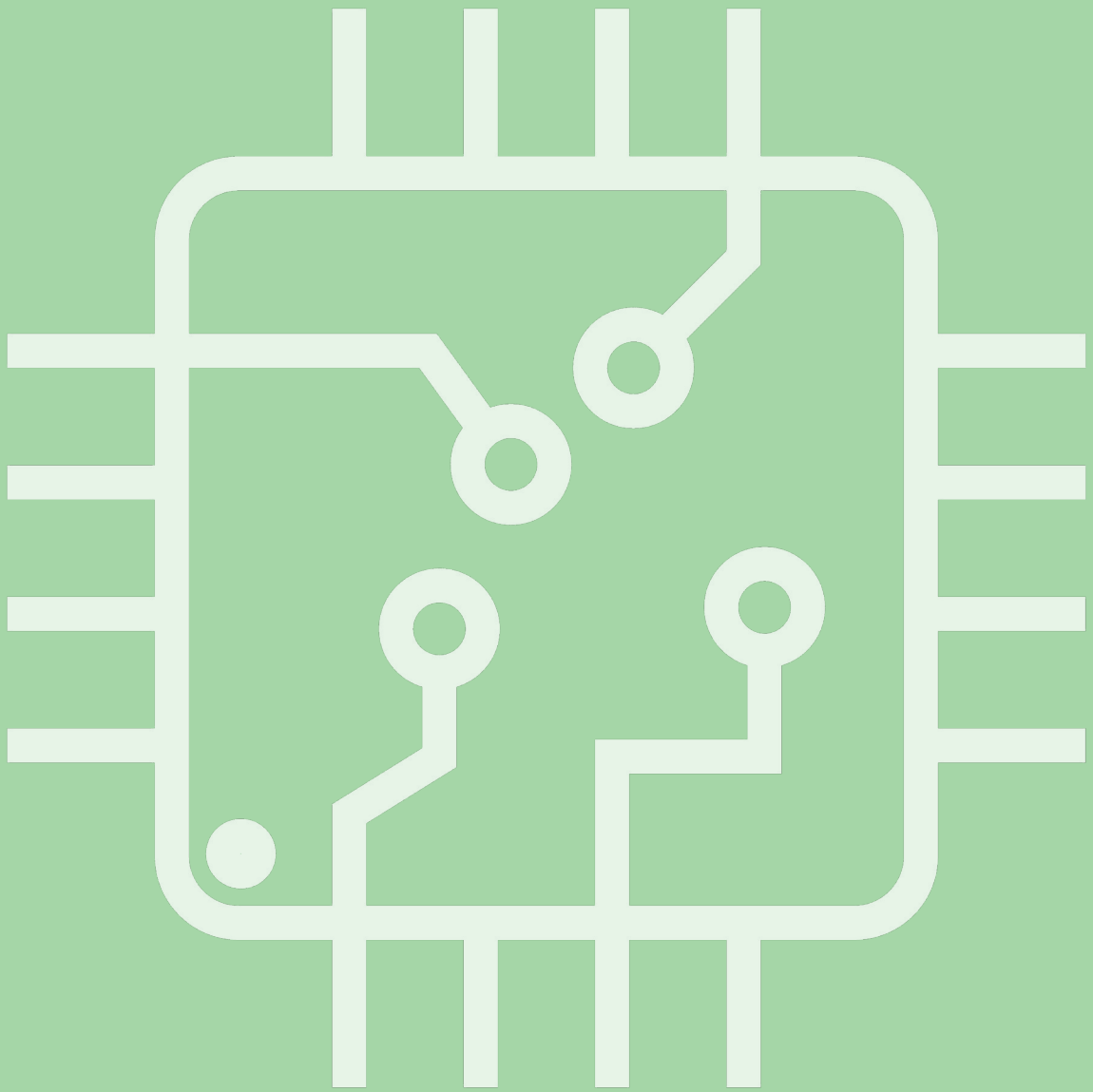
Ψηφιακά Συστήματα HW σε Χαμηλά Επίπεδα Λογικής II

Αναφορά Εργασίας

Αβραμίδης Παναγιώτης

Ιούνιος-Ιούλιος 2021



Άσκηση 1

Η πρώτη άσκηση ζητάει την υλοποίηση ενός Finite State Machine (FSM) με τρεις διαφορετικούς τρόπους:

α) συμπεριφορική (behavioral) verilog

β) χρησιμοποιώντας D flip-flop

γ) χρησιμοποιώντας JK flip-flop

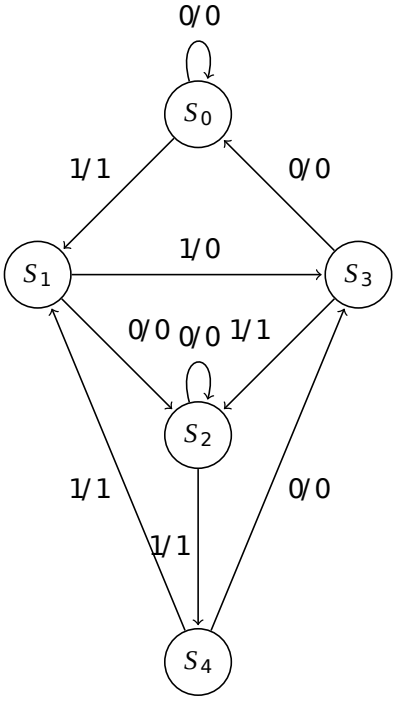
α) Συμπεριφορική Verilog

Βασίστηκε στο παράδειγμα του εργαστηρίου.

β) Δομική περιγραφή με D Flip-Flop

Κωδικοποίηση Καταστάσεων

Αρχικά ονοματίζω κάθε κατάσταση: $S_0 \rightarrow 001, S_1 \rightarrow 100$ και $S_2 \rightarrow 010, S_3 \rightarrow 011, S_4 \rightarrow 000$ Η κατάσταση S_0 επιλέχθηκε να είναι το σημείο εκκίνησης του FSM ενώ οι υπόλοιπες χωρίς κάποια άλλη λογική, δηλαδή τυχαία. Στα δεξιά απεικονίζεται το μηχάνημα με τα ονόματα που επιλέχθηκαν. Για τη σχεδίαση του διαγράμματος χρησιμοποιήθηκε το πακέτο TikZ της latex [4].



