ΨΗΦΙΑΚΑ ΣΥΣΤΗΜΑΤΑ HW ΣΕ ΧΑΜΗΛΑ ΕΠΙΠΕΔΑ ΛΟΓΙΚΗΣ ΙΙ

ΕΡΓΑΣΙΑ 2020/21

Ονοματεπώνυμο: Πορλού Χάιδω

ΑΕΜ: 9372

ΑΣΚΗΣΗ 1

Εισαγωγικά Συμπεράσματα & Ανάλυση του FSM

Στην άσκηση 1, το FSM του σχήματος αποτελεί τύπου Mealy FSM, καθώς η έξοδος y_out εξαρτάται και από την τρέχουσα κατάσταση αλλά και από την τιμή της εισόδου x_in, όπως φαίνεται από τις μεταβάσεις στο δοσμένο σχήμα.

α) Στο ερώτημα α καλούμαστε να περιγράψουμε το FSM με τον κλασσικό τρόπο, δηλαδή με συμπεριφορική Verilog.

Οι καταστάσεις του FSM είναι ήδη κωδικοποιημένες στο σχήμα, και η καθεμία χαρακτηρίζεται απο 3 bits. Ο λόγος για αυτό είναι πως έχουμε 5 κατστάσεις, άρα δεν μπορεί η κάθε κατάσταση να κωδικοποιηθεί με 2 bits (2^2=4 καταστάσεις max), άρα τα 3 bits ανά κατάσταση είναι η επόμενη λογική λύση (2^3=8 καταστάσεις, υπεραρκετές).

Για ευκολία και καλύτερη διαχείρηση των καταστάσεων στην υπόλοιπη άσκηση, έχω αντιστοιχήσει ονομασίες στην καθεμία, οι οποίες φαίνονται στον παρακάτω πίνακα.

Κατάσταση FSM	Κωδικοποίηση
S1	001
S2	011
S3	100
S4	010
S5	000

Εφόσον έχουμε 3 bits ανά κατάσταση, η Verilog χρησιμοποιεί για την κωδικοποίηση της καθεμίας 3 Flip Flop. Επίσης, δεν προχωράμε σε κωδικοποιήση για την είσοδο ή την έξοδο καθώς είναι ήδη γνωστές κ

αντιστοιχίζονται σε 1 bit. Μελετώντας το διάγραμμα του FSM, στη συνέχεια προχωράμε στην υλοποίηση του πίνακα μεταβάσεων αυτού:

Current State	Next State		Output (y_out)	
	x_in = 0	x_in = 1	x_in = 0	x_in = 1
S1	S1	S3	0	1
S2	S4	S2	0	0
S3	S1	S4	0	1
S4	S4	S5	0	1
S5	S2	S3	0	1

Γνωρίζοντας τα παραπάνω, το επόμενο βήμα είναι η εξαγωγή των λογικών εξισώσεων των καταστάσεων (κύκλωμα πριν τον καταχωρητή) και της λογικής εξίσωσης εξόδου (κύκλωμα μετά τον καταχωρητή) για την υλοποίηση του FSM. Παρακάτω παρουσιάζεται ο πίνακας αληθείας για το Next State.

Current State			Input	Next State		
D2	D1	D0	x_in	D2’	D1’	D0’
0	0	1	0	0	0	1
0	0	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
0	1	1	0	0	0	1
0	1	1	1	0	1	0
0	1	0	0	0	1	0
0	1	0	1	0	0	0
0	0	0	0	0	1	1
0	0	0	1	1	0	0

Χρησιμοποιώντας πίνακες καρνό με τις παραπάνω πληροφορίες, καταλήγουμε στις εξής εξισώσεις για τα bits του Next State:

$$D2' = \overline{D2D1X}$$

$$D1' = \overline{D2D0X} + D2\overline{D1D0} + \overline{D2}D1D0X$$

$$D0' = \overline{D2D1X} + \overline{D2}D0\overline{X} + D2\overline{D1D0}X$$

Παρόμοια, ο πίνακας αληθείας για την έξοδο, και η αντίστοιχη εξίσωση:

Current State			Input	Output
D2	D1	D0	x_in	y_out
0	0	1	0	0
0	0	1	1	1
1	0	0	0	0
1	0	0	1	0
0	1	1	0	0
0	1	1	1	1
0	1	0	0	0
0	1	0	1	1
0	0	0	0	0
0	0	0	1	1

$$y\_out = \overline{D2}X$$

Υλοποίηση module & Περιγραφή

Ο behavioural κώδικας που υλοποιήθηκε για την περιγραφή του παραπάνω FSM είναι ο εξής:

```
*

`timescale 1ns/1ns

module FSM_a(output reg y_out,

        input wire x_in, CLK, Reset);

reg [2:0] CurrentState, NextState;

parameter s1 = 3'b001,

      s2 = 3'b011,

      s3 = 3'b100,

      s4 = 3'b010,

      s5 = 3'b000;

always @(posedge CLK or posedge Reset)

  begin

    if(Reset)
```

```
      CurrentState <= s1;
    else
      CurrentState <= NextState;
  end

always @(CurrentState or x_in)
  begin
    case(CurrentState)
     s1 : if(x_in == 1'b0) NextState = s1;
          else NextState = s3;
     s2 : if(x_in == 1'b0) NextState = s1;
               else NextState = s4;
     s3 : if(x_in == 1'b0) NextState = s4;
               else NextState = s2;
     s4 : if(x_in == 1'b0) NextState = s4;
               else NextState = s5;
     s5 : if(x_in == 1'b0) NextState = s2;
               else NextState = s3;
     default : NextState = s1;
    endcase
  end

always @(CurrentState or x_in)
  begin
   case(CurrentState)
     s1 : if(x_in == 1'b0) y_out = 1'b0;
```

```
            else y_out = 1'b1;
      s2 : if(x_in == 1'b0) y_out = 1'b0;
            else y_out = 1'b1;
      s3 : y_out = 1'b0;
         s4 : if(x_in == 1'b0) y_out = 1'b0;
            else y_out = 1'b1;
         s5 : if(x_in == 1'b0) y_out = 1'b0;
            else y_out = 1'b1;
      default : y_out = 1'b0;
    endcase
  end
endmodule
*
```

Αρχικά, ορίζουμε το timescale ως 1ns/1ns, το οποίο στην συνέχεια θα χρησιμοποιηθεί για τον ορισμό της περιόδου του ρολογιού μας. Ορίσματα εισόδου του FSM αποτελούν η είσοδος (x_in), το ρολόι (CLK) και το reset, ως wires. Την έξοδο αποτελεί το y_out, με την μορφή reg.

Πριν ξεκινήσει η λειτουργία του FSM, ορίζονται σαν reg vectors (ή αλλιώς arrays) των 3 bit/θέσεων τα CurrentState και NextState, και στη συνέχεια η κάθε κατάσταση ορίζεται ως παράμετρος ώστε να μπορεί να αναφέρεται ευκολότερα και πιο ξεκάθαρα μέσα στην υλοποίηση.

Το πρώτο always block μας δίνει την κύρια λειτουργία του FSM, δηλαδή τη μετάβαση στην επόμενη κατάσταση σε κάθε θετική ακμή του ορισμένου ρολογιού (από 0 σε 1). Όσον αφορά το reset (που εδώ είναι active high), όταν δοθεί με τιμή 1, η τρέχουσα κατάσταση γίνεται η κατάσταση S1 (001), σύμφωνα με την εκφώνηση. Το παραπάνω υλοποιείται με το αντίστοιχο if.

Ο ρόλος του επόμενου always block είναι να ορισθούν οι επόμενες καταστάσεις με ορισμένες τις τρέχουσες καταστάσεις, όταν δοθεί είσοδος 1 ή 0. Το block «ενεργοποιείται» με την αλλαγή της τρέχουσας κατάστασης ή όταν δίνεται/αλλάζει η είσοδος x_in. Ο ορισμός των

επόμενων κασταστάσεων γίνεται μέσω ενός case block, το default του οποίου είναι η κατάσταση S1(001), καθώς εκεί οδηγεί και το reset.

Το τρίτο και τελευταίο block του behavioural FSM ορίζει την έξοδο του όταν δοθεί μια συγκεκριμένη (0 ή 1) είσοδος. Ο ορισμός της εξόδου υλοποιείται, ξανά, με ένα case block, και παίρνει την επιθημητή τιμή της ανάλογα με την τρέχουσα κατάσταση. Αξίζει να σημειωθεί ότι, επειδή πραγματευόμαστε ένα Mealy FSM, το συγκεκριμένο always block «ενεργοποιείται» όταν αλλάζει η τρέχουσα κατάσταση αλλά και όταν έχουμε μία νέα είσοδο, με αποτέλεσμα η εξόδος να μην καθυστερεί έναν κύκλο ρολογιού, όπως θα συνέβαινε με ένα Moore FSM.

Υλοποίηση testbench & Περιγραφή

Το testbench που υλοποιήθηκε για τον έλεγχο του FSM είναι το εξής:

*

```
`timescale 1ns/1ns

module FSM_testbench_a;

  reg CLK_TB;

  reg Reset_TB;

  reg x_in_TB;

  wire y_out_TB;


  integer i;

  reg Expected_y_out;

  reg [0:2] TestVector [0:18];


  FSM_a DUT(.CLK(CLK_TB),.Reset(Reset_TB),.x_in(x_in_TB),.y_out(y_out_TB));


initial

  begin

    CLK_TB = 1'b1;
```

```
  end


initial
  begin
    $readmemb("TestVector1a.txt",TestVector);
    i = 0;
    Reset_TB = 1;
    x_in_TB = 0;
  end


always
  begin
   #5 CLK_TB = ~CLK_TB;
  end


always @(posedge CLK_TB)
  begin
    {Reset_TB,x_in_TB,Expected_y_out} = TestVector[i]; #5
    $display(Reset_TB,x_in_TB,Expected_y_out);
  end


always@(negedge CLK_TB)
  begin
    if(Expected_y_out !== y_out_TB) begin
       $display("Wrong output for inputs %b, %b !=
%b",{Reset_TB,x_in_TB},Expected_y_out,y_out_TB);
```

```
    end

    i = i + 1;

  end

endmodule
```

*

Αρχικά, ορίζουμε το timescale 1ns, με step 1 ns για την προσομοίωση. Στη συνέχεια, ορίζουμε σαν register τις εισόδους που θα περάσουμε στο FSM, και σαν wire την έξοδο του.

