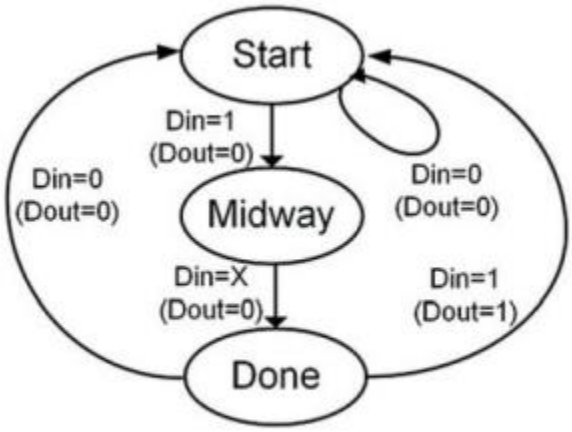
Αναφορά εργασίας

Άσκηση 1α:

Στην πρώτη άσκηση καλούμαστε να σχεδιάσουμε σε Verilog ένα μοντέλο συμπεριφοράς που υλοποιεί το FSM του σχήματος 1 και να το επαληθεύσουμε με τη χρήση κατάλληλου testbench.



Σχήμα 1. Διάγραμμα μεταβάσεων για την άσκηση 1.

Παρατηρούμε από το διάγραμμα μεταβάσεων ότι όταν βρεθεί το FSM στην κατάσταση **Done**, με είσοδο 0 η έξοδος γίνεται 0, ενώ με είσοδο 1, η έξοδος γίνεται 1 (κάτι που θα φανεί και στον πίνακα αλήθειας παρακάτω). Έτσι καταλαβαίνουμε ότι πρόκειται για FSM τύπου **Mealy.**

Κωδικοποίηση καταστάσεων:

State encoding	
State	Encoding $D_{1:0}$
S0(Start)	00
S1(Midway)	01
S2(Done)	10

Πίνακας αλήθειας:

Mealy state transition and output table			
Current State S	Input Din	Next State S'	Output Y
S0	0	S0	0
S0	1	S1	0
S1	X	S2	0
S2	0	S0	0
S2	1	S0	1

Λογικές εξισώσεις εξόδου FSM:

Με βάση τον πίνακα αλήθειας (και την κωδικοποίηση των καταστάσεων) η έξοδος του FSM περιγράφεται απο την εξίσωση **Dout = D_1*Din**

Κώδικας Verilog: (Ακολουθεί παράθεση του συνολικού κώδικα σε κομμάτια για την ταυτόχρονη επεξήγηση αυτού) (Αρχείο **PartA.v**)

Αρχικά ορίζουμε το **module** fsml_behavioral χρησιμοποιώντας τον ορισμό θυρών που ζητήθηκε από την εκφώνηση:

```
module fsml_behavioral (output reg Dout,
            input wire Clock, Reset, Din);
```

Ύστερα ακολουθούν οι αρχικοί ορισμοί των στοιχείων που διατηρούν την υπάρχουσα και την επόμενη κατάσταση ανά πάσα στιγμή (currentState, nextState), καθώς και η κωδικοποίηση των καταστάσεων (start, midway, done)

```
localparam[1:0] //2bit
    start = 0, //00
    midway = 1, //01
    done = 2; //10
    reg[1:0] currentState, nextState;
```

Μετά τους αρχικούς ορισμούς, ακολουθούν 3 procedural blocks:

Το πρώτο (**STATE_MEMORY**):

```
always @(posedge Clock or posedge Reset)
begin: STATE_MEMORY
    if(Reset) currentState <=start;
    else currentState<=nextState;
end
```

είναι υπεύθυνο για τη διατήρηση της κατάστασης της μηχανής σύμφωνα με τις μεταβάσεις του σήματος ρολογιού (Clock) και επαναφοράς (Reset). Όσο το Reset=1, τότε η μηχανή θα μένει στην κατάσταση Start, ενώ όταν το Reset γίνει 0, τότε θα μπορέσει να μεταβεί σε επόμενη κατάσταση η οποία προσδιορίζεται από το 2ο block.

Το δεύτερο procedural block (**NEXT_STATE_LOGIC**):

```
always @(currentState or Din)//Block whe
begin: NEXT_STATE_LOGIC
    case(currentState)
        start: if(Din) nextState=midway;
               else nextState = start;
        midway: nextState=done;
        done: nextState=start;
    default: nextState=start;
    endcase
end
```

Σε αυτό το block, όπως φαίνεται, αναλόγως της κατάστασης που βρισκόμαστε ανά πάσα στιγμή, συναρτήσει της εισόδου Din, προσδιορίζεται η κατάλληλη μετάβαση κατάστασης που υπακούει στο διάγραμμα μεταβάσεων που δίνεται στην εκφώνηση.

Έτσι όταν το FSM βρίσκεται στην κατάσταση start, αν η είσοδος είναι 1, τότε μεταβαίνει η μηχανή μας στην κατάσταση midway, αλλιώς θα παραμείνει στην κατάσταση start, όταν βρίσκεται στην κατάσταση midway, μεταβαίνει στην κατάσταση done (είτε έχουμε σαν είσοδο 1 είτε 0), και αντίστοιχα από την κατάσταση done, θα μεταβεί στην κατάσταση start. (Για τη διάκριση των περιπτώσεων αυτών, χρησιμοποιήθηκε η εντολή case).

Το τρίτο block έχει να κάνει με τον προσδιορισμό της εξόδου του FSM ανάλογα της κατάστασης που βρίσκεται αυτό κάθε στιγμή καθώς και συναρτήσει της εισόδου (αφού πρόκειται για μηχανή τύπου Mealy)

```
always@(currentState or Din)//Block
        begin: OUTPUT_LOGIC
        case (currentState)
                done:if(Din==1'b1)
                        Dout=1'b1;
                     else
                        Dout=1'b0;
        default: Dout=1'b0;
        endcase
end
```

Βλέπουμε λοιπόν ότι όταν βρεθεί η μηχανή στην κατάσταση done και η είσοδος είναι Din = 1, τότε η έξοδος θα πάρει την τιμή 1. Στις υπόλοιπες περιπτώσεις η έξοδος θα είναι 0. (Κάτι τέτοιο φαίνεται και στον πίνακα αλήθειας που παρουσιάστηκε αρχικά.

Κώδικας testbench (αρχείο **AtestBench.v**)

Στη συνέχεια δημιουργήθηκε ένα test bench για να ελέγξουμε αν οι έξοδοι του DUT είναι οι αναμενόμενες για όλους τους δυνατούς συνδυασμούς εισόδων (που γράφτηκαν σε ένα αρχείο **tbValues.txt** που θα παρατεθεί)

Το περιεχόμενο του αρχείου tbValues.txt είναι αυτό:

```
10
00
01
01
01
00
01
00
00
00
01
00
01
00
01
00
```

(Πάνω πάνω βλεπουμε το διάνυσμα 10, που δηλώνει Reset=1 και in=0)

Όπως φαίνεται από τα διανύσματα του αρχείου, ξεκινάμε από την κατάσταση όπου το Reset = 1, (άρα το FSM όπως εξηγήθηκε θα παραμείνει στην κατάσταση start μέχρι Reset -> 0).