Στη συνέχεια παρατηρώντας το διάγραμμα μετάβασης καταστάσεων κατασκευάζω τον αντίστοιχο πίνακα καταστάσεων και εξόδου του FSM:

State	State number (D)	Input	Output	Next State:S'
S_4	000	0	0	S_3:011
S_4	000	1	1	S_1:100
S_0	001	0	0	S_0:001
S_0	001	1	1	S_1:100
S_2	010	0	0	S_2:010
S_2	010	1	1	S_4:000
S_3	011	0	0	S_0:001
S_3	011	1	1	S_2:010
S_1	100	0	0	S_2:010
S_1	100	1	0	S_3:011
-	101	0	X	X
-	101	1	X	X
-	110	0	X	X
-	110	1	X	X
-	111	0	X	X
-	111	1	X	X

Αξίζει να σημειωθεί ότι η επιλογή κωδικοποίησης καταστάσεων δεν διευκρινίζεται από την εκφώνηση, επομένως αφήνεται στην κρίση του φοιτητή. Τρεις είναι οι πιο συχνοί τρόποι κωδικοποίησης καταστάσεων των FSM : binary, one-hot, και Gray coding, ενώ υπάρχει εκτενής έρευνα για επιπλέον τρόπους και αλγόριθμους. Συνοπτικά, η δυαδική κωδικοποίηση ελαχιστοποιεί των αριθμό των flip-flops, ενώ η one-hot τον μεγιστοποιεί. Το μειονέκτημα της δυαδικής κωδικοποίησης είναι να απαιτεί περισσότερες λογικές πύλες, που απαιτούνται για την αποκωδικοποίηση. Η κωδικοποίηση Gray αν και απαιτεί τον ίδιο αριθμό flip-flops με αυτήν της δυαδικής, έχει σαν αποτέλεσμα λιγότερα glitches και μικρότερη κατανάλωση καθώς μόνο ένα bit αλλάζει με μία μετάμαση κατάστασης. [5]

Τελικά θα επιλέγεται δυαδική κωδικοποίηση, για την ελαχιστοποίηση των flip-flops. Έχουμε 5 διακριτές καταστάσεις συνολικά, άρα 3 bits θα είναι αρκετά για να τις κωδικοποιήσουμε. Ακόμη, η έξοδος παίρνει δύο διακριτές καταστάσεις και άρα ένα bit είναι αρκετό. Άρα οι καταστάσεις και οι έξοδοι είναι ήδη ελαχιστοποιημένες.

Λογικές Εξισώσεις

Στη συνέχεια θα διατυπώνωνται τις λογικές εξισώσεις των καταστάσεων χρησιμοποιώντας τους εξής συμβολισμούς: Το D_i είναι το i-οστό ψηφίο της κατάστασης (3 συνολικά) και T είναι το bit της εισόδου. Με τον τόνο συμβολίζεται η επόμενη κατάσταση. Οι εξισώσεις προκύπτουν από τον πίνακα καταστάσεων.

$$D_0' = \overline{D_2}\,\overline{D_1}\,\overline{D_0}\,T + \overline{D_2}\,\overline{D_1}\,D_0\,\overline{T} + \overline{D_2}\,D_1\,D_0\,\overline{T} + D_2\,\overline{D_1}\,\overline{D_0}\,T$$

$$D_1' = \overline{D_2}\,\overline{D_1}\,\overline{D_0}\,T + \overline{D_2}\,D_1\,\overline{D_0}\,\overline{T} + \overline{D_2}\,D_1\,D_0\,T + D_2\,\overline{D_1}\,\overline{D_0}\,\overline{T} + D_2\,\overline{D_1}\,\overline{D_0}\,T$$

$$D_2' = \overline{D_2}\,\overline{D_1}\,\overline{D_0}\,T + \overline{D_2}\,\overline{D_1}\,D_0\,T$$

Στη συνέχεια προχωράμε στην ελαχιστοποίησή τους. Για το D_0 έχουμε:

$$D_0' = \overline{D_2}\,\overline{D_1}\,\overline{D_0}\,T + \overline{D_2}\,\overline{D_1}\,D_0\,\overline{T} + \overline{D_2}\,D_1\,D_0\,\overline{T} + D_2\,\overline{D_1}\,\overline{D_0}\,T \Leftrightarrow$$

$$D_0' = \overline{D_2}\,\overline{D_1}\,T\,(D_0 + \overline{D_0}) + \overline{D_2}\,D_1\,D_0\,\overline{T} + D_2\,\overline{D_1}\,\overline{D_0}\,T \Leftrightarrow$$

$$D_0' = \overline{D_2}\,\overline{D_1}\,T + \overline{D_2}\,T\,D_1 D_0 + D_2\,T\,\overline{D_1}\,\overline{D_0} \Leftrightarrow$$

$$D_0' = \overline{D_2}\,T\,(\overline{D_1} + D_1 D_0) + D_2\,T\,\overline{D_1}\,\overline{D_0} \Leftrightarrow$$

$$D_0' = \overline{D_2}\,T\,D_0 + \overline{D_2}\,T\,\overline{D_1} + D_2\,T\,\overline{D_1}\,\overline{D_0}$$

Για το D_1 έχουμε:

$$D_1' = \overline{D_2}\,\overline{D_1}\,\overline{D_0}\,T + \overline{D_2}\,D_1\,\overline{D_0}\,\overline{T} + \overline{D_2}\,D_1\,D_0\,T + D_2\,\overline{D_1}\,\overline{D_0}\,\overline{T} + D_2\,\overline{D_1}\,\overline{D_0}\,T \Leftrightarrow$$

$$D_1' = \overline{D_1}\,\overline{D_0}\,T\,(D_2 + \overline{D_2}) + \overline{D_2}\,D_1\,\overline{D_0}\,\overline{T} + \overline{D_2}\,D_1\,D_0\,T + D_2\,\overline{D_1}\,\overline{D_0}\,T \Leftrightarrow$$

$$D_1' = \overline{D_1}\,\overline{D_0}\,T + \overline{D_2}\,D_1\,\overline{D_0}\,\overline{T} + \overline{D_2}\,D_1\,D_0\,T + D_2\,\overline{D_1}\,\overline{D_0}\,T \Leftrightarrow$$

$$D_1' = \overline{D_1}\,\overline{D_0}\,(\overline{T} + D_2\,T) + \overline{D_2}\,D_1\,\overline{D_0}\,\overline{T} + \overline{D_2}\,D_1\,D_0\,T \Leftrightarrow$$