Για τη δοκιμή και τον έλεγχο μέσω του testbench, δημιουργούμε ένα αρχείο txt, μέσα στο οποίο τοποθετούμε τιμές της μορφής <reset><x_in>_<y_out>. Το txt αυτό θα χρησιμοποιούμε για ευκολότερη επαλήθευση της εξόδου για δεδομένη είσοδο στη συνέχεια. Για τον σκοπό αυτό, ορίζουμε τον TestVector, με 19 «σειρές» (όσες θα γράψουμε στο txt), η κάθε σειρά του οποίου αποτελείται απο 3 bits. Ο register Expected_y_out είναι η αναμενόμενη τιμή για την έξοδο και, τέλος, ο integer i είναι ένας μετρητής για προσπέλαση του TestVector.

Στη συνέχεια, προχωράμε στην κλήση του FSM που υλοποιήσαμε στο προηγούμενο βήμα, με port calls για κάθε είσοδο και έξοδο.

Τα initial blocks που ακολουθούν παίζουν το ρόλο της αρχικοποίησης του ρολογιού (στην τιμή 1), και της αρχικοποίησης του μετρητή (0), του Reset (1) και της εισόδου (0). Το δεύτερο block αποσκοπεί και στην ανάγνωση του txt αρχείου.

Όσον αφορά το ρολόι, θεωρούμε την περιόδο του ως 10 ns, άρα με το always block του ρολογιού κάνουμε κατάλληλη αλλαγή του κάθε 5 ns.

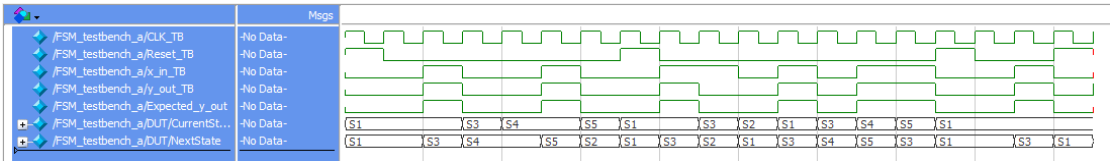
Τέλος, περνάμε στην κύρια λειτουργία του testbench, η οποία επιτελείται με τα δύο τελευταία always blocks. Το πρώτο διαβάζει κάθε τιμή των reset, x_in και αναμενόμενης εξόδου (Expected_y_out) από τον TestVector σε κάθε θετική ακμή του ρολογιού. Στη συνέχεια, σε κάθε αρνητική ακμή του ρολογιού γίνεται μια σύγκριση της αναμενόμενης τιμής εξόδου και της εξόδου που δίνει το υλοποιημένο FSM, ώστε να ολοκληρωθεί η επαλήθευση. Σε περίπτωση σφάλματος, εμφανίζεται στην κονσόλα αντίστοιχο μήνυμα.

Το txt είναι συμπληρωμένο με τέτοιο τρόπο ώστε να ξέρουμε από πριν ποιά είναι η επιθημητή έξοδος κάθε φορά. Πιο αναλύτικα παρουσιάζονται παρακάτω όλες οι μεταβάσεις που χρησιμοποιούνται για τον έλεγχο:

Reset	Input	Expected Output
1	0	0
0	0	0
0	1	1
0	0	0
0	0	0
0	1	1
0	0	0
1	0	0
0	1	1
0	1	0
0	0	0
0	1	1
0	0	0
0	1	1
0	1	1
1	0	0
0	0	0
0	1	1
1	0	0

Προσωμοίωση & Παρατηρήσεις

Μετά από προσομοίωση προκύπτει το εξής διάγραμμα:



Χρησιμοποιούμε το παρακάτω radix για πιο ξεκάθαρη περιγραφή των καταστάσεων:

radix define States {

3'b001 "S1",

3'b011 "S2",

3'b100 "S3",

3'b010 "S4",

3'b000 "S5",

-default hex}

Παρατηρούμε ότι όντως προκύπτει η αναμενόμενη έξοδος κάθε φορά, το οποίο επαληθεύει την υλοποίηση του FSM μας. Η διάρκεια της προσομοίωσης είναι 190 ns, καθώς η περίοδος του ρολογιού είναι 10 ns και έχουμε ορίζει στο αρχείο μας 19 μεταβάσεις.

Σαν υποσημείωση θέλω να επισημάνω πως το Reset δεν φαίνεται να λειτουργεί αν η είσοδος και η αναμενόμενη έξοδος δεν πάρουν την τιμή 0 (δεν τίθεται προτεραιότητα στο Reset).

β) Στο ερώτημα β καλούμαστε να περιγράψουμε το FSM χρησιμοποιώντας D Flip Flop, αποθηκεύοντας δηλαδή τις καταστάσεις του σε αυτα.

Στο παραπάνω συνεισφέρουν οι εξισώσεις που βρήκαμε στο πρώτο ερώτημα όσον αφορά τα D2', D1', D0' και x_in. Χρησιμοποιώντας τις εξισώσεις αυτές, προχωράμε σε structural περιγραφή του FSM (χρησιμοποιώτας λογικές πύλες). Το εσωτερικό του D Flip Flop, από την άλλη, περιγράφεται με behavioural Verilog.

<u>Υλοποίηση module & Περιγραφή</u>

Ο κώδικας για το εσωτερικό του D Flip Flop παρουσιάζεται παρακάτω:

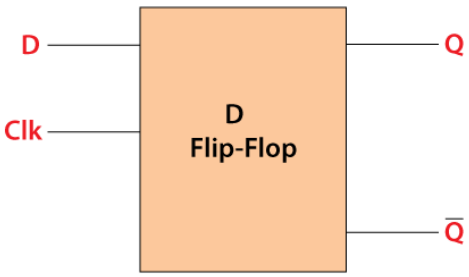
*

```
module dflipflop (output reg Q, Qn,
                  input wire CLK, Reset, D, R);
 always @ (posedge CLK or posedge Reset)
  if (Reset)
   if (R == 1)
    begin
        Q <= 1'b1;
        Qn <= 1'b0;
     end
```

```
    else
     begin
          Q <= 1'b0;
          Qn <= 1'b1;
     end
   else
    begin
          Q <= D;
          Qn <= ~D;
    end
endmodule
*
```

Την έξοδο του φλιπ φλοπ αποτελούν τα Q και Qn (συμπλήρωμα του Q), ορισμένα ως reg. Την είσοδο αποτελούν η κλασσική είσοδος του D φλιπ φλοπ, το reset, το clock, καθώς και μία μεταβλητής R, ο ρόλος της οποίας περιγράφεται στη συνέχεια (όλα ορισμένα ως wire).

Η λειτουργία του περιγράφεται με ένα always block, το οποίο «ενεργοποιείται» με τη θετική ακμή του ρολογιού ή με active high reset, το περιεχόμενο του οποίου προκύπτει από τον κλασσικό πίνακα αληθείας ενός D Flip Flop:



Clock	D	Q_{next}
Rising edge	0	0
Rising edge	1	1
Non-rising	X	Q

Όταν λοιπόν έχουμε θετική ακμή του ρολογιού, στο Q περνάει η είσοδος D, και στο Qn προφανώς το συμπλήρωμα του.

Από την άλλη, αν δοθεί active high reset προχωράμε σε μία είδους «αρχικοποιήση» του φλιπ, κατά την οποία το Q παίρνει την τιμή 0 και το Qn την τιμή 1.

Επειδή όμως η κατάσταση του FSM μας στην οποία επιστρέφει όταν δοθεί reset είναι η κατάσταση 001, το 3ο κατά σειρά φλιπ φλοπ πρέπει να έχει σαν έξοδο την τιμή 1 και οχι την τιμή 0. Συνεπώς, συμπεριλαμβάνουμε στον κωδικά μας τη μεταβλητή R, η τιμή της οποίας δείχνει αν πρέπει το φλιπ φλοπ να αρχικοποιηθεί με τιμή 0 (1ο και 2ο φλιπ φλοπ) ή με τιμη 1.

Ο κώδικας για την περιγραφή του FSM με τη χρήση D Flip Flop και structural Verilog παρουσιάζεται παρακάτω:

*

```
module FSM_b(output wire y_out,
        input wire x_in, CLK, Reset);
wire D2, D1, D0;
wire Q2, Q1, Q0, Qn2, Qn1, Qn0;

dflipflop DUT2 (.Q(Q2), .Qn(Qn2), .CLK(CLK), .Reset(Reset), .D(D2), .R(0));
dflipflop DUT1 (.Q(Q1), .Qn(Qn1), .CLK(CLK), .Reset(Reset), .D(D1), .R(0));
dflipflop DUT0 (.Q(Q0), .Qn(Qn0), .CLK(CLK), .Reset(Reset), .D(D0), .R(1));

assign D2 = Qn2 & Qn1 & x_in;
assign D1 = (Qn2 & Qn0 & ~x_in) | (Q2 & Qn1 & Qn0) | (Qn2 & Q1 & Q0 & x_in) ;
assign D0 = (Qn2 & Qn1 & ~x_in) | (Qn2 & Q0 & ~x_in) | (Q2 & Qn1 & Qn0 & x_in);
assign y_out = x_in & Qn2;

endmodule
```

*

Είσοδο του FSM αποτελούν, κλασσικά, η είσοδος x_in, το ρολόι και το reset, όλα σε μορφή wire. Την έξοδο του από την άλλη αποτελεί η έξοδος y_out ως reg.