Ο κώδικας του testbench λοιπόν είναι ο εξής:

```
`timescale 1ns/1ns
module ATB ();
    reg clk;
    reg reset;
    reg in;

    wire out;

    fsm1_behavioral dut(.Dout(out), .Clock(clk), .Reset(reset), .Din(in));
    reg [1:0] testVector[17:0];
```

Αρχικά ορίζουμε το **timescale** σε 1ns/1ns, τις εισόδους **clk**, **reset**, **in** (ως reg) που θα οδηγήσουν τις αντίστοιχες εισόδους του DUT, καθώς και την έξοδο **out** ως wire που θα οδηγηθεί απο την έξοδο του DUT **(Dout)**. Ύστερα δηλώνουμε το DUT στο testbench και ορίζουμε έναν πίνακα τύπου reg 2bit, 18 θέσεων όπου θα μπουν τα vectors του tbValues.txt που αναφέρθηκαν πριν.

Στη συνέχεια χρησιμοποιείται ένα initial block **(1)** κατά το οποίο αρχικοποιείται το reset και το in σύμφωνα με το πρώτο διάνυσμα του tbValues.txt (reset = 1, in = 0) και με χρήση καθυστερήσεων παρέχονται οι τιμές εισόδου και reset στο DUT (ουσιαστικά ανά κύκλο ρολογιού παρέχεται νέα τιμή εισόδου ώστε να είναι πιο εύκολη η επαλήθευση, καθώς κάθε διάνυσμα οδηγεί σε μία μόνο μετάβαση κατάστασης. Στην περίπτωση όπου βάζαμε μεγαλύτερες καθυστερήσεις, η είσοδος που θα παρέχονταν απο ένα διάνυσμα θα έμενε ίδια για πάνω από εναν κύκλο ρολογιού, οπότε θα ευθυνόταν για περισσότερες μεταβάσεις -ή μη- καταστάσεων)

```
initial
 begin
    $readmemb("tbValues.txt", testVector);

    {reset, in} = testVector[0];
    #25{reset, in} = testVector[1];
    #5{reset, in} = testVector[2];
    #10{reset, in} = testVector[3];
    #10{reset, in} = testVector[4];
    #10{reset, in} = testVector[5];
    #10{reset, in} = testVector[6];
    #10{reset, in} = testVector[7];
    #10{reset, in} = testVector[8];
    #10{reset, in} = testVector[9];
    #10{reset, in} = testVector[10];
    #10{reset, in} = testVector[11];
    #10{reset, in} = testVector[12];
    #2{reset, in} = testVector[13];
    #2{reset, in} = testVector[14];
    #2{reset, in} = testVector[15];
    #2{reset, in} = testVector[16];
    #2{reset, in} = testVector[17];
```

(1)

```
end
```

Ακολουθεί ένα άλλο initial block στο οποίο αρχικοποιείται το clk -> 1:

```
initial
    begin
            clk=1'b1;
    end
```

Και τέλος, με τη βοήθεια ενός always block, εναλλάσσουμε την τιμή του clk μεταξύ των τιμών 0 και 1 με ενδιάμεση καθυστέρηση 5ns. Έτσι δημιουργείται ένα ρολόι με περίοδο 10ns.

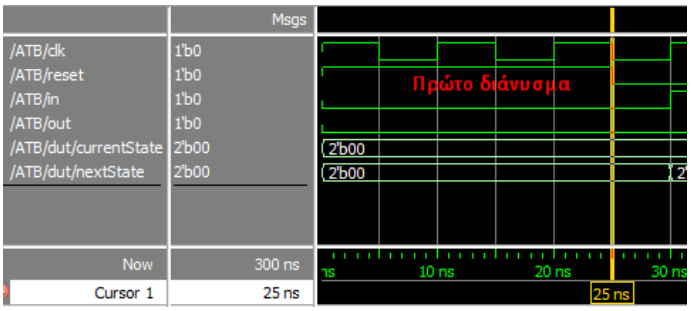
```
always
    begin
        #5 clk= ~clk;
    end
```

Αιτιολόγηση σωστής λειτουργίας του FSM (Ακολουθεί με λόγια η περιγραφή της αναμενόμενης συμπεριφοράς και αμέσως μετά screenshot της προσομοίωσης)

(Υπενθύμιση του πίνακα κωδικοποίησης καταστάσεων **start->00**, **midway->01**, **done->10**)

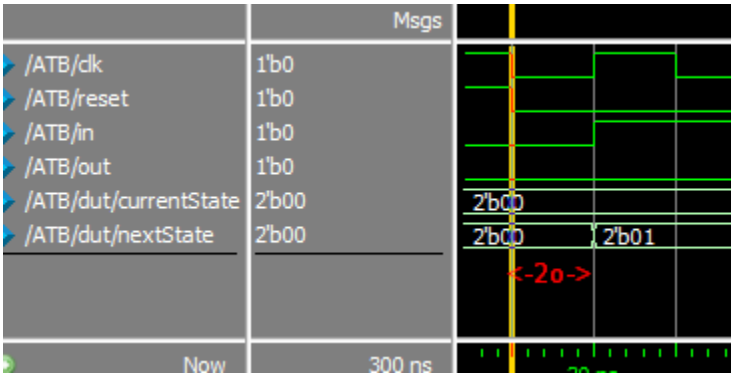
Βλέποντας τις τιμές που έχουν γραφτεί στο αρχείο tbValues, σε συνάρτηση με το διάγραμμα μεταβάσεων περίμενουμε την εξής συμπεριφορά:

Το πρώτο διάνυσμα (10) κρατάει το reset στο 1, οπότε όπως αναφέρθηκε, το FSM θα παραμένει στην κατάσταση start.



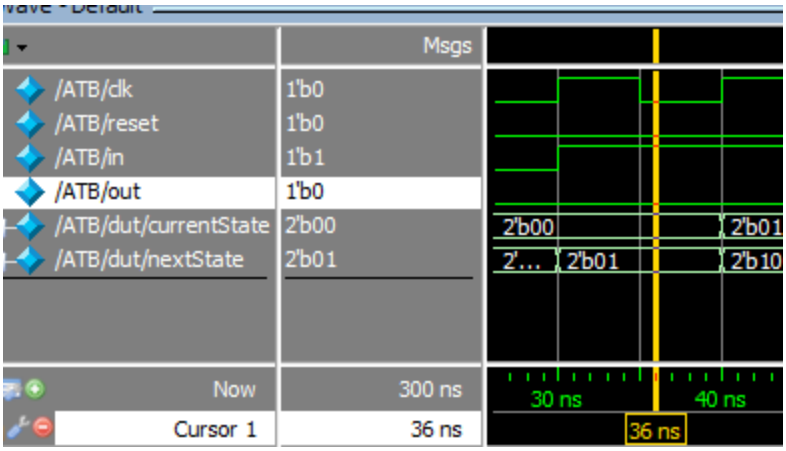
Και όντως βλέπουμε ότι το reset =1 μέχρι τα 25ns (όπου θα έρθει η τιμή του 2^{ου} διανύσματος), και μέχρι τότε το currentState και το nextState έχουν τιμή 2'b00.

Το δεύτερο διάνυσμα (00 -> `#25{reset, in} = testVector[1];`) βάζει στο reset την τιμή 0 και στο in την τιμή 0, κάτι που σύμφωνα με το διάγραμμα μεταβάσεων, θα οδηγήσει στη διατήρηση του FSM στην κατάσταση start (αν currentState->start και Din->0, nextState->start).



Βλέπουμε εδώ ότι στα 25ns το reset = 0 και επειδή το in = 0 το nextState παραμένει 2'b00, όπως ορίζει ο πίνακας μεταβάσεων.