$$D_1' = \overline{D_1 D_0}(\overline{T} + D_2) + \overline{D_2} D_1 \overline{D_0 T} + \overline{D_2} D_1 D_0 T \Leftrightarrow$$

$$D_1' = \overline{D_1 D_0 T} + D_2 \overline{D_1 D_0} + \overline{D_2} D_1 \overline{D_0 T} + \overline{D_2} D_1 D_0 T \Leftrightarrow$$

$$D_1' = \overline{D_0 T}(\overline{D_1} + D_1 \overline{D_2}) + \overline{D_2} D_1 D_0 T + D_2 \overline{D_1 D_0} \Leftrightarrow$$

$$D_1' = \overline{D_0 T}(\overline{D_1} + \overline{D_2}) + \overline{D_2} D_1 D_0 T + D_2 \overline{D_1 D_0} \Leftrightarrow$$

$$D_1' = \overline{D_1 D_0 T} + \overline{D_2 D_0 T} + \overline{D_2} D_1 D_0 T + D_2 \overline{D_1 D_0} \Leftrightarrow$$

$$D_1' = \overline{D_2 D_0 T} + \overline{D_2} D_1 D_0 T + D_2 \overline{D_1 D_0}$$

Στο τελευταίο βήμα ο πρώτος όρος καλύπτεται από τους 2 και 4 και για αυτό απαλείφθηκε.

Προχωράμε με το D_2

$$D_2' = \overline{D_2 D_1 D_0} T + \overline{D_2 D_1} D_0 T \Leftrightarrow$$

$$D_2' = \overline{D_2 D_1 D_0} T + \overline{D_2 D_1} D_0 T = \overline{D_2 D_1} T(\overline{D_0} + D_0) \Leftrightarrow$$

$$D_2' = \overline{D_2 D_1} T$$

Συγκεντρωτικά οι εξισώσεις καταστάσεων θα είναι: (δεν λήφθηκαν υπόψιν τ don't cares από αρχική αβλεψία. Στο JK λήφθηκαν)

$$D_0' = \overline{D_2 T} D_0 + \overline{D_2 T} D_1 + D_2 T \overline{D_1 D_0}$$

$$D_1' = \overline{D_2 D_0 T} + \overline{D_2} D_1 D_0 T + D_2 \overline{D_1 D_0}$$

$$D_2' = \overline{D_2 D_1} T$$

Οι σχετικοί πίνακες Karnaugh δημιουργήθηκαν στο texworks με βάση τον κώδικα από εδώ[3] , τροποποιημένο ελαφρώς.

D_0'

D_1D_2 \ D_0T	00	01	11	10
00	1	0	0	1
01	0	0	0	1
11	0	0	0	0
10	0	1	0	0

D_1'

D_1D_2 \ D_0T	00	01	11	10
00	1	0	0	0
01	1	0	1	0
11	0	0	0	0
10	1	1	0	0

D_2'

D_1D_2 \ D_0T	00	01	11	10
00	0	1	1	0
01	0	0	0	0
11	0	0	0	0
10	0	0	0	0

Για την έξοδο, από τον πίνακα καταστάσεων επίσης ισχύει:

$$Y = \overline{D_2 D_1 D_0} T + \overline{D_2 D_1} D_0 T + \overline{D_2} D_1 \overline{D_0} T + \overline{D_2} D_1 D_0 T \Leftrightarrow$$

$$Y = \overline{D_2 D_1} T(D_0 + \overline{D_0}) + \overline{D_2} D_1 T(D_0 + \overline{D_0}) \Leftrightarrow$$

$$Y = \overline{D_2 D_1} T + \overline{D_2} D_1 T \Leftrightarrow \quad Y = \overline{D_2} T$$

Y

D_1D_2 \ D_0T	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	0	0	0	0
10	0	0	0	0

Πρότυπο D Flip-Flop

Η verilog δεν διαθέτει primitives για latches και flip-flops. Επομένως θα πρέπει να συνθέσουμε εμείς δικά μας module που να περιγράφουν τη λειτουργία τους. Για το σκοπό αυτό, έγινε χρήση του κώδικα που παρατίθεται στις σημειώσεις του μαθήματος (Verilog – Μέρος ΙΙΙ). Για την αρχικοποίηση του FSM στην κατάσταση $S_0 \rightarrow 001$ έγινε χρήση των σημάτων Reset και Preset.

γ) Δομική περιγραφή με JK Flip-Flop

Για την κωδικοποίηση καταστάσεων δεν χρειάζεται να γίνει αλλαγή, καθώς αντιστοιχίζεται και πάλι ένα δυαδικό ψηφίο σε κάθε flip-flop. Ωστόσο χρειάζεται προσοχή στο γεγονός ότι σε αντίθεση με το D flip-flop, το JK έχει δύο εισόδους και άρα δεν υπάρχει ένα προς ένα αντιστοίχιση μεταξύ εισόδων και εξόδων του κάθε flip-flop. Για αυτό θα χρειαστεί υπολογισμός των λογικών εξισώσεων από την αρχή, με επίσης εκ νέου κατασκευή του πίνακα καταστάσεων.

Το πως γίνεται η κατασκευή φαίνεται επίσης εδώ [1] (5b), όπου θα πρέπει να βρεθούν τα κατάλληλα inputs των JK FF που παράγουν τις αντίστοιχες μεταβάσεις καταστάσεων. Αυτό γίνεται με τη βοήθεια του Excitation Table του JK flip-flop και τον πίνακα καταστάσεων που ήδη κατασκευάστηκε για το προηγούμενο ερώτημα. Αυτό δίνει την πληροφορία του ποιες θα πρέπει να είναι οι τιμές των JK για κάθε bit της κατάστασης. Για κάθε bit χρειαζόμαστε ένα flip-flop, άρα θα έχουμε 3 JK flip-flop. Ο πίνακας που προκύπτει είναι ο ακόλουθος.