Αρχικά, ορίζουμε τις εισόδους των φλιπ φλοπ D2, D1, D0, καθώς και τις εξόδους τους Q2, Q1, Q1, Qn2, Qn1 και Qn0, ως wire. Αμέσως μετά καλούνται τα 3 φλιπ φλοπ που θα χρησιμποιηθούν, το 3ο των οποίων καλείται με τιμή R=1 ώστε να αρχικοποιηθεί σωστά σε περίπτωση reset.

Τέλος, χρησιμοποιώντας assign και λογικές πύλες, γράφουμε αναλυτικά τις εξισώσεις (οι οποίες παρατίθενται ξανά παρακάτω), και δίνουμε τιμές στις εισόδους των φλιπ φλοπ D, τιμές οι οποίες με κάθε επόμενο χτύπο ρολογιού περνούν στις εξόδους Q και οδηγούν στη σωστή λειτουργία του FSM. Οι 3 είσοδοι D λοιπόν (και κατα συνέπεια οι έξοδοι Q) δείχνουν τα 3 bits της κάθε κατάστασης του FSM και αλλάζουν αναλόγως.

$$D2' = \overline{D2D1X}$$

$$D1' = \overline{D2D0X} + D2\overline{D1D0} + \overline{D2}D1D0X$$

$$D0' = \overline{D2D1X} + \overline{D2}D0\overline{X} + D2\overline{D1D0X}$$

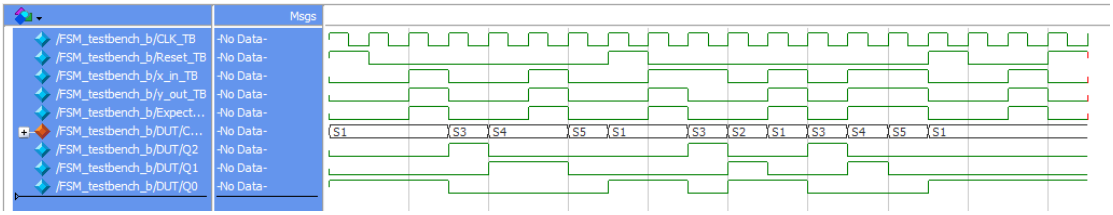
$$y\_out = \overline{D2}X$$

Υλοποίηση testbench & Περιγραφή

Το testbench που χρησιμοποιείται για την επαλήθευση του FSM είναι ακριβώς το ίδιο με το προηγούμενο ερώτημα, με μόνη διαφορά προφανώς την κλήση πλεον του structural FSM που περιγράψαμε παραπάνω.

Προσωμοίωση & Παρατηρήσεις

Μετά από προσομοίωση προκύπτει το εξής διάγραμμα:



Μπορεί να φανεί καθαρά η έξοδος Q των φλιπ φλοπ που έχουμε ορίσει με structural Verilog. Οι τρεις έξοδοι Q μαζί μας δείχνουν το current state του FSM.

Χρησιμοποιούμε (όπως και στο πρηγούμενο ερώτημα) το παρακάτω radix για πιο ξεκάθαρη περιγραφή των καταστάσεων:

radix define States {

3'b001 "S1",

3'b011 "S2",

3'b100 "S3",

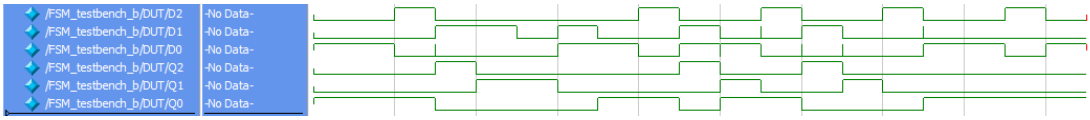
3'b010 "S4",

3'b000 "S5",

-default hex}

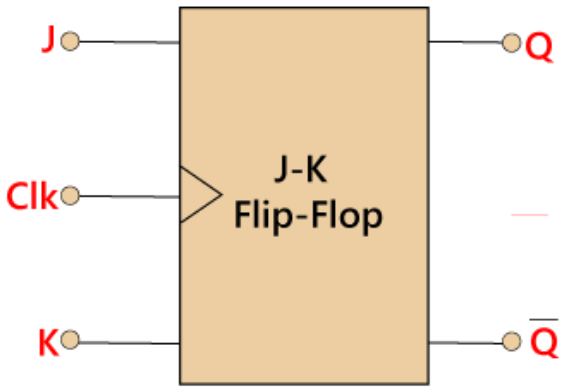
Παρατηρούμε ότι όντως προκύπτει η αναμενόμενη έξοδος κάθε φορά, επαληθεύουμε λοιπόν την υλοποίηση του FSM. Η διάρκεια της προσομοίωσης είναι 190 ns, καθώς η περίοδος του ρολογιού είναι 10 ns και έχουμε ορίζει στο αρχείο μας 19 μεταβάσεις.

Ως υποσημείωση, επισημαίνουμε πως υπάρχουν glitches σε κάποια σημεία των D, τα οποία εκτιμώ πως δημιουργούνται λόγω της αλλαγής της εισόδου πριν προλάβει να γίνει η μετάβαση σε επόμενη κατάσταση.



γ) Στο ερώτημα γ καλούμαστε να περιγράψουμε το FSM χρησιμοποιώντας JK Flip Flop, αποθηκεύοντας δηλαδή τις καταστάσεις του σε αυτα, ερώτημα παρόμοιο με το προηγούμενο, με μόνη αλλαγή το είδος του φλιπ φλοπ.

Για την υλοποίηση των παραπάνω χρειάστηκε δημιουργία ενός νέου πίνακα αληθείας για το FSM, ο οποίο προκύπτει από τον κλασσικό πίνακα αληθείας ενός JK Flip Flop:



Q	Q_{next}	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

Μέσω του παρακάτω πίνακα αληθείας προκύπτουν και οι αντίστοιχες σχέσεις, μέσω των οποίων προχωράμε σε structural περιγραφή του FSM. Τέλος, το εσωτερικό του JK Flip Flop περιγράφεται με behavioural Verilog.

Πίνακας αληθείας για τον προσδιορισμό των J, K και σχέσεις:

D2	D1	D0	x_in	y_out	D2'	D1'	DO'	J2	K2	J1	K1	J0	K0
0	0	1	0	0	0	0	1	0	X	0	X	X	0
0	0	1	1	1	1	0	0	1	X	0	X	X	1
1	0	0	0	0	0	1	0	X	1	1	X	0	X
1	0	0	1	0	0	1	1	X	1	1	X	1	X
0	1	1	0	0	0	0	1	0	X	X	1	X	0
0	1	1	1	1	0	1	0	0	X	X	0	X	1
0	1	0	0	0	0	1	0	0	X	X	0	0	X
0	1	0	1	1	0	0	0	0	X	X	1	0	X
0	0	0	0	0	0	1	1	0	X	1	X	1	X
0	0	0	1	1	1	0	0	1	X	0	X	0	X

$J2 = \overline{D2D1}X$ $J1 = \overline{D1D0X} + D2\overline{D1D0}$ $J0 = \overline{D2D1X} + D2\overline{D1D0X}$

$K2 = 1$ $K1 = \overline{D2D0X}\ \overline{D2}D0\overline{X}$ $K0 = \overline{D2}X$

<u>Υλοποίηση module & Περιγραφή</u>

Ο κώδικας για το εσωτερικό του JK Flip Flop παρουσιάζεται παρακάτω:

*

```
module jkflipflop (output reg Q, Qn,
                   input wire J, K, CLK, Reset, R);
```

```
always @ (posedge CLK or posedge Reset)
if (Reset)
 if (R == 1)
  begin
        Q <= 1'b1;
        Qn <= 1'b0;
   end
 else
  begin
        Q <= 1'b0;
        Qn <= 1'b1;
  end
else
 begin
  case ({J,K})
    2'b00 :  begin
                  Q <= Q;
                  Qn <= ~Q;
                 end
    2'b01 :  begin
                  Q <= 1'b0;
                  Qn <= 1'b1;
                 end
    2'b10 :  begin
                  Q <= 1'b1;
```

```
                    Qn <= 1'b0;
                   end
        2'b11 :  begin
                    Q <= ~Q;
                    Qn <= Q;
                   end
     endcase
   end
endmodule
*
```

Την έξοδο του φλιπ φλοπ αποτελούν τα Q και Qn (συμπλήρωμα του Q), ορισμένα ως reg. Την είσοδο αποτελούν η κλασσικές είσοδοι του JK φλιπ φλοπ (J, K), το reset, το clock, καθώς και μία μεταβλητής R, ο ρόλος της οποίας είναι ακριβώς ο ίδιος με το πρηγούμενο ερώτημα (αρχικοποίηση του 3ου φλιπ φλοπ έξοδο 1 και όχι 0).

Η λειτουργία του περιγράφεται με ένα always block, το οποίο «ενεργοποιείται» με τη θετική ακμή του ρολογιού ή με active high reset, το περιεχόμενο του οποίου προκύπτει από τον πίνακα αληθείας του JK φλιπ φλοπ:

JK latch truth table			
J	**K**	**Q_{next}**	**Comment**
0	0	Q	No change
0	1	0	Reset
1	0	1	Set
1	1	$\overline{Q}$	Toggle

Αν δοθεί active high reset τότε το φλιπ φλοπ προχωράει σε αρχικοποίηση, δηλαδή στην έξοδο Q δίνεται 0, εκτός από το τελευταίο φλιπ φλοπ στο οποίο δίνεται 1.

Η κανονική λειτουργία του φλιπ φλοπ, από την άλλη, υλοποιείται με ένα case block, ορίσματα του οποίου είναι τα (J, K), και ανάλογα με αυτά δίνεται η αντίστοιχη έξοδος με βάση τον παραπάνω πίνακα αληθείας.