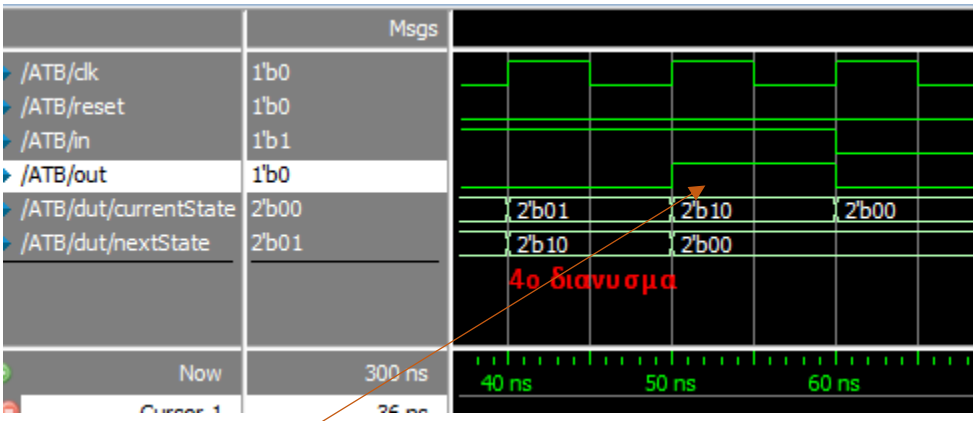
Το τρίτο διάνυσμα (01 `#5{reset, in} = testVector[2];`) θα δώσει στην είσοδο την τιμή 1, και αφού το reset=0, το FSM θα μεταβεί στην κατάσταση midway, στον αμέσως επόμενο κύκλο ρολογιού.



Φαινεται στις κυμματομορφές ότι στα 30ns το in=1 -> κάτι που πυροδοτεί το NEXT_STATE_LOGIC που αναφέρθηκε στην επεξήγηση του κώδικα περιγραφής του FSM και δίνεται η τιμή 2'b01 (δηλαδή η κατάσταση midway) στο nextState, και με την ερχόμενη ακμή ρολογιού (40ns) το currentState παίρνει αυτή την τιμή.

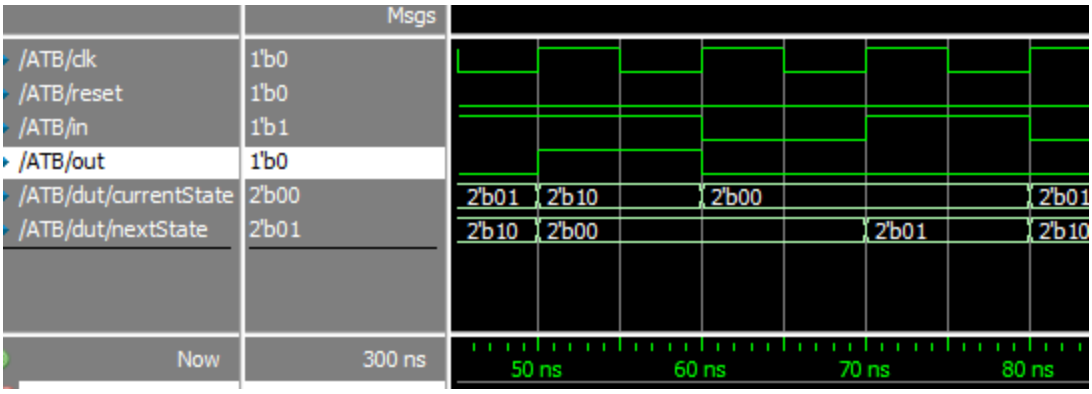
Μετά θα ακολουθήσει το 4ο διάνυσμα (01) (40ns) όπου θα οριστεί αντίστοιχα ως nextState -> 10 (done)

Οπότε αναμένουμε στα 50ns (που θα έρθει το επόμενο posedge του ρολογιού) το currentState να πάρει την τιμή του nextState και έτσι το FSM να βρεθεί στην κατάσταση done.

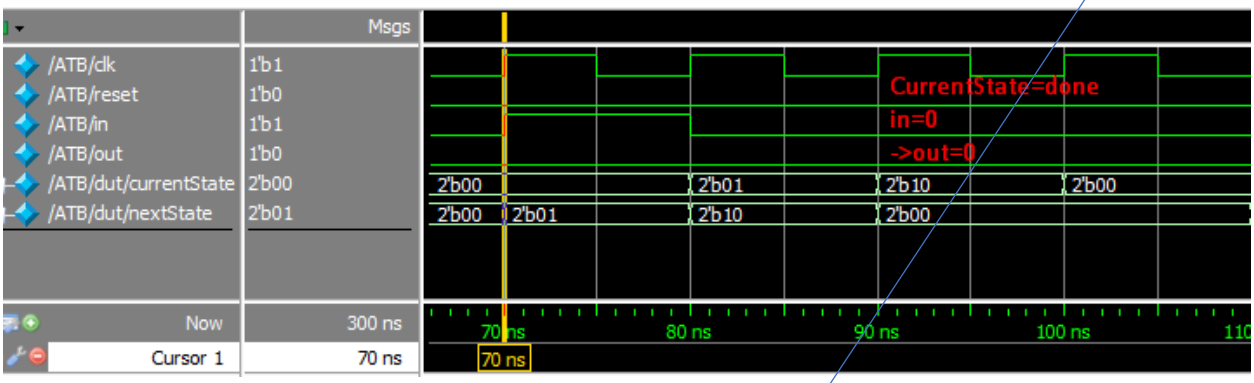


Στα 50ns έρχεται και το επόμενο διάνυσμα (01) και διαπιστώνουμε (όπως αναμενόταν) ότι η έξοδος του FSM έγινε 1 (και το nextState -> 2'b00)

Στη συνέχεια (στα 60ns όπου βρισκόμαστε πλέον πάλι στο start -> 00 – και η έξοδος ξαναγίνεται 0 αφού όπως είπαμε για έξοδο 1 θέλουμε currentState = done και Din = 1) δίνουμε το διάνυσμα 00, για να δούμε ότι το FSM θα αποκτήσει nextState->00 και στα 70ns (όπου έρχεται το επόμενο posedge του clk) θα παραμείνει στην κατάσταση (currentState) start.



Ύστερα για να παρουσιαστεί και η περίπτωση όπου μεταβαίνουμε στην κατάσταση done (απο την midway) με Din/in -> 0 αλλά και για να δούμε ότι η έξοδος παραμένει μηδέν σε περίπτωση που το FSM βρεθεί στην κατάσταση done αλλά η είσοδος είναι 0 επιλέχθηκαν τα διανύσματα 01, 00, 00, 00 και βλέπουμε την αντίστοιχη συμπεριφορά στην προσομοίωση:



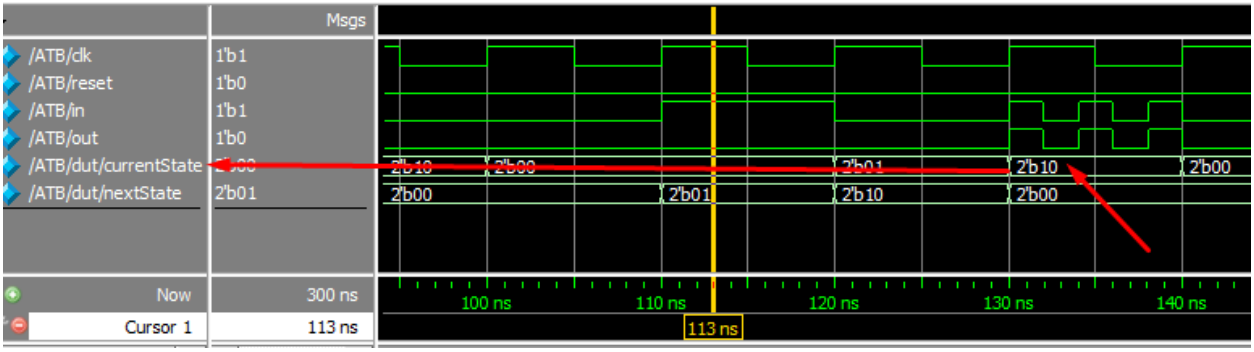
Σε αυτή την εικόνα βλέπουμε, ο συνδιασμός των διανυσματων που επιλεχθηκαν μας εφεραν στα 100ns στην κατάσταση start (οπως αναμενόταν) και επειδη το in είναι μηδέν, το nextState θα μείνει 2'b00 (start). Στο 110 όμως θα έρθει το διάνυσμα 01 (`#10{reset, in} = testVector[10];`)