Excitation Table				
Q	Q_{next}	Σχόλιο	J	K
0	0	No Change	0	X
0	1	Set	1	X
1	0	Reset	X	1
1	1	No Change	X	0

Πίνακας Μετάβασης Καταστάσεων με JK flip-flop								
State number (D)	Input(T)	Next State:S'	J2	K2	J1	K1	J0	K0
000	0	S_3:011	0	X	1	X	1	X
000	1	S_1:100	1	X	0	X	0	X
001	0	S_0:001	0	X	0	X	X	0
001	1	S_1:100	1	X	0	X	X	1
010	0	S_2:010	0	X	X	0	0	X
010	1	S_4:000	0	X	X	1	0	X
011	0	S_0:001	0	X	X	1	X	0
011	1	S_2:010	0	X	X	0	X	1
100	0	S_2:010	X	1	1	X	0	X
100	1	S_3:011	X	1	1	X	1	X
101	0	X	X	X	X	X	X	X
101	1	X	X	X	X	X	X	X
110	0	X	X	X	X	X	X	X
110	1	X	X	X	X	X	X	X
111	0	X	X	X	X	X	X	X
111	1	X	X	X	X	X	X	X

Λογικές Εξισώσες για το JK FF

Όπως είναι εμφανές έχουμε 6 συνολικά εισόδους, άρα θα χρειαστούμε 6 εξισώσεις. Για την κατασκευή των εξισώσεων αυτή τη φορά χρησιμοποιήθηκε η μέθοδος πινάκων Karnaugh, με παρατήρηση του πίνακα μετάβασης καταστάσεων. Οι 6 πίνακες που τελικά προκύπτουν είναι οι εξής:

J_2

D_2D_1 \ D_0T	00	01	11	10
00	0	1	1	0
01	0	0	0	0
11	X	X	X	X
10	X	X	X	X

K_2

D_2D_1 \ D_0T	00	01	11	10
00	X	X	X	X
01	X	X	X	X
11	X	X	X	X
10	1	1	X	X

J_1

D_2D_1 \ D_0T	00	01	11	10
00	1	0	0	0
01	X	X	X	X
11	X	X	X	X
10	1	1	X	X

K_1

D_2D_1 \ D_0T	00	01	11	10
00	X	X	X	X
01	0	1	0	1
11	X	X	X	X
10	X	X	X	X

J_0

D_2D_1 \ D_0T	00	01	11	10
00	1	0	X	X
01	0	0	X	X
11	X	X	X	X
10	0	1	X	X

K_0

D_2D_1 \ D_0T	00	01	11	10
00	X	X	1	0
01	X	X	1	0
11	X	X	X	X
10	X	X	X	X

Οι ελαχιστοποιημένες εξισώσεις των JK πλέον βγαίνουν άμεσα:

$$J_2=\overline{D_1}T, K_2=1 \quad , \quad K_2=1$$

$$J_1=\overline{D_0T}+D_2, K_1=\overline{D_0}T+D_0\overline{T}=D_0\oplus T$$

$$J_0=\overline{D_2D_1T}+D_2T, K_0=T$$

Πρότυπο JK Flip-Flop

Για το JK flip-flop, αντί για σχεδιασμό από την αρχή επιλέχθηκε να σχεδιαστεί με βάση αυτό του D flip-flop. Πιο συγκεκριμένα, έγινε μετατροπή από το D στο JK με βάση τις εξισώσεις που προκύπτουν από την ανάλυση εδώ [6]. Η τελική εξίσωση είναι η εξής: $D = J\overline{Q_n} + \overline{K}Q_n$.

Το τελικό σχέδιο αποτυπώνεται με βάση την αρχή την δομικής ιεραρχικής σχεδίασης.

Κώδικες και προσομοιώσεις πρώτης άσκησης

Για το (α)

```
// Behavioural State Machine.
module ask1_a_fsm (x_in,y_out,clk,reset);
    input     x_in,clk,reset;
    output    y_out;

localparam [2:0]
    S0 = 3'b001,
    S1 = 3'b100,
    S2 = 3'b010,
    S3 = 3'b011,
    S4 = 3'b000;
    reg[2:0] currentState,nextState;

always @(posedge clk)
begin
    if(reset)     currentState <=S0;
    else          currentState <=nextState;
end
always @*
    case(currentState)
    S0: if(x_in) nextState=S1;
        else  nextState=S0;
    S1: if(x_in) nextState=S3;
        else  nextState=S2;
    S2: if(x_in) nextState=S4;
        else  nextState=S2;
    S3: if(x_in) nextState=S2;
        else  nextState=S0;
    S4: if(x_in) nextState=S1;
        else  nextState=S3;
    default:    nextState = S0;
    endcase

    assign y_out = (x_in == 1'b1 & currentState != S1);

endmodule
```

Για το (β)

```
// Structural State Machine (D Flip-Flop).
module ask1_b_fsm (x_in,y_out,clk,reset);
	input	x_in,clk,reset;
	output	y_out;

	wire D2, D1, D0, T; //States and inputs
	wire y_out;
	wire D2n, D1n, D0n, Tn;
	//wire resetn;

	buf buf1 (T, x_in);
	not R (resetn, reset);

	not U0 (D2n, D2);
	not U1 (D1n, D1);
	not U2 (D0n, D0);
	not Ui (Tn, T);

	// Product terms of D0
	and U4 (s1, D2n, Tn, D0);
	and U5 (s2, D2n, Tn, D1n);
	and U6 (s3, D2, T, D1n, D0n);

	// Product terms of D1
	and U7 (s4, D2n, D0n, Tn);
	and U8 (s5, D2n, D1, D0, T);
	and U9 (s6, D2, D1n, D0n);

	// Product term of D2
	and U10 (s7, D2n, D1n, T);

	// Sum of products for D0 and D1
	or U11 (p0 , s1, s2, s3);
	or U12 (p1 , s4, s5, s6);

	// Store FSM information in D fLip-flops
	dflipflop SD0 (.Q(D0), .Clock(clk), .Reset(1'b1), .Preset(resetn), .D(p0));
	dflipflop SD1 (.Q(D1), .Clock(clk), .Reset(resetn), .Preset(1'b1), .D(p1));
	dflipflop SD2 (.Q(D2), .Clock(clk), .Reset(resetn), .Preset(1'b1), .D(s7));

	// FSM output

	and (y_out, D2n, T);
endmodule
```