Ο κώδικας για την περιγραφή του FSM με τη χρήση JK Flip Flop και structural Verilog παρουσιάζεται παρακάτω:

```
*

module FSM_c(output wire y_out,

        input wire x_in, CLK, Reset);


wire J2, J1, J0, K2, K1, K0;

wire Q2, Q1, Q0, Qn2, Qn1, Qn0;


jkflipflop DUT2 (.Q(Q2), .Qn(Qn2), .J(J2), . K(K2), .CLK(CLK), .Reset(Reset), .R(0));

jkflipflop DUT1 (.Q(Q1), .Qn(Qn1), .J(J1), . K(K1), .CLK(CLK), .Reset(Reset), .R(0));

jkflipflop DUT0 (.Q(Q0), .Qn(Qn0), .J(J0), . K(K0), .CLK(CLK), .Reset(Reset), .R(1));


assign J2 = Qn2 & Qn1 & x_in;

assign K2 = 1'b1;

assign J1 = (Qn1 & Qn0 & ~x_in) | (Q2 & Qn1 & Qn0);

assign K1 = (Qn2 & Qn0 & x_in) | (Qn2 & Q0 & ~x_in);

assign J0 = (Qn2 & Qn1 & ~x_in) | (Q2 & Qn1 & Qn0 & x_in);

assign K0 = Qn2 & x_in;


assign y_out = x_in & Qn2;


endmodule

*
```

Είσοδο του FSM αποτελούν, κλασσικά, η είσοδος x_in, το ρολόι και το reset, όλα σε μορφή wire. Την έξοδο αποτελεί η έξοδος y_out ως reg.

Αρχικά, ορίζουμε τις εισόδους των φλιπ φλοπ J2, J1, J0, K2, K1, K0, καθώς και τις εξόδους τους Q2, Q1, Q1, Qn2, Qn1 και Qn0, ως wire. Στη συνέχεια,

καλούνται τα 3 φλιπ φλοπ που θα χρησιμποιηθούν, το 3ο εκ των οποίων καλείται με τιμή R=1 ώστε να αρχικοποιηθεί σωστά σε περίπτωση reset.

Τέλος, χρησιμοποιώντας assign και λογικές πύλες, γράφουμε αναλυτικά τις εξισώσεις (οι οποίες παρατίθενται ξανά παρακάτω), και δίνουμε τιμές στις εισόδους των φλιπ φλοπ (J, K), τιμές οι οποίες οδηγούν στη σωστή λειτουργία του FSM μέσω του εσωτερικού case block στον προαναφερθέντα κώδικα. Οι 3 έξοδοι Q λοιπόν δείχνουν τα 3 bits της κάθε κατάστασης του FSM και αλλάζουν αναλόγως.

$J2 = \overline{D2D1X}$ $J1 = \overline{D1D0X} + D2\overline{D1D0}$ $J0 = \overline{D2D1X} + D2\overline{D1D0X}$

$K2 = 1$ $K1 = \overline{D2D0X}\ \overline{D2}D0\overline{X}$ $K0 = \overline{D2}X$

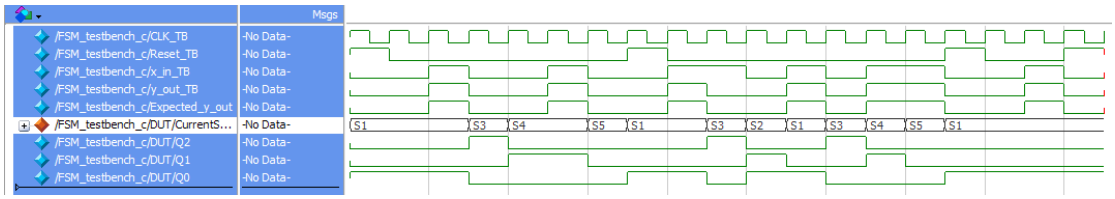
$y\_out = \overline{D2}X$

Υλοποίηση testbench & Περιγραφή

Το testbench που χρησιμοποιείται για την επαλήθευση του FSM είναι ακριβώς το ίδιο με τα προηγούμενα ερωτήματα, και φυσικά εδώ καλείται το FSM που δημιουργήθηκε με JK φλιπ φλοπ.

Προσωμοίωση & Παρατηρήσεις

Μετά από προσομοίωση προκύπτει το εξής διάγραμμα:



Φαίνεται χαρακτηριστικά η έξοδος Q των φλιπ φλοπ που έχουμε ορίσει με structural Verilog. Οι τρεις έξοδοι Q μαζί μας δείχνουν το current state του FSM.

Χρησιμοποιούμε παρόμοια με τα προηγούμενα ερωτήματα το παρακάτω radix για πιο ξεκάθαρη περιγραφή των καταστάσεων:

radix define States {

3'b001 "S1",

3'b011 "S2",

3'b100 "S3",

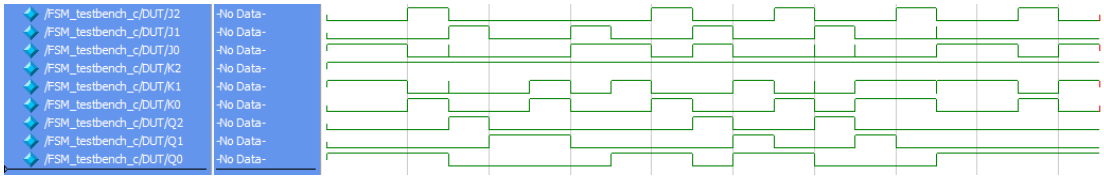
3'b010 "S4",

3'b000 "S5",

-default hex}

Παρατηρούμε ότι όντως προκύπτει η αναμενόμενη έξοδος κάθε φορά και έτσι επαληθεύουμε την υλοποίηση του FSM. Η διάρκεια της προσομοίωσης είναι 190 ns, καθώς η περίοδος του ρολογιού είναι 10 ns και έχουμε ορίζει στο αρχείο μας 19 μεταβάσεις.

Ως υποσημείωση, επισημαίνουμε πως υπάρχουν glitches σε κάποια σημεία των (J, K), τα οποία εκτιμώ πως δημιουργούνται λόγω της αλλαγής της εισόδου πριν προλάβει να γίνει η μετάβαση σε επόμενη κατάσταση.



ΑΣΚΗΣΗ 2

Εισαγωγικά Συμπεράσματα & Ανάλυση

Στην άσκηση 2, καλούμαστε να σχεδιάσουμε ένα απαριθμητή BCD τεσσάρων ψηφίων (ο οποίος μετράει από το 0000 μέχρι το 9999) με χρήση τη γλώσσας Verilog και αρχές ιεραρχική σχεδίασης. Καλούμαστε επίσης να υλοποιήσουμε κάθε module (εκτός του testbench) με structural Verilog.

Για την υλοποίηση των παραπάνω δημιουργήσαμε 4 modules και ένα testbench. Επιγραμματικά τα modules αποτελούν τα εξής:

- T Flip Flop, για την αποθήκευση κάθε bit του απαριθμητή
- BCD, ο κλασσικός μετρητής 4 bit που μετράει από το 0 έως το 9
- Decoder, ο οποίος καλεί τον BCD και δίνει τιμές στα LED που αποτελούν την δεκαδική έξοδο του συστήματος
- Decoder_4, module που ενώνει του 4 απαριθμητές και τους χρονίζει κατάλληλα ώστε να γίνεται σωστά η μέτρηση
- Counter_TB, testbench το οποίο ελέγχει τη σωστή λειτουργία του συστήματος

Υλοποίηση module & Περιγραφή

Αρχικά, ώστε να ξεκινήσουμε την υλοποίηση, δημιουργήσαμε το module ενός T Flip Flop. Εδώ πρέπει να σημειώσω ότι υπήρξε πρόβλημα στην υλοποίηση του με structural Verilog, οπότε προχώρησα στην συγγραφή του με behavioural Verilog ώστε να μπορέσω να συνεχίσω την άσκηση. Σημειώνω με σχόλια της προσπάθεια υλοποίησης με structural.

Ο κώδικας παρουσιάζεται παρακάτω:

*

```
module Tflipflop (output reg Q, Qn,
                  input wire CLK, T, Reset);


always @ (posedge CLK or posedge Reset)
begin
if (Reset)
 begin
  Q <= 1'b0;
  Qn <= 1'b1;
 end

if (T == 0 && Reset == 0)
 begin
  Q <= Q;
  Qn <= Qn;
 end

if (T == 1 && Reset ==0)
 begin
```

```
  Q <= Qn;

  Qn <= Q;

 end

end


//structural

//assign Q = ~((Q & T & CLK) | Qn);

//assign Qn = ~((T & CLK & Qn) | Q);


endmodule
```

*

Είσοδο του φλιπ φλοπ αποτελούν τα clock, reset και φυσικά T (toggle) σε μορφή wire. Έξοδο του αποτελούν οι έξοδοι Q και Qn σε μορφή reg.

Η λειτουργία του έχει ως εξής: Στην θετική ακμή του ρολογιού ή σε active high reset, «ενεργοποιείται» το always block που το αποτελεί.

Σε περίπτωση reset, η έξοδος του μηδενίζεται, κ φυσικά το συμπλήρωμα της παίρνει την τιμή 1.

Σε περίπτωση που η είσοδος (T) είναι 0, δεν έχουμε toggle και έτσι οι τιμές των Q και Qn παραμένουν ίδιες. Από την άλλη, όταν T=1, υπάρχει toggle, δηλαδή τα Q και Qn παίρνουν αντίθετες από την προηγούμενη κατάσταση τους τιμές (0 → 1, 1 → 0).

Στη συνέχεια, υλοποιήσαμε το module του μετρητή, το σημαντικότερο ίσως module στο σύστημα μας. Το module του BCD καλεί τα 4 φλιπ φλοπ που χρησιμοποιούνται ώστε να γίνεται σωστά η μέτρηση από το 0 μέχρι το 9. Η μέτρηση αυτή υλοποιείται και αναλύεται σαν ένα FSM, και με αυτό τον τρόπο περιγράφονται και οι σχέσεις για τις εισόδους T των φλιπ φλοπ.