άρα αναμένουμε στο 120ns να πάμε σε currentState=2'b01, ακολουθεί το διάνυσμα (στο 120ns) 00 που πρέπει να μας πάει στα 130ns στην κατάσταση currentState=done (2'b10). Στα 130ns θα ακολουθήσουν τα διανύσματα που φαίνονται στην εικόνα:

```
#10{reset, in} = testVector[12];
#2{reset, in} = testVector[13];
#2{reset, in} = testVector[14];
#2{reset, in} = testVector[15];
#2{reset, in} = testVector[16];
#2{reset, in} = testVector[17];
```

Το διάνυσμα που βρίσκεται στο testVector[12] θα έρθει στα 130ns, μετά απο 2ns θα έρθει το testVector[13] (που αμα δούμε την αντιστοιχία στο tbValues.txt είναι το 00) και θα ακολουθήσουν μερικές ακόμα εναλλαγές του in (με το reset να μένει στο 0) με καθυστέρηση 2ns η μία απο την άλλη.

Αυτό έγινε για να διαπιστώσουμε ότι το FSM μεταξύ 2 θετικών ακμών ρολογιού (από θετική ακμή θετική ακμή περνάνε 10ns), ώντας στην κατάσταση done (130ns έως 140ns), παίρνει στην έξοδό του τιμή που αναλογεί στο εκάστοτε in, σύμφωνα με τον πίνακα μεταβάσεων που δίνεται στην εκφώνηση. Δείχνεται δηλαδή ότι η έξοδος του FSM εξαρτάται από την είσοδο (όπως και πρέπει).



Άσκηση 1β:

Στο β) ερώτημα καλούμαστε να κάνουμε ό,τι κάναμε στο ερώτημα α), με τη διαφορά ότι θα χρησιμοποιηθεί one-hot κωδικοποίηση καταστάσεων, έτσι έχουμε:

Κωδικοποίηση καταστάσεων:

State encoding One-hot	
State	Encoding D2:0
S0(Start)	001
S1(Midway)	010
S2(Done)	100

Πίνακας αλήθειας:

Mealy state transition and output table			
Current State S	Input Din	Next State S'	Output Y
S0	0	S0	0
S0	1	S1	0
S1	X	S2	0
S2	0	S0	0
S2	1	S0	1

Λογικές εξισώσεις εξόδου FSM:

Dout = D_2*Din

Κώδικας Verilog: (Αρχείο **PartB.v**)

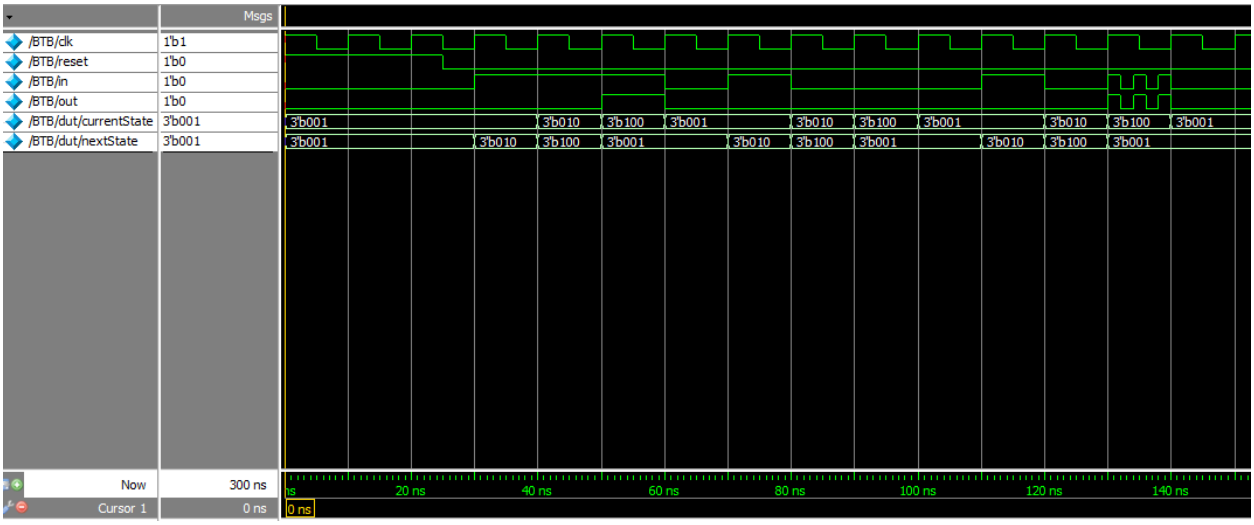
Ουσιαστικά, σε σχέση με τον κώδικα του πρώτου ερωτήματος η διαφορά φαίνεται εδώ:

```
localparam[2:0] //3bit
    start = 001, //001
    midway = 010, //010
    done = 100; //100
    reg[2:0] currentState, nextState;
```

```
module fsml_behavioralB (output reg Dout,
            input wire Clock, Reset, Din);
localparam[2:0] //3bit
    start = 001, //001
    midway = 010, //010
    done = 100; //100
    reg[2:0] currentState, nextState;

always @(posedge Clock or posedge Reset)//
begin: STATE_MEMORY
    if(Reset) currentState <=start;
    else currentState<=nextState;
end

always @(currentState or Din)//Block where
begin: NEXT_STATE_LOGIC
    case(currentState)
        start: if(Din) nextState=midway;
               else nextState = start;
        midway: nextState=done;
        done: nextState=start;
    default: nextState=start;
    endcase
end

always@(currentState or Din)//Block where
        begin: OUTPUT_LOGIC
        case (currentState)
                done:if(Din==1'b1)
                        Dout=1'b1;
                     else
                        Dout=1'b0;
        default: Dout=1'b0;
        endcase
end

endmodule
```

Κώδικας testbench: (αρχείο **BtestBench.v**)

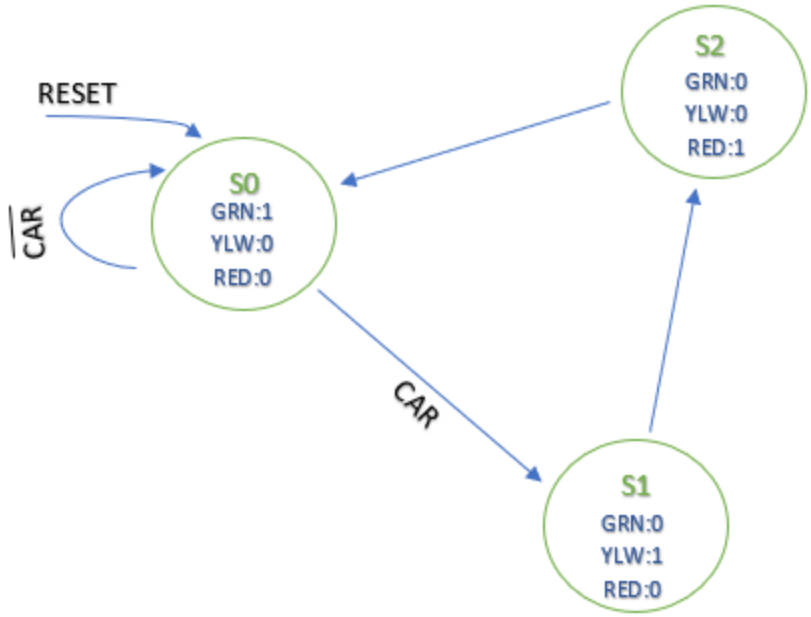
Μιας και περιγράφουμε ουσιαστικά το ίδιο FSM, χρησιμοποιήθηκε το ίδιο testbench με το πρώτο ερώτημα και τα αποτελέσματα της προσομοίωσης φαίνονται στο ακόλουθο screenshot:



Άσκηση 2:

Στη δεύτερη άσκηση καλούμαστε να δημιουργήσουμε με τη χρήση Verilog ένα μοντέλο συμπεριφοράς ενός FSM για ένα ελεγκτή φαναριών και συγκεκριμένα να σχεδιάσουμε τα σήματα ελέγχου μόνο για το κόκκινο, πορτοκαλί και πράσινο ενός δρόμου ταχείας κυκλοφορίας.