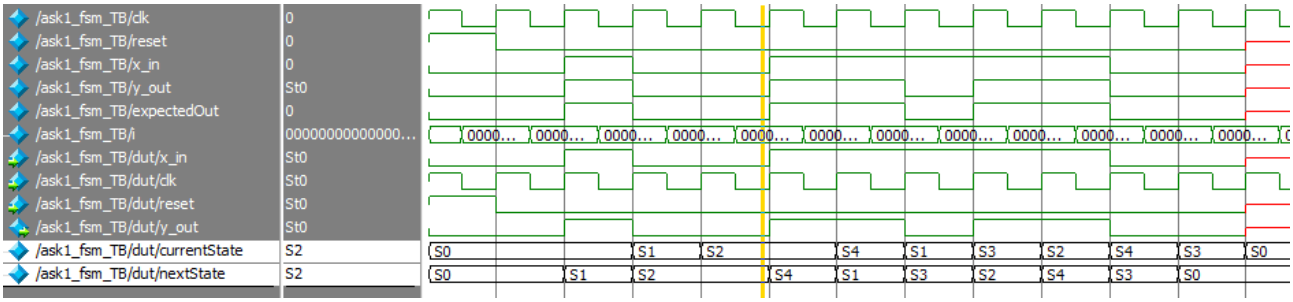
Για το (γ)

```
// Structural State Machine (D Flip-Flop).
module ask1_c_fsm (x_in,y_out,clk,reset);
	input	x_in,clk,reset;
	output	y_out;

	wire D2, D1, D0, T; //States and inputs
	wire y_out;
	wire D2n, D1n, D0n, Tn;

	wire J2, K2, J1, K1, J0, K0;
	//wire J2n, K2n, J1n, K1n, J0n, K0n;

	buf buf1 (T, x_in);
	not R (resetn, reset);

	not U0 (D2n, D2);
	not U1 (D1n, D1);
	not U2 (D0n, D0);
	not Uin (Tn, T);

	// J2
	and U4 (J2, D1n, T);
	// set K2

	// J1
	and U5 (s1, D0n, Tn);
	or P1 (J1, s1, D2);
	// K1
	xor X1 (K1, D0, T);

	// J0
	and U6 (s2, D2n, D1n, Tn);
	and U7 (s3, D2, T);
	or P2 (J0, s2, s3);
	// K0
	buf B1 (K0, T);

	// Store FSM information in D fLip-flops
	jkflipflop SD0 (.Q(D0), .Clock(clk), .Reset(1'b1), .Preset(resetn), .J(J0), .K(K0));
	jkflipflop SD1 (.Q(D1), .Clock(clk), .Reset(resetn), .Preset(1'b1), .J(J1), .K(K1));
	jkflipflop SD2 (.Q(D2), .Clock(clk), .Reset(resetn), .Preset(1'b1), .J(J2), .K(1'b1));

	// FSM output
	and (y_out, D2n, T);
endmodule
```

Για το testbench (κοινό σε όλα)

```
`timescale 1ns/1ns
module ask1_fsm_TB;

    reg clk;
    reg reset;
    reg x_in;

    wire y_out;
    reg expectedOut;
    reg [31:0] i;

    ask1_a_fsm dut(x_in,y_out,clk,reset);

    reg [2:0] testVector[12:0];

    initial
    begin
        $readmemb("TestBenchVector",testVector);
        i=0;
        reset=1; x_in=0;
    end
    always@(posedge clk)
    begin
        {reset,x_in,expectedOut}=testVector[i];#10;
        $display(reset,x_in,expectedOut);
        end
    always@(negedge clk)
        begin
            if(expectedOut !== y_out) begin
                $display("Wrong output for inputs %b, %b!=%b",
                {reset,x_in},expectedOut,y_out);
            end
            i=i+1;
        end
    always
        begin
            clk <=1; #5;
            clk <=0; #5;
        end
endmodule
```

Με την εντολή “radix define States {3'b001 "S0",3'b100 "S1",3'b010 "S2",3'b011 "S3",3'b000 "S4",-default hex}”



Άσκηση 2

Για την άσκηση 2 ζητείται ένα σύστημα τετραπλού μετρητή BCD με T flip-flop και ένας αποκωδικοποιητής 7-segment για κάθε ψηφίο, ιεραρχικά δομημένος κώδικας, με T- flip-flop.

Για τη υλοποίηση ακολουθήθηκε η εξής ιεραρχία: Ένα module για ένα μετρητή BCD ενός ψηφίου, ένα για έναν αποκωδικοποιητή ενός ψηφίου και ένα για τοτελικό σχέδιο με 4 binary to bcd counters και 4 απκωδικοποιητές. Για την επαλήθευση κάθε module έγινε το κατάλληλο testbench.

Κώδικας Verilog για το T flip-flop.

Για την υλοποίηση του flip-flop χρησιμοποιήθηκε το primitive module του D flip-flop της άσκησης 1. Η μεθοδολογία μετατροπής είναι διαθέσιμη εδώ [8]. Απλά μία xor πύλη οδηγεί της έξοδο Q και της είσοδο T, στην είσοδο D του D flip-flop.

```
// T FLIP-FLOP (D_ff_with_asynch_reset_behavior)
module tflipflop(output Q, Qn,
                 input wire Clock, Clear, Preset, T);

    wire x;

    xor (x, Q, T);


    dflipflop DFF (.Q(Q), .Qn(Qn), .Clock(Clock), .Reset(Clear), .Preset(Preset),
.D(x));
```

Truth Table of T Flip-flop

Input	Outputs	
	Present State	Next State
T	Q_n	Q_{n+1}
0	0	0
0	1	1
1	0	1
1	1	0

T Input	Outputs		D Input
	Present State	Next State	
T	Q_n	Q_{n+1}	D
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

D to T Conversion Table

Excitation Table of D Flip-flop

Outputs		Input
Present State	Next State	
Q_n	Q_{n+1}	D
0	0	0
0	1	1
1	0	0
1	1	1

Κώδικας Verilog για το μονό BCD.