Πιο συγκεκριμένα, παρακάτω παρουσιάζεται ο πίνακας με τα current states, next states και είσοδους T για τον μετρητή, καθώς και οι σχέσεις που προκύπτουν:

Current State				Next State				Toggle			
C3	C2	C1	C0	C3’	C2’	C1’	CO’	T3	T2	T1	T0
0	0	0	0	0	0	0	1	0	0	0	1
0	0	0	1	0	0	1	0	0	0	1	1
0	0	1	0	0	0	1	1	0	0	0	1
0	0	1	1	0	1	0	0	0	1	1	1
0	1	0	0	0	1	0	1	0	0	0	1
0	1	0	1	0	1	1	0	0	0	1	1
0	1	1	0	0	1	1	1	0	0	0	1
0	1	1	1	1	0	0	0	1	1	1	1
1	0	0	0	1	0	0	1	0	0	0	1
1	0	0	1	0	0	0	0	1	0	0	1

$T3 = \overline{C3}C2C1C0 + C3\,\overline{C2C1}C0$

$T2 = \overline{C3}C1C0$

$T1 = \overline{C3}C0$

$T0 = \overline{C3} + \overline{C2C1}$ (εδώ $T0 = 1$)

Ο παραπάνω πίνακας συμπληρώθηκε με τη βοήθεια του κλασσικού πίνακα αληθείας ενός T φλιπ φλοπ:

T	**Q**	**Qn**	**Qnext**	**Qnnext**
0	0	1	0	1
0	1	0	1	0
1	0	1	1	0
1	1	0	0	1

Ακολουθεί ο κώδικας που περιγράφει τον μετρητή:

*

```
module BCD(input wire CLK, Reset, En,
        output wire Q1, Q2, Q3, Q0);
```

```
wire Qn3, Qn2, Qn1, Qn0;

wire T3, T2, T1, T0;


Tflipflop DUT3 (.Q(Q3), .Qn(Qn3), .T(T3), .CLK(CLK), .Reset(Reset));

Tflipflop DUT2 (.Q(Q2), .Qn(Qn2), .T(T2), .CLK(CLK), .Reset(Reset));

Tflipflop DUT1 (.Q(Q1), .Qn(Qn1), .T(T1), .CLK(CLK), .Reset(Reset));

Tflipflop DUT0 (.Q(Q0), .Qn(Qn0), .T(T0), .CLK(CLK), .Reset(Reset));


assign T1 = Qn3 & Q0;

assign T2 = Qn3 & Q1 & Q0;

assign T0 = En;

assign T3 = (Qn3 & Q2 & Q1 & Q0) | (Q3 & Qn2 & Qn1 & Q0);


endmodule
```

*

Είσοδος του συγκεκριμένου αποτελεί (όπως πάντα) το clock, το reset και το enable που ελέγχει τη λειτουργία ή μη του απαριθμητή, μορφής wire. Εξόδους αποτελούν όλοι οι έξοδοι Q των φλιπ φλοπ, οι οποίες μας δείχνουν το state του μετρητή αλλά θα χρησιμοποιηθούν και στη συνέχεια.

Αρχικά, καλούνται τα 4 T Flip Flop που χρησιμοποιούνται για τη μέτρηση με χρήση 4 bit. Στη συνέχεια δίνονται τιμές στις εισόδους T των φλιπ φλοπ μέσω των προαναφερθείσων σχέσεων που προέκυψαν από τον πίνακα αληθείας. Έτσι, με κάθε χτύπο ρολογιού (λόγω του εσωτερικού των φλιπ φλοπ), ο μετρητής περνάει στον επόμενο αριθμό.

Αξίζει να σημειωθεί ότι το T0, το οποίο μέσω των σχέσεων βρήκαμε ως 1, παίρνει σαν είσοδο το enable, το οποίο είναι 1 όταν η μετρητής λειτουργεί και μηδενίζεται αν σταματήσει η λειτουργία του. Συνεπώς, αν δοθεί enable $= 0$ από σημείο ανώτερης ιεραρχίας, ο απαριθμητής παύει να λειτουργεί, καθώς ένα από τα φλιπ φλοπ του παίρνει λανθασμεένη τιμή. Παρόμοια

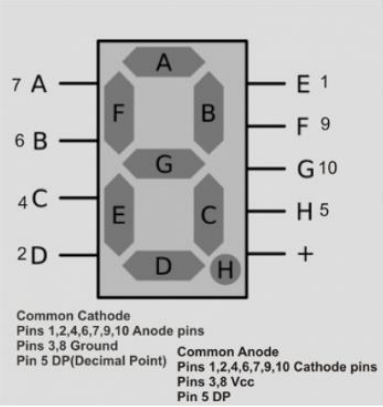
(κάτι που ελέγχεται από το εσωτερικό των φλιπ φλοπ) αν δοθει reset = 1, ο απαριθμητής παύει να λειτουργεί κ ξεκινάει ξανά από την αρχή.

Στη συνέχεια, το επόμενο κομμάτι της ιεραρχίας αποτελεί ο αποκωδικοποιητής, το module δηλαδή το οποίο εμφανίζει τις τιμές του LED που δείχνει στο δεκαδικό σύστημα τον αριθμό στον οποίο βρισκόμαστε.

Χρησιμοποιούμε ξανά ένα πίνακα αληθείας, οι σχέσεις του οποίου προκύπτουν παρόμοια με τα παραπάνω. Ο πίνακας αληθείας αυτός είναι ο εξής:

C3	C2	C1	C0	A	B	C	D	E	F	G
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

Τα A, B, C, D, E, F, G αποτελούν τις εξόδους του LED σύμφωνα με την δοσμένη εικόνα:



Το H μπορεί να λειτουργήσει σαν υποδιαστολή στις χιλιάδες (δηλαδή να είναι πάντα αναμμένη στον πρώτο LED μετρητή), κάτι το οποίο υλοποιείται στη συνέχεια μέσα στον κώδικα.

Οι σχέσεις που προκύπτουν από τον πίνακα αληθείας είναι οι εξής:

$$A = \overline{C3}C1 + \overline{C3C2C0} + \overline{C3}C2C0 + C3\overline{C2C1}$$

$$B = \overline{C3C2} + \overline{C2C1} + \overline{C3C1C0} + \overline{C3}C1C0$$

$$C = \overline{C2C1} + \overline{C3}C0 + \overline{C3}C2$$

$$D = \overline{C3C2C0} + \overline{C3C2}C1 + \overline{C3}C1\overline{C0}\ C3\overline{C2C1} + \overline{C3}C2\overline{C1}C0$$

$$E = \overline{C2C1C0} + \overline{C3}C1\overline{C0}$$

$$F = \overline{C3C1C0} + \overline{C3}C2\overline{C1} + C3\overline{C2C1}$$

$$G = \overline{C3C2}C1 + \overline{C3}C1\overline{C0} + \overline{C3}C2\overline{C1} + C3\overline{C2C1}$$

Ο κώδικας για τον αποκωδικοποιητή παρουσιάζεται παρακάτω:

*

```
module Decoder (input wire CLK, Reset, Type, En, H,
                output wire [0:7] LED, wire Q0, Q1, Q2, Q3);


wire Qn0, Qn1, Qn2, Qn3;


assign Qn0 = ~Q0;

assign Qn1 = ~Q1;

assign Qn2 = ~Q2;

assign Qn3 = ~Q3;


BCD DUT(.CLK(CLK), .Reset(Reset), .Q1(Q1), .Q2(Q2), .Q3(Q3), .Q0(Q0), .En(En));


assign A = ((Qn3 & Q1) | (Qn3 & Qn2 & Qn0) | (Qn3 & Q2 & Q0) | (Q3 & Qn2 & Qn1))
^ Type;

assign B = ((Qn3 & Qn2) | (Qn2 & Qn1) | (Qn3 & Qn1 & Qn0) | (Qn3 & Q1 & Q0)) ^
Type;

assign C = ((Qn2 & Qn1) | (Qn3 & Q0) | (Qn3 & Q2)) ^ Type;

assign D = ((Qn3 & Qn2 & Qn0)| (Qn3 & Qn2 & Q1) | (Qn3 & Q1 & Qn0) | (Q3 & Qn2
& Qn1) | (Qn3 & Q2 & Qn1 & Q0)) ^ Type;

assign E = ((Qn2 & Qn1 & Qn0) | (Qn3 & Q1 & Qn0)) ^ Type;
```

```
assign F = ((Qn3 & Qn1 & Qn0) | (Qn3 & Q2 & Qn1)| (Qn3 & Q2 & Qn0) | (Q3 & Qn2 & Qn1)) ^ Type;

assign G = ((Qn3 & Qn2 & Q1) | (Qn3 & Q1 & Qn0) | (Qn3 & Q2 & Qn1) | (Q3 & Qn2 & Qn1)) ^ Type;


assign LED[0] = A;

assign LED[1] = B;

assign LED[2] = C;

assign LED[3] = D;

assign LED[4] = E;

assign LED[5] = F;

assign LED[6] = G;

assign LED[7] = H;


endmodule
```

*

Είσοδο του module αποτελούν τα clock, reset, type (common anode ή common cathode), enable, καθώς και το H, όλα σε μορφή wire. Την έξοδο αποτελούν φυσικά ένα vector LED[0:7], καθώς και όλες οι έξοδοι Q των τρανζίστορ που θα μας χρησιμεύσουν στη συνέχεια.

Αρχικά, καλείται ο απαριθμητής BCD με τις απαραίτητες εισόδους, ώστε να εκκινήσει το σύστημα. Στη συνέχεια, με assign δίνονται τιμές στα LED A μέχρι G, χρησιμοποιώντας τις σχέσεις που βρήκαμε παραπάνω. Οι τιμές αυτές, τέλος, περνούν σε ένα vector ώστε να φαίνονται πιο ξεκάθαρα όταν προχωρήσουμε στην προσομοίωση. Εδώ σημειώνεται πως η τιμή του H, η οποία αποτελεί είσοδο του μετρητή (άρα τη δίνουμε εμείς επιλεκτικά σε ανώτερο επίπεδο ιεραρχίας) περνάει επίσης στο vector των LED σαν τελευταία τιμή.