Ο κάθετος (στον ταχείας κυκλοφορίας) δρόμος διαθέτει έναν αισθητήρα, με τη βοήθεια του οποίου, βλέπουμε πότε βρίσκεται αυτοκίνητο σε αυτόν. Όσο δε βρίσκεται αυτοκίνητο σε αυτόν, το πράσινο πρέπει να είναι μονίμως αναμένο, ενώ όταν έρθει αυτοκίνητο, αυτός ο αισθητήρας θα θέσει ένα σήμα CAR και τότε το πράσινο στον ταχείας κυκλοφορίας θα σβήνει και θα ανάβει το πορτοκαλί, το οποίο μετά από λίγο θα σβήνει και θα ανάβει το κόκκινο. Στο κόκκινο θα μένει για 15 δευτερόλεπτα και μετά θα γίνεται πάλι πράσινο. Το **διάγραμμα μεταβάσεων των καταστάσεων** που προκύπτει λοιπόν βάσει αυτής της περιγραφής είναι το ακόλουθο:



Πίνακας κωδικοποίησης καταστάσεων:

State encoding	
State	Encoding $S_{1:0}$
S0 (green)	00
S1(yellow)	01
S2(red)	10

Πίνακας αλήθειας:

Output Table	
Current State	Outputs $L_{2:1:0}$
S0 (00)	001
S1 (01)	010
S2 (10)	100

Εξισώσεις εξόδου:

$L_0 = S_{1bar} * S_{0bar}$
$L_1 = S_0$
$L_2 = S_1$

Κώδικας Verilog: (Ακολουθεί παράθεση του συνολικού κώδικα σε κομμάτια για την ταυτόχρονη επεξήγηση αυτού) (Αρχείο **Circuit.v**)

Αρχικά ορίζουμε το **module** TrafficLights χρησιμοποιώντας τον ορισμό θυρών που ζητήθηκε από την εκφώνηση:

```
module TrafficLights (output reg GRN,YLW,RED,
                      input wire Clock,Reset,CAR);
```

Ύστερα ακολουθούν οι αρχικοί ορισμοί των στοιχείων που διατηρούν την υπάρχουσα και την επόμενη κατάσταση ανά πάσα στιγμή (currentState, nextState), καθώς και η κωδικοποίηση των καταστάσεων (GREEN,YELLOW,REDD):

```
reg [1:0] currentState,nextState;

localparam [1:0]//state encoding
        GREEN=2'b00,
        YELLOW=2'b01,
        REDD=2'b10;
```

Επίσης δηλώνονται τα σήματα SEC15, SEC3, count, TIMEOUT, TIMEOUTYLW που θα βοηθήσουν στο timing του φαναριού:

```
localparam [31:0] SEC15=32'd15000;//to count 15 seconds
localparam [31:0] SEC3=32'd3000;//to count 3 seconds
reg [31:0] count;
reg TIMEOUT,TIMEOUTYLW;
```

Μετά τους αρχικούς ορισμούς, ακολουθούν 3 procedural blocks:

Το πρώτο (**STATE_MEMORY**):

```
always@(posedge Clock or posedge Reset)
        begin: STATE_MEMORY
                if(Reset) begin currentState<=GREEN;
                                TIMEOUT<=0;
                                TIMEOUTYLW<=0;
                                count<=SEC3;//needs to be reset when we leave RED state
                          end
                        else currentState<=nextState;
        end
```

Είναι υπεύθυνο για τη διατήρηση της κατάστασης της μηχανής σύμφωνα με τις μεταβάσεις του σήματος ρολογιού (Clock) και επαναφοράς (Reset). Όσο το Reset=1, τότε η μηχανή θα μένει στην κατάσταση GREEN,τα TIMEOUT, TIMEOUTYLW σήματα θα παραμείνουν στο 0 και το σήμα count θα διατηρεί την τιμή που του δίνεται από το σήμα SEC3, ενώ όταν το Reset γίνει 0, τότε θα μπορέσει να μεταβεί σε επόμενη κατάσταση η οποία προσδιορίζεται από το 2^{ο} block.

Το δεύτερο procedural block (**NEXT_STATE_LOGIC**):

```
always@(currentState or CAR or posedge TIMEOUT or posedge TIMEOUTYLW)/
        begin: NEXT_STATE_LOGIC
                case(currentState)
                        GREEN: if(CAR) nextState=YELLOW;//if GREEN and
                                else nextState=GREEN;//if GREEN and th
                        YELLOW: if(TIMEOUTYLW)begin nextState=REDD;//W

                                                end
                                else nextState=YELLOW;//if !TIMEOUTYLW
                        REDD:if(TIMEOUT)begin nextState=GREEN;//When R

                                                end
                                else nextState=REDD;
                default: nextState=GREEN;
                endcase
        end
```

Σε αυτό το block, όπως φαίνεται, αναλόγως της κατάστασης που βρισκόμαστε ανά πάσα στιγμή, συναρτήσει της εισόδου CAR, προσδιορίζεται η κατάλληλη μετάβαση κατάστασης που υπακούει στο διάγραμμα μεταβάσεων που παρουσιάστηκε στην αρχή.

Έτσι όταν το FSM βρίσκεται στην κατάσταση **GREEN**, αν το CAR = 1 (υπάρχει αυτοκίνητο στον κάθετο δρόμο), τότε μεταβαίνει η μηχανή μας στην κατάσταση **YELLOW**, αλλιώς θα παραμείνει στην κατάσταση GREEN, όταν βρίσκεται στην κατάσταση **YELLOW**, μεταβαίνει (μετά από κάποιον χρόνο παραμονής στην κατάσταση YELLOW, το πέρας του οποίου προσδιορίζει το σήμα TIMEOUTYLW και η ακριβής λειτουργία του θα παρουσιαστεί στη συνέχεια) στην κατάσταση **REDD**, και αντίστοιχα από την κατάσταση **REDD**, θα μεταβεί (πάλι μετά το πέρας κάποιου χρόνου που θα το γνωστοποιήσει το σήμα TIMEOUT) στην κατάσταση **GREEN**. Άρα με απλά λόγια το FSM μεταβαίνει από την κατάσταση GREEN στην YELLOW όταν ο αισθητήρας δει ότι υπάρχει αυτοκίνητο στον κάθετο δρόμο, και μετά ανεξαρτήτως του τι δείχνει ο αισθητήρας θα ακολουθήσει η κατάσταση REDD και μετά πάλι η κατάσταση GREEN.