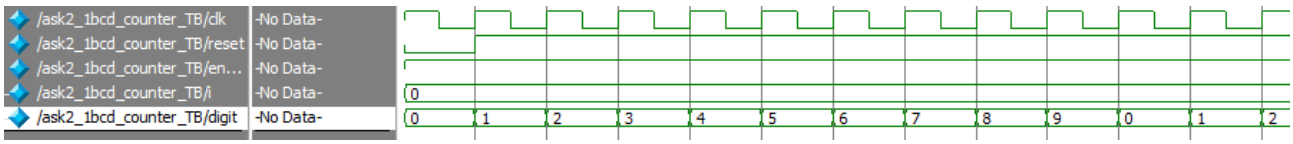
Για το μονό BCD έχουμε 4 flip-flop T που μετράνε μέχρι 4 το 16 και μερικές πύλες συνθέτουν το σύγχρονο μετρητή, μερικές που ελέγχουν την υπερχείλιση στο 10, και για το enable bit.

```
// A single digit BCD counter
module ask2_1bcd_counter (output wire [3:0]digit,
                          input wire clk, reset, enable);

    parameter
        ONE = 1'b1,
        ZERO = 1'b0;

    wire qn1, qn2, qn3, qn4;
    wire s1;
    wire t3, t4, internal;

    //buf (internal, reset); // deactivate reset function.
```

Για το testbench έχουμε:

```
`timescale 1ns/1ns
module ask2_1bcd_counter_TB;

    reg clk;
    reg reset;
    reg enable;


    reg [31:0] i;

    wire [3:0]digit;

    ask2_1bcd_counter bcd_single(digit,clk,reset, enable);


    initial
    begin
        i=0;
        enable = 1;
        reset=0;
        #10;
        reset = 1;
        #130;
        enable = 0;
        #30
        enable = 1;

    end

    always
        begin
            clk <=1; #5;
            clk <=0; #5;
        end
endmodule
```

Το αποτέλεσμα της προσομοίωσης:



Κώδικας Verilog για το 7-segment decoder

```
module ask2_1bcd_to_7segm(output wire [6:0] seg_out,
                          input wire [3:0] bcd, wire LED_type_ctl);

    // set LED_type_ctl = 1 for common anode.
    reg [6:0] decoded;

    //assign

//always block for converting bcd digit into 7 decodedment format
  always @(bcd)
  begin
    case (bcd) //case statement
      0 : decoded = 7'b0000001;
      1 : decoded = 7'b1001111;
      2 : decoded = 7'b0010010;
      3 : decoded = 7'b0000110;
      4 : decoded = 7'b1001100;
      5 : decoded = 7'b0100100;
      6 : decoded = 7'b0100000;
      7 : decoded = 7'b0001111;
      8 : decoded = 7'b0000000;
      9 : decoded = 7'b0000100;
      //switch off 7 decodedment character when the bcd digit is not a
decimal number.
      default : decoded = 7'b1111111;
    endcase
  end

    assign seg_out = LED_type_ctl ? decoded : ~decoded;

endmodule
```

Ο πίνακας των εξόδων για κοινής ανόδου:

Digit	a	b	c	d	e	f	g
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1

Και το αντίστοιχο testbench. Ο κώδικας δοκιμάζει όλα τις εισόδων των ακέραιων ως 0-9 και για τα δύο είδη (κοινής ανόδου και κοινής καθόδου)

```
`timescale 1ns/1ns
module ask2_1bcd_to_7segm_TB;

    reg [3:0] bcd;
    wire [6:0] seg;
    integer i, b;
    reg cathode_type;
    reg [6:0] TBVector_ask3_7seg[19:0];

    // Instantiate the Unit Under Test (UUT)
    ask2_1bcd_to_7segm uut (.bcd(bcd), .seg_out(seg), .LED_type_ctl(cathode_type));

    //Apply inputs
    initial begin
    $readmemb("TBVector_ask3_7seg",TBVector_ask3_7seg);
    cathode_type = 1'b0;

        for (b=0; b<2; b=b+1) begin
            cathode_type = b==0 ? 1'b1 : 1'b0;
            $display ("Cathode type: %b", cathode_type);
            for(i = 0;i < 10;i = i+1) //run loop for 0 to 15.
            begin

                bcd = i ;
                #10; //wait for 10 ns
                if(seg !== TBVector_ask3_7seg[i+ 10 * !cathode_type]) begin
                    $display("Wrong output for cathode_type: %b, i: %d, seg: %b, vector: %b"
                    ,cathode_type,i, seg, TBVector_ask3_7seg[i]);
                end else begin
                    $display("Test %d passed!", i);
                end

            end
        end
```

Κώδικας για το τελικό module της τρίτης άσκησης

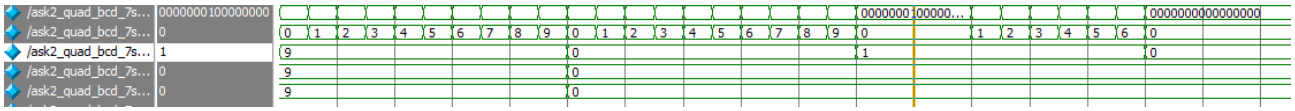
```
// A single digit BCD counter
module ask2_quad_bcd_7seg (output wire [15:0]digits, [23:0]seg_out,
                           input wire clk, reset, enable);


    wire [3:0] digit1, digit2, digit3, digit4;
    wire a1, a2, a3;
    assign digits = {digit1, digit2, digit3, digit4};

    and A1(a1, digit1[1], digit1[3]);
    and A2(a2, digit2[1], digit2[3]);
    and A3(a3, digit3[1], digit3[3]);


    ask2_1bcd_counter C1(.digit(digit1), .clk(clk), .reset(reset), .enable(enable));
    ask2_1bcd_counter C2(.digit(digit2), .clk(a1), .reset(reset), .enable(enable));
    ask2_1bcd_counter C3(.digit(digit3), .clk(a2), .reset(reset), .enable(enable));
    ask2_1bcd_counter C4(.digit(digit4), .clk(a3), .reset(reset), .enable(enable));


    wire LED_type_ctl;

    wire [6:0] seg_out1, seg_out2, seg_out3, seg_out4;
    assign seg_out = {seg_out1, seg_out2, seg_out3, seg_out4};

    ask2_1bcd_to_7segm S1(.seg_out(seg_out1), .bcd(digit1), .LED_type_ctl(LED_type_ctl));
    ask2_1bcd_to_7segm S2(.seg_out(seg_out2), .bcd(digit2), .LED_type_ctl(LED_type_ctl));
    ask2_1bcd_to_7segm S3(.seg_out(seg_out3), .bcd(digit3), .LED_type_ctl(LED_type_ctl));
    ask2_1bcd_to_7segm S4(.seg_out(seg_out4), .bcd(digit4), .LED_type_ctl(LED_type_ctl));


endmodule
```