Προφανώς, όταν ένα LED είναι αναμμένο παίρνει την τιμή 1, και όταν είναι σβηστό την τιμή 0. Πρόσθετα σχολιάζεται εδώ ότι η μεταβλητή type ενώνεται με μία πύλη XOR με κάθε LED, ώστε αν είναι 0 το LED να

παρεμένει ίδιο (κοινής ανόδου) και αν είναι 1 να αντιστρέφεται (κοινής καθόδου).

Το τελευταίο module πριν την προσομοίωση αποτελεί ο παρακάτω κώδικας, ο οποίο περιγράφει τον ολοκληρωμένο απαραιθμητή, ενώνει δηλαδή τους 4 εσωτερικούς απαριθμητές για να γίνει τελικώς μέτρηση τεσσάρων ψηφίων:

*

```
module Decoder_4 (input wire CLK, Reset, Type, En,
                  output wire [0:7] LED1, [0:7] LED2, [0:7] LED3, [0:7] LED4);


Decoder DUT0 (.CLK(CLK), .Reset(Reset), .Type(Type), .En(En), .H(1'b0), .LED(LED4),
.Q0(Q0_0), .Q1(Q1_0), .Q2(Q2_0), .Q3(Q3_0));


Decoder DUT1 (.CLK(~(Q0_0 | Q1_0 | Q2_0 | Q3_0)), .Reset(Reset), .Type(Type),
.En(En), .H(1'b0), .LED(LED3), .Q0(Q0_1), .Q1(Q1_1), .Q2(Q2_1), .Q3(Q3_1));


Decoder DUT2 (.CLK(~(Q0_1 | Q1_1 | Q2_1 | Q3_1)), .Reset(Reset), .Type(Type),
.En(En), .H(1'b0), .LED(LED2), .Q0(Q0_2), .Q1(Q1_2), .Q2(Q2_2), .Q3(Q3_2));


Decoder DUT3 (.CLK(~(Q0_2 | Q1_2 | Q2_2 | Q3_2)), .Reset(Reset), .Type(Type),
.En(En), .H(1'b1), .LED(LED1), .Q0(Q0_3), .Q1(Q1_3), .Q2(Q2_3), .Q3(Q3_3));


endmodule
```

*

Είσοδο του παραπάνω κώδικα αποτελούν τα clock, reset, type και enable, τα οποία έχουν όλα εξηγηθεί παραπάνω. Την έξοδο αποτελούν 4 vectors, τα οποία δείχνουν ουσιαστικά τον αριθμό στον οποίο βρίσκεται ο κάθε μετρητής, ολοκληρώνοντας έτσι τη μέτρηση των 4 ψηφίων.

Στο εσωτερικό του module, καλούνται οι 4 απαριθμητές. Το λεπτό σημείο εδώ είναι πως σαν clock του κάθε απαριθμητή (εκτός του πρώτου στον

οποίο δίνεται το αρχικό clock) δίνεται το συμπλήρωμα του bit που προκύπτει αν οι 4 έξοδοι Q του προηγούμενου απαριθμητή ενωθούν μεταξύ τους με πύλες OR. Ο σκοπός της σχεδίασης αυτής είναι ο εξής: Θέλουμε ο επόμενος απαριθμητής πάντα να ξεκινάει/αλλάζει τιμή κάθε φορα που ο προηγούμενος απαριθμητής κάνει ένα κύκλο, μεταβαίνει λοιπόν από το 9 (1001) στο 0 (0000). Μιας και η μόνη περίπτωση που η παραπάνω σχέση μπορεί να είναι 1 είναι όταν η κατάσταση είναι 0000, ορίσαμε το ρολόι με αυτό τον τρόπο.

Υλοποίηση testbench & Περιγραφή

Ο κώδικας που αποτελεί το testbench είναι ο εξής:

*

```
`timescale 1ns/1ns


module Counter_TB;


  reg CLK_TB, Type;

  reg Reset_TB, En;

  wire [0:7] LED1;

  wire [0:7] LED2;

  wire [0:7] LED3;

  wire [0:7] LED4;


  Decoder_4 DUT(.CLK(CLK_TB), .Reset(Reset_TB), .Type(Type), .En(En),
.LED1(LED1[0:7]), .LED2(LED2[0:7]), .LED3(LED3[0:7]), .LED4(LED4[0:7]));


initial

  begin

    CLK_TB = 1'b1;

  end
```

```
initial
  begin
    Reset_TB = 1;
  end

initial
  begin
    Type = 1'b0;
  end

initial
  begin
    En = 1'b0;
  end

always
  begin
   #1200 En = 1'b1;
 end

always
  begin
   #5000 Reset_TB = 0;
 end
```

```
always
  begin
    #500 CLK_TB = ~CLK_TB;
  end

endmodule
```

*

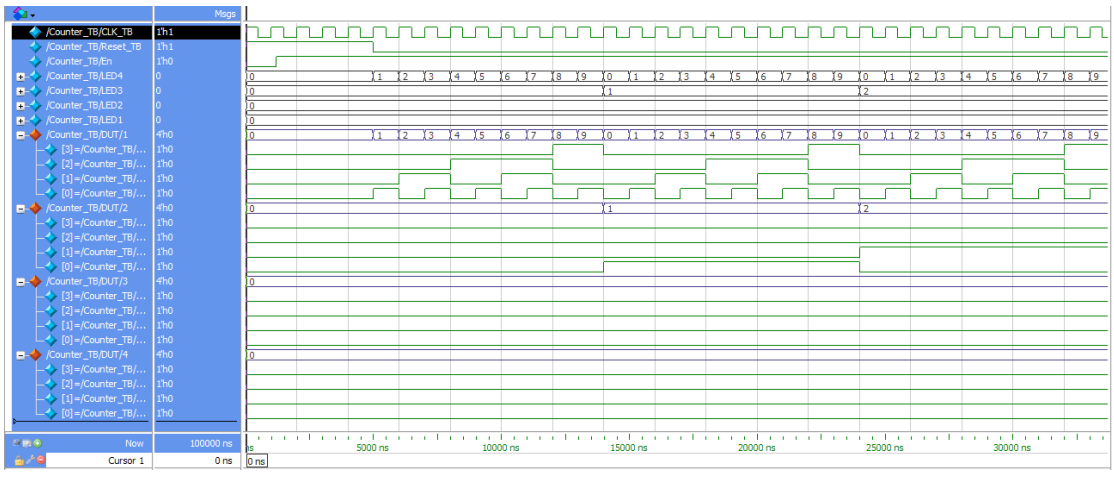
Αρχικά, ορίζονται τα clock, type, reset, enable, καθώς και 4 vectors για τα LED, ώστε να μπορεί να τρέξει σωστά το σύστημα μας. Στη συνέχεια, καλείται ο απαριθμητής τεσσάρων bit, ο οποίο ιεραρχικά καλεί όλα τα υπόλοιπα modules.

Τα 4 initial blocks που έχουμε ορίσει δίνουν αρχικές τιμές στα: clock (0), reset (1), type (0) και enable (0). Θέλουμε στο σύστημα μας συχνότητα 1 MHz, άρα περίοδο 1μs, γι'αυτο και ορίζουμε την περίοδο του ρολογιού έτσι στο τελευταίο always block (διότι έχουμε timescale 1ns/1ns).

Τέλος, όσον αφορά τα always blocks για τα reset και enable, δίνουμε τυπικά την τιμή 1 για την έναρξη του απαριθμητή σε 2 χρονικές στιγμές, κι έτσι έχουμε enable = 1 στα 1200 ns και reset =0 στα 5000 ns, όπου αναμένουμε η μέτρηση να ξεκινήσει να μετρά κανονικά. Όποτε ξαναδοθεί reset, ο μετρητής θα γυρίσει στο 0000 και θα ξαναξεκινήσει να μετρά κανονικά.

Προσωμοίωση & Παρατηρήσεις

Μετά από προσομοίωση προκύπτει το εξής διάγραμμα:



Δεν είναι δυνατόν να φανεί όλη η λειτουργία του απαριθμητή, οπότε παρουσίαζουμε ένα κομμάτι του. Ομαδοποιήσαμε τις εξόδους Q ώστε μα φανεί καλύτερα ο κάθε αριθμός, και καθαρά φαίνται και οι αριθμοί που προκύπτουν από τα LED.

Το σύστημα μας λειτουργεί σωστά, καθώς λειτουργούν οι μεταβάσεις κ η μέτρηση γίνεται κανονικά (αν φαινόταν όλη ο προσομοίωση θα γινόταν μέτρηση μεχρι το 9999).

Τα radix που χρησιμοποιήθηκαν είναι τα εξής:

Q3, Q2, Q1, Q0:

radix define States_1 {

4'b0000 "0",

4'b0011 "1",

4'b0010 "2",

4'b0011 "3",

4'b0100 "4",

4'b0101 "5",

4'b0110 "6",

4'b0111 "7",

4'b1000 "8",

4'b1001 "9",

-default hex}

LED (H = 0):

radix define States_2 {

8'b11111100 "0",

8'b01100000 "1",

8'b11011010 "2",

8'b11110010 "3",

8'b01100110 "4",

8'b10110110 "5",

8'b10111110 "6",

8'b11100000 "7",

8'b11111110 "8",

8'b11110110 "9",

-default hex}

LED (H = 1):

radix define States_3 {

8'b11111101 "0",

8'b01100001 "1",

8'b11011011 "2",

8'b11110011 "3",

8'b01100111 "4",

8'b10110111 "5",

8'b10111111 "6",

8'b11100001 "7",

8'b11111111 "8",

8'b11110111 "9",

-default hex}

Στην παραπάνω προσομοίωση φαίνεται και η χρήση του enable σαν «ενεργοποιητή» του συστήματος μας. Η τυπική προσομοίωση που έτρεξε διήρκησε 100 μs.