Το τρίτο block έχει να κάνει με τον προσδιορισμό της εξόδου του FSM (**OUTPUT_LOGIC**) ανάλογα της κατάστασης που βρίσκεται αυτό κάθε στιγμή (αφού πλέον έχουμε να κάνουμε με μηχανή Moore):

```
always@(currentState)
        begin: OUTPUT_LOGIC//depending on the state the FSM is
        case(currentState)
                GREEN:  begin{GRN,YLW,RED}=3'b100;
                        TIMEOUT=0; //reset TIMEOUT to 0
                        end
                YELLOW: begin{GRN,YLW,RED}=3'b010;
                        count=SEC3;//count=3seconds -> it serve
                        end
                REDD: begin{GRN,YLW,RED}=3'b001;
                         count=SEC15;//count=15seconds
                         TIMEOUTYLW=0; //reset TIMEOUTYLW to 0
                         end
        default:{GRN,YLW,RED}=3'b100;
        endcase
        end
```

Βλέπουμε λοιπόν ότι όταν το FSM βρίσκεται στην κατάσταση GREEN, η έξοδος GRN (που ελέγχει το πράσινο φανάρι) γίνεται 1, η έξοδος YLW (που ελέγχει το πορτοκαλί) γίνεται μηδέν και η έξοδος RED (που ελέγχει το αν είναι αναμένο ή όχι το κόκκινο) γίνεται επίσης 0.

Αντίστοιχα, στην κατάσταση YELLOW, μόνο η έξοδος YLW παίρνει την τιμή 1 (για να ανάψει το πορτοκαλί μόνο φανάρι)

και στην κατάσταση REDD, γίνεται 1 μόνο η έξοδος RED. Βλέπουμε σε αυτό το block επίσης ότι στην κατάσταση GREEN μηδενίζεται το σήμα TIMEOUT και στην κατάσταση REDD μηδενίζεται το σήμα TIMEOUTYLW.

Αναφέρθηκε πριν ότι τη μετάβαση από την κατάσταση YELLOW στην κατάσταση REDD την επιτρέπει το σήμα **TIMEOUTYLW**. Αυτό, όπως θα φανεί στην παρουσίαση του τελευταίου always block, γίνεται 1 αφότου παραμείνουμε στην κατάσταση YELLOW για 3 δευτερόλεπτα. Έτσι αφότου «κάνει τη δουλειά του» και μας επιτρέψει τη μετάβαση στην κατάσταση REDD, όταν το FSM πάει στην κατάσταση REDD, το TIMEOUTYLW μηδενίζεται, ώστε να είναι έτοιμο να προκαλέσει στον επόμενη YELLOW to REDD μετάβαση το delay που χρειαζόμαστε. Αντίστοιχα δουλεύει και ο μηδενισμός του **TIMEOUT** όταν βρεθεί το FSM στην κατάσταση GREEN (REDD to GREEN transition χρειάζεται 15 δευτερόλεπτα delay).

Παράλληλα βλέπουμε ότι το σήμα **count** στην κατάσταση YELLOW παίρνει την τιμή που δίνεται από το SEC3 (πρακτικά έτσι το count θα μετρήσει 3 δευτερόλεπτα προτού τεθεί το σήμα TIMEOUTYLW από 0 σε 1) και αντίστοιχα όταν βρεθεί το FSM στην κατάσταση REDD, το count θα πάρει την κατάλληλη τιμή ώστε να μετρηθούν 15 δευτερόλεπτα προτού το σήμα TIMEOUT γίνει απο 0, 1 και επιτραπεί η μετάβαση από την REDD στην GREEN.

Το τελευταίο block (**COUNTER)** αναλαμβάνει να κάνει ουσιαστικά τις μετρήσεις για 3 δευτερόλεπτα όταν το FSM πρόκειται να πάει απο την κατάσταση YELLOW στην κατάσταση REDD και για 15 δευτερόλεπτα όταν το FSM πρόκειται να πάει από την κατάσταση REDD στην κατάσταση GREEN.

```
always@(posedge Clock)//Timer block
        begin: COUNTER
                case(currentState)
                        REDD:begin//if currentState=REDD, coun
                                count=count-1;//count is decre
                                if(count==0)TIMEOUT=1'b1;//TIM
                             end
                        YELLOW:begin
                                count=count-1;
                                if(count==0)TIMEOUTYLW=1'b1;//
                             end
                endcase
        end

endmodule
```

Αυτό το block λειτουργεί σε κάθε θετική ακμή του Clock και (όπως φαίνεται) στην περίπτωση που το FSM βρίσκεται στην currentState = REDD ή currentState = YELLOW, ανά posedge του Clock μειώνεται κατά ένα η τιμή του count. Είδαμε στο προηγούμενο block ότι το count στην περίπτωση της κατάστασης REDD αρχικοποιείται με τιμή κατάλληλη ώστε να μετρηθούν 15sec ενώ στην περίπτωση που το FSM βρίσκεται στην κατάσταση YELLOW αρχικοποιείται έτσι ώστε να μετρήσει 3sec. Όταν λοιπόν το count γίνει μηδέν (ενώ το FSM βρίσκεται στη REDD) τίθεται το σήμα TIMEOUT σε 1, και επιτρέπεται όπως δείχθηκε προηγουμένως η μετάβαση από την κατάσταση REDD στην κατάσταση GREEN. Αντίστοιχα για τη μετάβαση απο τη YELLOW στη REDD, τίθεται το σήμα TIMEOUTYLW σε 1.

Κώδικας testbench (αρχείο **TestBench.v**)

Στη συνέχεια δημιουργήθηκε ένα test bench για να ελέγξουμε αν οι έξοδοι του DUT είναι οι αναμενόμενες ανά πάσα στιγμή λειτουργίας.

Ο κώδικας λοιπόν του testbench ειναι ο εξής:

```
`timescale 100us/10us
module ATB ();
    reg clk;
    reg reset;
    reg in;

    wire g,y,r;

    TrafficLights dut(g, y, r, clk,reset,in);
```

Αρχικά ορίζεται το **timescale** σε 100us/10us, τις εισόδους **clk,reset,in** (ως reg) που θα οδηγήσουν τις αντίστοιχες εισόδους του DUT, καθώς και τις εξόδους **g,y,r** ως wire που θα οδηγηθούν από τις ανάλογες εξόδους του DUT (**GRN,YLW,RED** αντίστοιχα). Ύστερα δηλώνεται το DUT στο testbench.

Στη συνέχεια χρησιμοποιείται ένα initial block κατά το οποίο αρχικοποιείται το reset σε 1 και το in σε 0:

```
initial
        begin
                clk=1'b1;
                in=0;
                reset=1;
        end
```

Να σημειωθεί ότι το reset είναι active high, άρα όταν reset = 1, γίνεται reset.

Ύστερα έχουμε ένα initial block που με χρήση κατάλληλων καθυστερήσεων παρέχονται οι τιμές εισόδου στο DUT:

```
initial
        begin
                #5000 in=1;//0.5sec | 5000*100us(timescale)= 500.000us=500ms=0.5sec
                #2000 in=0;//0.7sec
                #8000 in=1;//1.5sec
                #10000 in=0;//2.5sec
                #180000 in=1;//20.5sec
                #10000 in=0;//21.5sec


        end
```

Πρακτικά αφού το timescale έχει οριστεί σε 100us, η είσοδος θα γίνει από 0, 1 στα 0,5sec κάτι που υπολογίζεται στα σχόλια της πάνω φωτογραφίας.

Ακολουθεί ένα άλλο initial block που θέτει ουσιαστικά το FSM σε κανονική λειτουργία θέτοντας το reset σε 0:

```
initial
        begin
                #10000 reset=0;//1sec at reset state

        end
```

Όπως βλέπουμε, το reset γίνεται 0 με καθυστέρηση 10000=>1sec.