Και το τελικό testbench:

```
`timescale 1ns/1ns
module ask2_quad_bcd_7seg_TB;
	//sim 100400 ns
	reg clk;
	reg reset;
	reg enable;
	wire [15:0]digits;
	wire [3:0] digit1, digit2, digit3, digit4;
	assign {digit1, digit2, digit3, digit4} = digits;
	wire [23:0]seg_out;
	wire [6:0] seg_out1, seg_out2, seg_out3, seg_out4;
	assign {seg_out1, seg_out2, seg_out3, seg_out4} = seg_out;
	integer i;

	ask2_quad_bcd_7seg ask3_final_quad(digits,seg_out,clk,reset, enable);

	initial
	begin
		i = -2;
		enable = 1;
		reset=0;
		#10;
		reset = 1;
		#100100;
		enable = 0;
		#30
		enable = 1;
		#20
		reset = 1;
		#40;
		reset = 0;
	end

	always@(posedge clk)
	begin
		i = i +1;
		$display("i: %d, Digits: %d%d%d%d    %d.",i, digit4, digit3, digit2, digit1,
digit4*1000+digit3*100+digit2*10+digit1);

		if (i == digit4*1000+digit3*100+digit2*10+digit1) begin
		//$display ("Numbers good");
		end else begin
		$display ("Numbers bad ");
		end
	end
	always
		begin
			clk <=1; #5;
			clk <=0; #5;
		end
endmodule
```

Και οι προσομοιώσεις ότι το Module δουλεύει όπως πρέπει, με reset και enable.

```
# i:        7014, Digits:  7 0 1 4          7014.
# i:        7015, Digits:  7 0 1 5          7015.
# i:        7016, Digits:  7 0 1 6          7016.
# i:        7017, Digits:  7 0 1 7          7017.
# i:        7018, Digits:  7 0 1 8          7018.
# i:        7019, Digits:  7 0 1 9          7019.
# i:        7020, Digits:  7 0 2 0          7020.
# i:        7021, Digits:  7 0 2 1          7021.
```



Άσκηση 3

Ζητείται η περιγραφή σε Verilog ενός συστήματος hamming (12, 5). Δηλαδή να κωδικοποιεί 12 – 5 = 7 bits πληροφορίας σε δώδεκα bit, προσθέτοντας 5 bit parity.

Συνοπτικά ο κώδικας Hamming δουλεύει ως εξής: Σε ένα μήνυμα προστίθεται επιπλέον bits πληροφορίας, έτσι ώστε να μπορεί να βρεθεί και να διορθωθεί ένα σφάλμα bit. Κάθε επιπλέον bit λέγεται parity bit. Επιπλέον, κάθε bit από το τελικό κωδικοποιημένο μήνυμα απαριθμείται με έναν αύξων αριθμό από το 1. Η δυαδική αναπαράσταση αυτών των αριθμών μπορεί να χρησιμοποιηθεί για να χωρίσει στα δύο το σύνολο των bit, με διαφορετικό τρόπο κάθε φορά. Για παράδειγμα το bit με αριθμό 0101 ανήκει στην ομάδα 0 με βάση το πρώτο bit, στην 1 με βάση το δεύτερο κ.ο.κ. Γίνεται κατανοητό ότι ένα ψηφίο μπορεί να προσδιοριστεί από την πληροφορία των ομάδων στην οποία ανήκει. Στο κώδικα hamming κάθε parity bit μας λέει αν μία από τις δύο ομάδες έχει λάθος. Αυτό γίνεται με το να ανατεθεί σε αυτό, ένα bit έτσι ώστε ο αριθμός των 1 να είναι άρτιος. Έτσι, ελέγχοντας των αριθμό των 1 bit για κάθε parity group, μπορούμε να προσδιορίσουμε τη θέση του σφάλματος (με δεδομένο αν αυτό είναι 1) και επομένως να το διορθώσουμε.

Η εξίσωση $k=2^r-r-1$ όπου r είναι ο αριθμός των parity bit του κώδικα hamming, μας δίνει τον ο αριθμό των bit δεδομένων k [7]. Για $r=4$ και $r=5$, έχουμε $k=11$ και $k=26$ αντίστοιχα. Γίνεται κατανοητό ότι το πέμπτο parity bit θα αντιστοιχηθεί σε μικρό parity group, καθώς θα μπορούσε να χρησιμοποιηθεί για να σχηματιστεί κώδικας hamming με 26 ψηφία data.

Ο πίνακας Hamming(12,5) που τελικά διαμορφώνεται είναι ο παρακάτω. Να σημειωθεί ότι στον κώδικα Hamming parity bits επίτηδες τοποθετούνται πρώτα ε κάθε parity group, έτσι η θέση τους αντιστοιχεί σε μία δύναμη του 2.

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Encoded	p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12
p1	X		X		X		X		X		X		X		X		X
p2		X	X			X	X			X	X			X	X		
p4				X	X	X	X					X	X	X	X		
p8								X	X	X	X	X	X	X	X		
p16																X	X

Με βάση τα παραπάνω σχηματίζονται τα ακόλουθα κυκλώματα στη verilog.

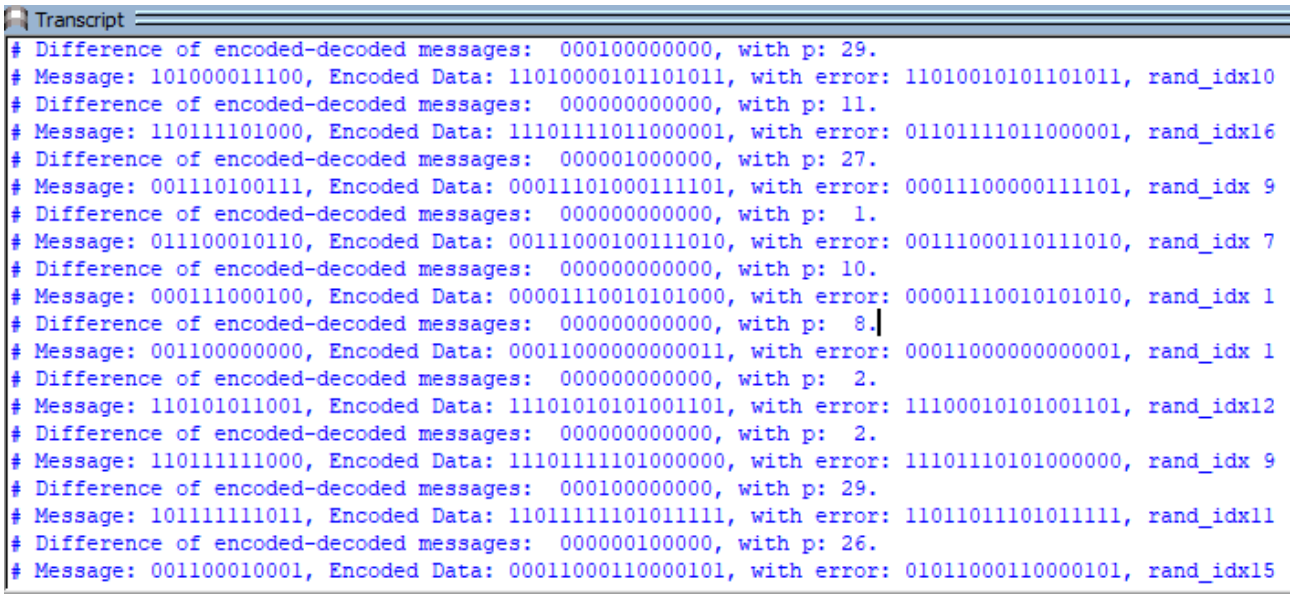
Για το κωδικοποιητή hamming:

```
// Hamming (12,5) data encoder
module ask3_hamming_encoder (output wire [16:0]enc_data,
                             input wire [11:0]data);

    wire [4:0]p;
    xor u1(p[0], data[0], data[1], data[3], data[4], data[6], data[8], data[10], data[11]);
    xor u2(p[1], data[0], data[2], data[3], data[5], data[6], data[9], data[10]);
    xor u3(p[2], data[1], data[2], data[3], data[7], data[8], data[9], data[10]);
    xor u4(p[3], data[4], data[5], data[6], data[7], data[8], data[9], data[10]);
    xor u5(p[4], data[11]);

    // buf b1(enc_message, {data[11], p[4], data[10:4], p[3], data[3:1], p[2], data[0], p[1], p[0] });
    assign enc_data[16:0] = {data[11], p[4], data[10:4], p[3], data[3:1], p[2], data[0], p[1], p[0] };


endmodule
```