ΑΣΚΗΣΗ 3

Εισαγωγικά Συμπεράσματα & Ανάλυση

Στην άσκηση 2, καλούμαστε να υλοποίησουμε ένα κύκλωμα κωδικοποίησης Hamming (12,5) καθώς επίσης ανίχνευσης και διόρθωσης σφάλματος ενός bit σε λέξεις δεδομένων 12-bit. Εν συντομία, η κωδικοποίηση σε κώδικα Hamming και η διόρθωση σφάλματος 1 bit με χρήση αυτού περιγράφεται ως εξής:

Αρχικά, εισάγεται στο σύστημα μια δυαδική λέξη 12 bit (στη δική μας περίπτωση). Η λέξη αυτή κωδικοποιείται με Hamming, και προστίθονται στην λέξη εξόδου άλλα 5 bits, τα οποία ονομάζονται parity bits. Τα parity bits τοποθετούνται στις θέσεις της λέξης που αποτελούν δυνάμεις του 2, εδώ στις θέσεις 1, 2, 4, 8, 16 (λόγω Verilog, θέσεις 0, 1, 3, 7 & 15).

Τα parity bits συμπληρώνονται με βάση τον παρακάτω πίνακα:

Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	...
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15	
Parity bit coverage	p1	✗		✗		✗		✗		✗		✗		✗		✗		✗		✗		
	p2		✗	✗			✗	✗			✗	✗			✗	✗			✗	✗		
	p4				✗	✗	✗	✗					✗	✗	✗	✗					✗	
	p8								✗	✗	✗	✗	✗	✗	✗	✗						
	p16																✗	✗	✗	✗	✗	

Πιο συγκεκριμένα, ένα parity bit παίρνει την τιμή 1 όταν το σύνολο των bit στην «ομάδα» του (π.χ. για το p1, θέσεις 1, 3, 5, 7 κ.ο.κ.) αποτελούν μονό αριθμό, ενώ την τιμή 0 αν αποτελούν ζυγό αριθμό. Σκοπός είναι, λοιπόν, αν σε κάθε «ομάδα» εκτελεσθεί πράξη XOR σε όλα τα ψηφία της, το αποτέλεσμα να είναι πάντα 0.

Ο ρόλος των parity bits που συμπληρώνουμε στο προηγούμενο βήμα είναι να εντοπίζουν ένα σφάλμα στη λέξη, με την προϋπόθεση ότι το σφάλμα αυτό θα είναι μοναδικό, και θα αποτελεί μια αλλαγή από 0 σε 1 ή από 1 σε 0.

Σε περίπτωση σφάλματος (στην συγκεκριμένη εργασία σφάλμα μπορεί να εισάγει ο «τυχαίος θόρυβος» που επίσης υλοποιούμε), υπάρχει μια διαδικασία εύρεσης και διόρθωσης σφάλματος που θα περιγραφεί στη συνέχεια.

Πρακτικά, ελέγχουμε την ορθότητα των parity bits. Πιο συγκεκριμένα, με χρήση XOR στις προαναφερθείσες «ομάδες», βλέπουμε αν το αποτέλεσμα

για κάθε μία είναι 0. Αν το αποτέλεσμα δεν είναι 0 σε οποιαδήποτε από τις «ομάδες», έχει δημιουργηθεί κάποιο σφάλμα κατά τη διάρκεια της διαδικασίας. Η θέση του σφάλματος αυτού μέσα στη λέξη φανερώνεται παίρνοντας τα αποτελέσματα της XOR σε όλες τις «ομάδες», και τοποθετώντας τα δίπλα δίπλα, ώστε να δημιουργηθεί μία νέα λέξη, η οποία μας δείχνει (στο δυαδικό φυσκά σύστημα) τη θέση του σφάλματος. Εδώ η λέξη αυτή θα είναι της μορφής p16_p8_p4_p2_p1.

Αφού βρούμε λοιπόν τη θέση της, διορθώνουμε το σφάλμα (0→1 ή 1→0), και δίνουμε σαν έξοδο του αποκωδικοποιητή την αρχική λέξη, αγνοώντας τα parity bits.

Υλοποίηση module & Περιγραφή

Το πρώτο module του κώδικα μας αποτελεί τον κωδικοποιητή Hamming, και είναι ο εξής:

*

```
module Hamming_encoder (input wire [0:11] in, reg Reset,
                        output reg [0:16] out);
integer i, j;


always @(in or posedge Reset)
 begin
  if (Reset)
   begin
    out = 0;
   end
  else
   begin
    i = 0;
    j = 0;
    while (i < 12 || j < 17)
```

```
    begin
     while (j==0 || j==1 || j==3 || j==7 || j==15)
      begin
           out[j] = 0;
           j = j + 1;
      end
     out[j] = in[i];
     j = j + 1;
     i = i + 1;
    end
    if (^(out & 17'b10101010101010101))
     out[0] = ~out[0];
    if (^(out & 17'b01100110011001100))
     out[1] = ~out[1];
    if (^(out & 17'b00011110000111100))
     out[3] = ~out[3];
    if (^(out & 17'b00000001111111100))
     out[7] = ~out[7];
    if (^(out & 17'b00000000000000011))
     out[15] = ~out[15];
   end
  end
endmodule
*
```

Είσοδο του κωδικοποιητή αποτελεί η λέξη εισόδου που θέλουμε να στείλουμε (12 bits), καθώς και το reset, σε μορφή wire. Έξοδο αποτελεί, φυσικά, η 17-bit λέξη που έχει πλέον τα parity bits ενσωματωμένα.

Το always block που περιγράφει τη λειτουργία του κωδικοποιητή «ενεργοποιείται» κάθε φορά που δίνεται μία λέξη εισόδου, ή αν δοθεί active high reset. Σε περίπτωση reset, η λέξη εξόδου αποτελείται μόνο από μηδενικά, είναι δηλαδή η λέξη 00000000000000000.

Όσον αφορά την κανονική λειτουργία του κωδικοποιητή, αρχικά ορίζουμε 2 μετρητές i και j, για προσπέλαση των λέξεων εισόδου και εξόδου. Ένα while block ορίζει την «διάρκεια» της κωδικοποίησης, και έτσι ξεκινάει η διαδικασία.

Πρώτο βήμα αποτελεί ο ορισμός όλων των parity bits στην νέα λέξη ως 0, κάτι που μπορεί να αλλάξει στη συνέχεια αλλά πρέπει να υπάρχει μια αρχικοποίηση. Εκτός του παραπάνω, «γεμίζει» και η νέα λέξη (σε όλες τις υπόλοιπες θέσεις) με την λέξη που δόθηκε αρχικά ως είσοδος.

Στη συνέχεια, η κωδικοποίηση ολοκληρώνεται με την σωστή συμπλήρωση των parity bits, με χρήση μιας XOR, καθώς και μίας πράξη AND ανάμεσα στην λέξη εξόδου κ σε μία διαφορετική δυαδική λέξη κάθε φορά.

Λίγο πιο επεξηγηματικά, συμβαίνει το εξής: Η πράξη AND μεταξύ της λέξης εξόδου και μιας 17-bit λέξης κάθε φορά έχει ως σκοπό την απομόνωση των ψηφίων που αποτελούν την «ομάδα» του κάθε parity bit. Στη συνέχεια, αφού αυτά τα ψηφία έχουν απομονωθεί, εκτελείται μία XOR επάνω τους, το αποτέλεσμα της οποία είναι το parity bit που χρειαζόμαστε. Αν λοιπόν η πράξη XOR δώσει 1 σαν αποτέλεσμα, το if block που ακολουθεί αντιστρέφει το αρχικό 0 (το οποίο είχε δοθεί σε όλα τα parity bits), ώστε να είναι η τιμή του parity bit σωστή σε κάθε περίπτωση.

Την έξοδο λοιπόν αποτελεί πλέον μια λέξη 17 bit, η οποία στη συνέχεια δίνεται σαν είσοδος στο module του random noise που ακολουθεί:

*

```
module Random_noise (input wire [0:16] word, wire Reset,
                        output reg [0:16] error_word);
integer position;
integer error;


always @(word or posedge Reset)
```

```
 begin
if (Reset)
   begin
    error_word = 0;
   end
  error_word = word;


  position = $urandom_range(0,16);
  error = $urandom_range(0,1);
  if (error == 1)
   begin
    error_word[position] = ~error_word[position];
   end
 end


endmodule
*
```

Είσοδος του παραπάνω module αποτελεί, όπως προαναφέρθηκε, η 17-bit λέξη που προήλθε από τον κωδικοποιητή Hamming, καθώς και το reset. Έξοδο αποτελεί μια 17-bit λέξη, η οποία μπορεί ή όχι να έχει κάποιο σφάλμα ενός bit κάπου στο εσωτερικό της.

Το always block «ενεργοποείται» όταν δοθεί μία είσοδος, ή όταν υπάρχει active high reset. Αν δοθεί reset, όπως πάντα, η έξοδος μηδενίζεται.

Στην ουσία, ο ρόλος του παραπάνω module είναι να αποφασίζει αν θα εισαχθεί σφάλμα στη λέξη, και αν ναι, σε ποιό σημείο της λέξης θα εισαχθεί. Έτσι, η μεταβλητή error παίρνει τυχαία την τιμή 1 ή 0 (1 → υπάρχει σφάλμα, και αντίστροφα). Αν error = 1, τότε η μεταβλητή position παίρνει τυχαία μία τιμή ανάμεσα στο 0 και το 16, και δείχνει έτσι

τη θέση μέσα στη λέξη που θα εισαχθεί το σφάλμα. Στη θέση αυτή, αντιστρέφεται το bit (1→0 ή 0→1).

Η έξοδος λοιπόν πλέον είναι μια 17-bit λέξη, η οποία υπάρχει περίπτωση να έχει ή όχι σφάλμα σε 1 bit στο εσωτερικό της. Στο τελευταίο βήμα της υλοποίησης, η λέξη αυτή εισάγεται σε έναν αποκωδικοποιητή Hamming, ώστε να διορθωθεί.