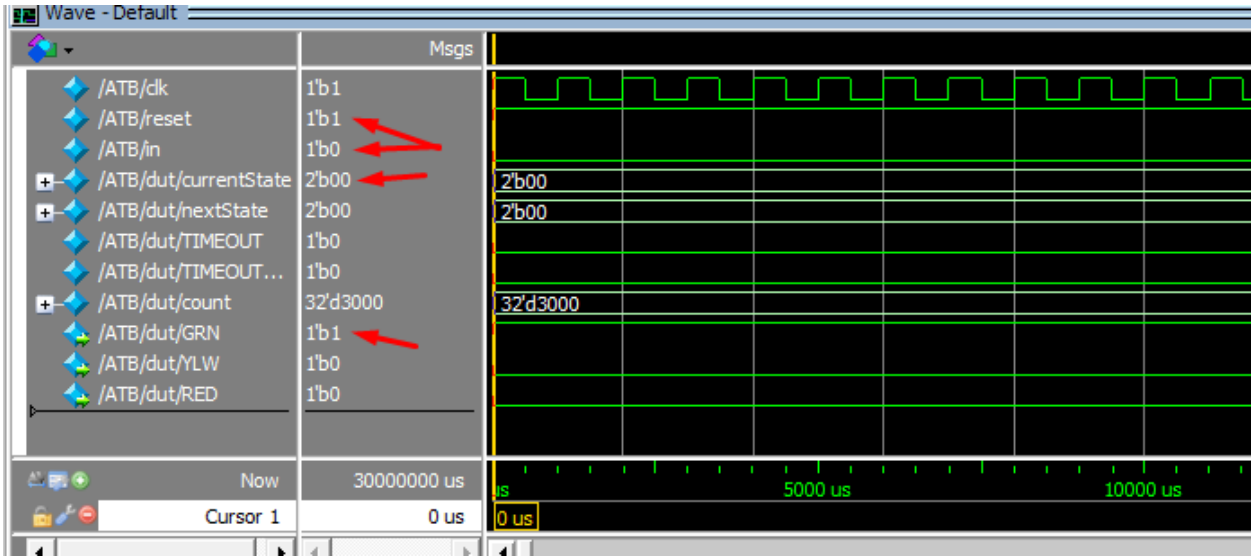
Τέλος έχουμε ένα always block, στο οποίο εναλλάσουμε την τιμή του clk μεταξύ των τιμών 0 και 1 με ενδιάμεση καθυστέρηση #5:

```
always
    begin
        #5 clk= ~clk;//5*100us=500us    *2->1ms = T clock
    end
endmodule
```

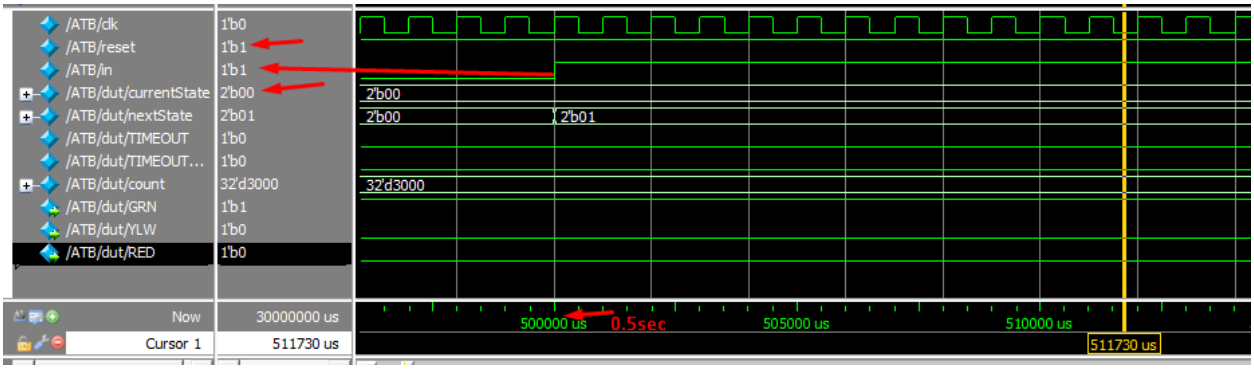
Οπότε πρακτικά δημιουργούμε ρολόι με περίοδο T=1ms (όπως φαίνεται και στο σχόλιο).

Οι καθυστερήσεις επιλέχθηκαν έτσι ώστε να δούμε τα εξής:

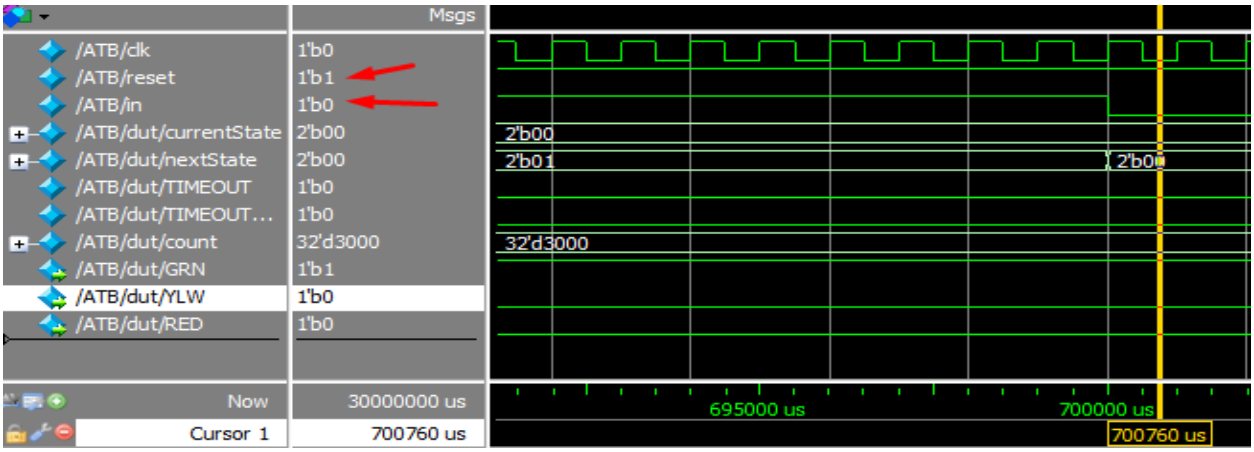
Αρχικά το reset θα μείνει για 1sec στην τιμή 1. Μέσα σε αυτό το πρώτο δευτερόλεπτο της προσομοίωσης θέτουμε την είσοδο ίση με 1 (στα 0,5sec) και την γυρνάμε πάλι στο 0 στα 0,7sec. Από αυτό περιμένουμε να δούμε ότι στην κατάσταση reset, με είσοδο 1, το FSM δε θα μεταβεί καταστάσεις.