Για το αποκωδικοποιητή hamming:

```
// Hamming (12,5) data decoder
module ask3_hamming_decoder (output reg [11:0]dec_data, wire [4:0]p,
                             input wire [16:0]enc_data);

    assign p[0] = (enc_data[0] ^ enc_data[2] ^ enc_data[4] ^ enc_data[6] ^ enc_data[8] ^ enc_data[10] ^ enc_data[12] ^ enc_data[14] ^ enc_data[16]);
    assign p[1] = (enc_data[1] ^ enc_data[2] ^ enc_data[5] ^ enc_data[6] ^ enc_data[9] ^ enc_data[10] ^ enc_data[13] ^ enc_data[14]);
    assign p[2] = (enc_data[3] ^ enc_data[4] ^ enc_data[5] ^ enc_data[6] ^ enc_data[11] ^ enc_data[12] ^ enc_data[13] ^ enc_data[14]);
    assign p[3] = (enc_data[7] ^ enc_data[8] ^ enc_data[9] ^ enc_data[10] ^ enc_data[11] ^ enc_data[12] ^ enc_data[13] ^ enc_data[14]);
    assign p[4] = (enc_data[16]);

    always @( enc_data[16:0] or p[4:0] )
    begin
        case ( p[4:0] )
            5'b0_0011: dec_data[11:0] <= { enc_data[16], enc_data[15:8], enc_data[6:4], !enc_data[2] }; // 3
            5'b0_0101: dec_data[11:0] <= { enc_data[16], enc_data[15:8], enc_data[6:5], !enc_data[4], enc_data[2] }; //5
            5'b0_0110: dec_data[11:0] <= { enc_data[16], enc_data[15:8], enc_data[6], !enc_data[5], enc_data[4], enc_data[2] }; // 6
            5'b0_0111: dec_data[11:0] <= { enc_data[16], enc_data[15:8], !enc_data[6], enc_data[5:4], enc_data[2] }; // 7
            5'b0_1001: dec_data[11:0] <= { enc_data[16], enc_data[15:9], !enc_data[8], enc_data[6:4], enc_data[2] }; // 9
            5'b0_1010: dec_data[11:0] <= { enc_data[16], enc_data[15:10], !enc_data[9], enc_data[8], enc_data[6:4], enc_data[2] }; // 10
            5'b0_1011: dec_data[11:0] <= { enc_data[16], enc_data[15:11], !enc_data[10], enc_data[9:8], enc_data[6:4], enc_data[2] }; // 11
            5'b0_1100: dec_data[11:0] <= { enc_data[16], enc_data[15:12], !enc_data[11], enc_data[10:8], enc_data[6:4], enc_data[2] }; // 12
            5'b0_1101: dec_data[11:0] <= { enc_data[16], enc_data[15:13], !enc_data[12], enc_data[11:8], enc_data[6:4], enc_data[2] }; // 13
            5'b0_1110: dec_data[11:0] <= { enc_data[16], enc_data[15:14], !enc_data[13], enc_data[12:8], enc_data[6:4], enc_data[2] }; // 14
            5'b0_1111: dec_data[11:0] <= { enc_data[16], enc_data[15], !enc_data[14], enc_data[13:8], enc_data[6:4], enc_data[2] }; // 15
            5'b1_0001: dec_data[11:0] <= { !enc_data[16], enc_data[15:8], enc_data[6:4], enc_data[2] }; // 17
            default: dec_data[11:0] <= { enc_data[16], enc_data[15:8], enc_data[6:4], enc_data[2] };
        endcase
    end

endmodule
```

Η προσομοίωσηδίνει:



Είναι προφανές ότι υπάρχει ένα λάθος γιατί μερικές φορές ο αποκωδικοποιητής επιστρέψει μήνυμα με ένα διαφορετικό bit, αν και το testbench φροντίζει να φλιπάρει κάθε φορά ένα τυχαίο bit.

References

[1]: *https://www.allaboutcircuits.com/textbook/digital/chpt-11/finite-state-machines/*

[2]: *https://pngtree.com/freepng/integrated-circuit-chip-icon-graphic-design-template-vector_3790112.html*

[3]: *https://tex.stackexchange.com/a/140581*

[4]: *https://tex.stackexchange.com/a/20786*

[5]: https://www.allaboutcircuits.com/technical-articles/encoding-the-states-of-a-finite-state-machine-vhdl/

[6]: https://www.allaboutcircuits.com/technical-articles/conversion-of-flip-flops-part-iv-d-flip-flops/

[7]: https://en.wikipedia.org/wiki/Hamming_code

[8] : https://www.allaboutcircuits.com/technical-articles/conversion-of-flip-flops-part-iv-d-flip-flops/