Ο κώδικας του αποκωδικοποιητή παρουσιάζεται στη συνέχεια:

*

```
module Hamming_decoder (input wire [0:16] error_word, wire Reset,
                        output reg [0:11] correct_word);


reg par1, par2, par4, par8, par16;

reg [0:4] par;

reg [0:16] middle_word;

integer i,j;


always @(error_word or posedge Reset)

 begin

  if (Reset)

   begin

    correct_word = 0;

   end

  else

   begin


    par1 = ^(error_word & 17'b10101010101010101);

    par2 = ^(error_word & 17'b01100110011001100);
```

```
  par4 = ^(error_word & 17'b00011110000111100);
  par8 = ^(error_word & 17'b00000001111111100);
  par16 = ^(error_word & 17'b00000000000000011);
  par = {par16, par8, par4, par2, par1};

  middle_word = error_word;
  middle_word[par-1] = ~middle_word[par-1]; //correction in place
  i = 0;
  j = 0;

  while (i < 12 || j < 17)
   begin
    while (j==0 || j==1 || j==3 || j==7 || j==15)
     begin
         j = j + 1;
     end
    correct_word[i] = middle_word[j];
    j = j + 1;
    i = i + 1;
   end
  end
 end

endmodule
*
```

Είσοδο του αποκωδικοποιητή αποτελεί η 17-bit λέξη (με ή χωρίς σφάλμα) και όπως πάντα το reset. Έξοδο αποτελεί η 12-bit διορθωμένη, αποκωδικοποιημένη (δηλαδή χωρίς τα parity bits) λέξη.

Το always block του αποκωδικοποιητή «ενεργοποιείται» όποτε δοθεί είσοδος στο module, αλλά και όταν δίνεται active high reset (σε αυτή την περίπτωση η λέξη εξόδου μηδενίζεται). Αρχικά, στο εσωτερικό του always block ορίζονται τα parity bits, με παρόμοιο τρόπο με τον κωδικοποιητή.

Αν δεν υπάρχει σφάλμα, αναμένεται το κάθε parity bit να έχει την τιμή 0, συνεπώς η 5-bit λέξη par θα έχει την τιμή 00000. Όταν συμβαίναι αυτό, δεν επέρχεται καμία αλλαγή στην λέξη εισόδου, πέραν της αφαίρεσης των parity bits.

Απο την άλλη, όταν υπάρχει σφάλμα στην λέξη, το 5-bit vector par μας δίνει τη θέση του σφάλματος αυτού. Συνεπώς, γίνεται διόρθωση του bit στη θέση par-1 (εκκίνηση της μέτρησης στην Verilog από το 0), και η λέξη είναι πλέον έτοιμη να προχωρήσει στην αφαίρεση των parity bits και στην έξοδο.

Τελευταίο βήμα, όπως αναφέρθηκε ήδη πιο πάνω, αποτελεί η αφαίρεση (πρακτικά γίνεται skip) των parity bits, ώστε στην έξοδο να φτάσει μόνο η 12-bit λέξη που είχε δοθεί στο πρώτο βήμα στον κωδικοποιητή.

Υλοποίηση testbench & Περιγραφή

Ο κώδικας για το testbench της 3ης και τελευταίας άσκησης παρουσιάζεται παρακάτω:

```
*

`timescale 1ns/1ns

module Hamming_TB;


reg CLK_TB;


reg [0:11]in_TB;

integer i;

reg [0:13] TestVector [0:6];
```

```
wire [0:16] out_TB;

wire [0:16] error_word;

wire [0:11] correct_word_TB;

reg Reset_TB;


Hamming_encoder DUT1 (.in(in_TB[0:11]), .out(out_TB[0:16]), .Reset(Reset_TB));

Random_noise DUT2 (.word(out_TB[0:16]), .error_word(error_word[0:16]),
.Reset(Reset_TB));

Hamming_decoder DUT3 (.error_word(error_word[0:16]), .Reset(Reset_TB),
.correct_word(correct_word_TB[0:11]));


initial
  begin
    Reset_TB = 1;
  end


initial
  begin
    CLK_TB = 1'b1;
  end


initial
  begin
    $readmemb("TestVector3.txt",TestVector);
    i = 0;
  end
```

```
always
  begin
   #5 CLK_TB = ~CLK_TB;
  end

always @(posedge CLK_TB)
  begin
    {Reset_TB, in_TB} = TestVector[i]; #5
     $display(in_TB);
     $display(correct_word_TB);
  end

always@(negedge CLK_TB)
  begin
    if(in_TB == correct_word_TB) begin
       $display("Correct output for inputs %b = %b",in_TB,correct_word_TB);
    end
    i = i + 1;
  end

endmodule
*
```

Στο testbench μας (με timescale 1ns/1ns) ορίζονται αρχικά τα: ρολόι/clock, το reset, η 12-bit λέξη εισόδου, η λέξη εξόδου του κωδικοποιητή (17-bit), η λέξη εξόδου του τυχαίου θορύβου (17-bit), η 12-bit λέξη εξόδου του αποκωδικοποιητή (και τελευταία λέξη που θέλουμε),

ένας μετρητής i για την προσπέλαση του αρχείου με τις λέξεις εισόδου, και ένα διάνυσμα TestVector που θα συμπληρωθεί διαβάζοντας τις λέξεις εισόδου (παρόμοια με το testbench της 1ης άσκησης).

Στη συνέχεια, προχωράμε στην κλήση των module που υλοποιήσαμε στο προηγούμενο βήμα, με port calls για κάθε είσοδο και έξοδο. Πιο συγκεκριμένα:

- Αρχικά καλείται ο κωδικοποιητής Hamming με λέξη εισόδου από το αρχείο TestVector3.txt (θα αναλυθεί στη συνέχεια η διαδικασία)
  Hamming_encoder DUT1 (.in(in_TB[0:11]), .out(out_TB[0:16]), .Reset(Reset_TB));
- Στη συνέχεια καλείται το module που «παράγει» τυχαίο θόρυβο και εισάγει ή όχι κάποιο σφάλμα στη λέξη εξόδου του κωδικοποιητή
  Random_noise DUT2 (.word(out_TB[0:16]), .error_word(error_word[0:16]), .Reset(Reset_TB));
- Τέλος, καλείται ο αποκωδικοποιητής και δίνεται σαν είσοδος αυτού η λέξη εξόδου που προέκυψε από τον τυχαίο θόρυβο. Αυτός είναι που δίνει και την τελική λέξη 12-bit που πρέπει να είναι όμοια με την αρχική λέξη
  Hamming_decoder DUT3 (.error_word(error_word[0:16]), .Reset(Reset_TB), .correct_word(correct_word_TB[0:11]));

Τα initial blocks που ακολουθούν δίνουν στα reset, clock και i τις τιμές 1, 1 και 0 αντίστοιχα. Φυσικά, πρέπει να σημειωθεί ότι στο τρίτο initial block, πέραν της αρχικοποίησης του i, υπάρχει και η ανάγνωση του αρχείου TestVector3.txt και η αποθήκευση του περιεχομένου του στο vector TestVector, που θα χρησιμοποιηθεί στη συνέχεια.

Η περίοδος του ρολογιού, εδώ ίση με 10 ns, ορίζεται στο always block που ακολουθεί. Στη συνέχεια, ορίζεται ένα always block που «ενεργοποιείται» σε κάθε θετική ακμή του ρολογιού, και ρόλος του είναι να διαβάζει την λέξη εισόδου, καθώς και το reset από το TestVector, και να εμφανίζει την λέξη αυτή, καθώς και την λέξη που προέκυψε μετά τη διαδικασία, ώστε να ελέγξουμε την ορθότητα της και μόνοι μας (αναμένουμε να είναι ίδιες).

Τέλος, το always block που ακολουθεί «ενεργοποιείται» με κάθε αρνητική ακμή του ρολογιού, και σκοπός του είναι να συγκρίνει την αρχική λέξη εισόδου με την λέξη εξόδου του αποκωδικοποιητή, και να εμφανίσει το ανάλογο μήνυμα αν η διαδικασία πέτυχε, καθώς και να αυξάνει κάθε φορά τον μετρητή i, ώστε να διαβάζεται σωστά το TestVector.

Το TestVector.txt είναι συμπληρωμένο, πιο αναλύτικα, με κάθε σειρά να είναι της μορφής Reset_InitialWord, και οι τιμές που έχουμε δώσει για τον έλεγχο του συστήματος είναι οι εξής:

1_000000000000

0_111000100100

0_111000100101

0_111000100110

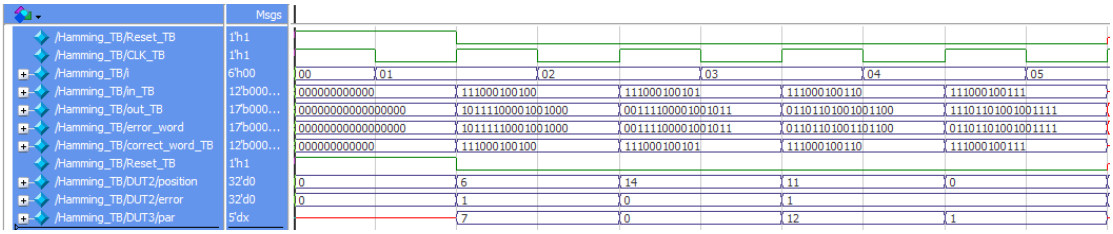
0_111000100111

0_100010110101

0_110110100100

Προσωμοίωση & Παρατηρήσεις

Μετά από προσομοίωση προκύπτει το εξής διάγραμμα:



Στο διάγραμμα αυτό φαίνεται η λέξη για κάθε βήμα της διαδικασίας, καθώς το αν εισάγεται ή όχι σφάλμα, η θέση του σφάλματος, και η θέση του σφάλματος όπως προκύπτει από την αποκωδικοποίηση Hamming (vector **par**, εδώ μία τιμή μεγαλύτερη από τη σωστή θέση κάθε φορά λόγω του συστήματος μέτρησης της Verilog).

Η παραπάνω προσομοίωση διαρκεί 50 ns, καθώς πρόκειται για 5 λέξεις, με περίοδο ρολογιού ίση με 10 ns. Να σημειωθεί, τέλος, ότι επιλέγονται κατάλληλα radix από τα προϋπάρχοντα του ModelSim για κάθε τιμή, ώστε να φαίνεται πιο ξεκάθαρα η διαδικασία.