0-1          All
```

Εδώ είναι το module select_add_n για ν = 32bit. Το σχεδίασα με βάση την φωτογραφία που βρίσκεται στην αναφορά. Χρησιμοποιά ένα cla4 αρχικά και μετά ξεκινάει ένα loop όπου δημιουργεί δυο επιπλέον  instances cla4 ένα πολυπλέκτη 2 σε 1 ενός bit και ακόμα ενα 2 σε 1 πολυπλέκτη μήκος 4 bit.Το sum των πρώτων τεσσάρων ψηφίων υπολογίζεται πάντα απο τον πρώτο cla4 ενώ τα υπόλοιπα ψηφία του sum και το carry out υπολογίζεται απο το επαναλαμβανόμενο κύκλωμα.  Ο αρχικός cla4 παίρνει ως input τα πρώτα τεσσερα ψηφία των 32bit input για να υπολογίσει τα πρώτα τέσσερα ψηφία του sum ενώ το επαναλμβανόμενο κύκλωμα παίρνει πακέτα των 4 bit και τα προσθέτει ξεχωριστά μέχρι να βγούν όλα τα υπόλοιπα ψηφία του sum. Αυτός ο κώδικας είναι η προσπάθεια μου να αναπραστήσω το κύκλωμα όμως είχα δυσκολία με τα πακέτα 4 bit και τα iterating variables και γιαυτό δεν έκανε compile.

```
module testselect_add_32;
reg [n:1] A,B;
reg Cin;
wire [n:1] S;
wire Cout;

select_add_n CUT1(A,B,Cin,S,Cout);

initial
begin
            A=0; B=0; Cin=0;
            #10
            A=32'hBADFECA8; B=32'hA39;
end
endmodule
```

Επίσης έκανα ένα πρόχειρο testbench select_add_n για ενδεικτικούς σκοπούς.