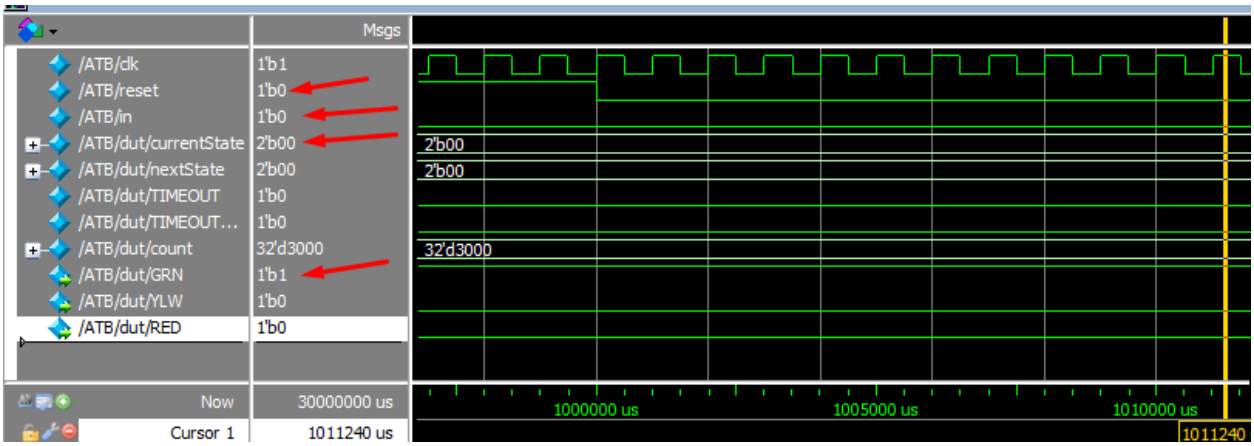
Στο παραπάνω screenshot βλέπουμε τα 10 πρώτα ms της λειτουργίας του. Περιμένουμε αλλαγή στα 500.000us->0.5sec όπου θα δούμε ότι το in θα γίνει 1 και το currentState θα μείνει στην κατάσταση 2'b00->GREEN->S0:



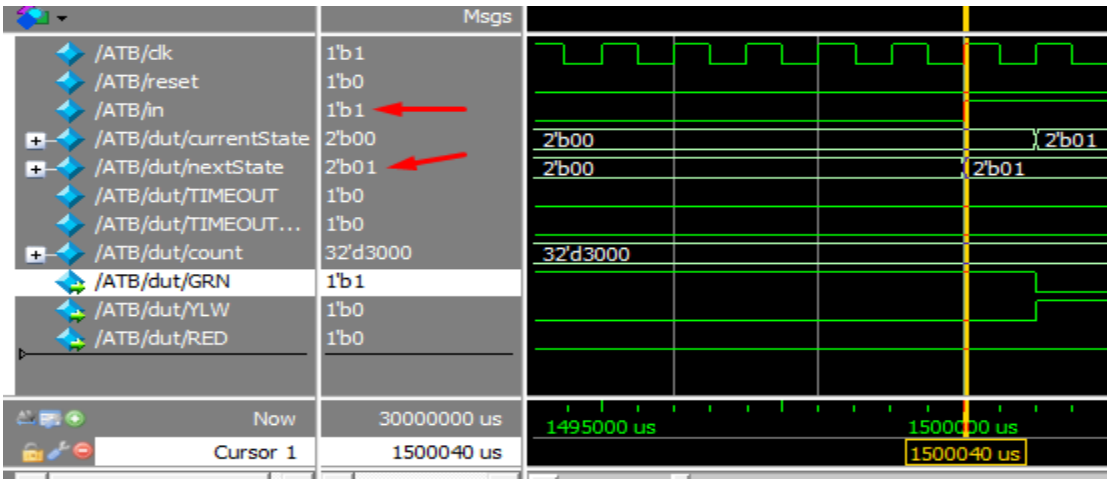
Στα 700000us περιμένουμε το in να γίνει 0 φυσικά χωρίς να έχει αλλάξει ακόμα ούτε το currentState, ούτε οι έξοδοι:



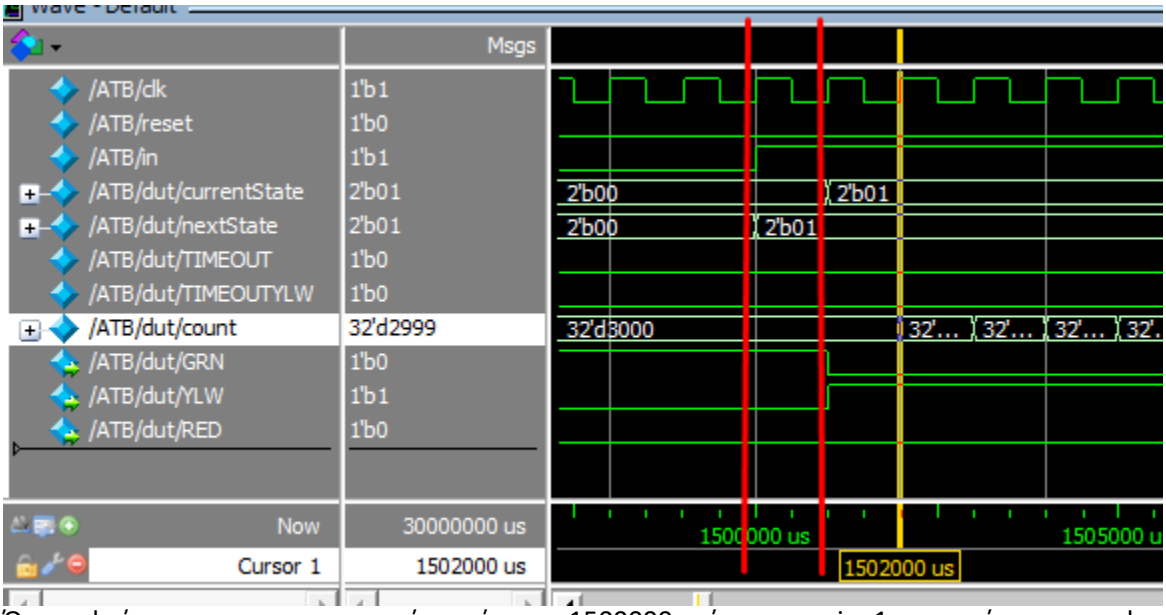
Στο 1.000.000us όπως είπαμε το reset γίνεται 0, και μέχρι να φτάσουμε στα 1.500.000us όπου θα γίνει το in=1, θα μένουμε στην κατάσταση 00->GREEN:



1.500.000us:

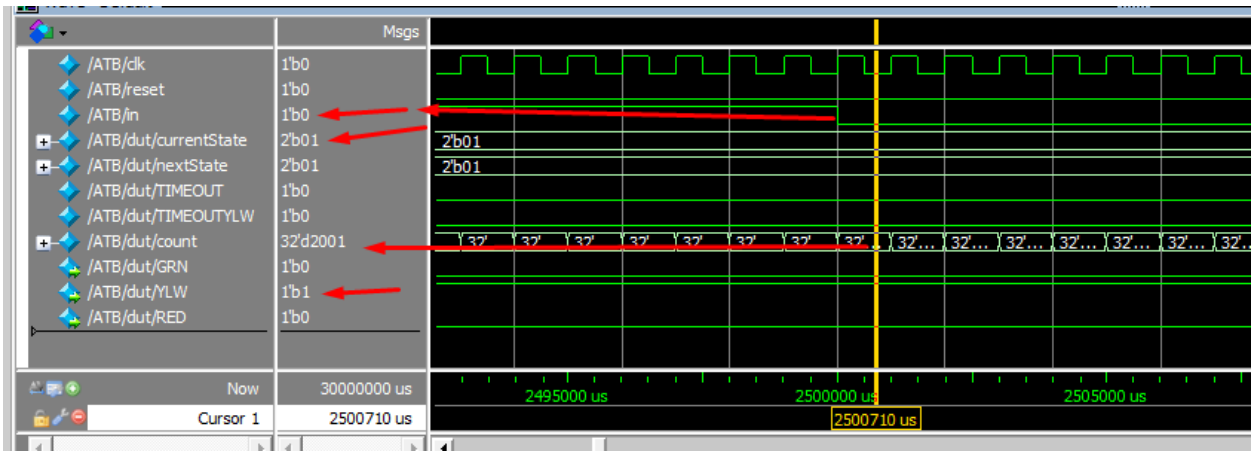


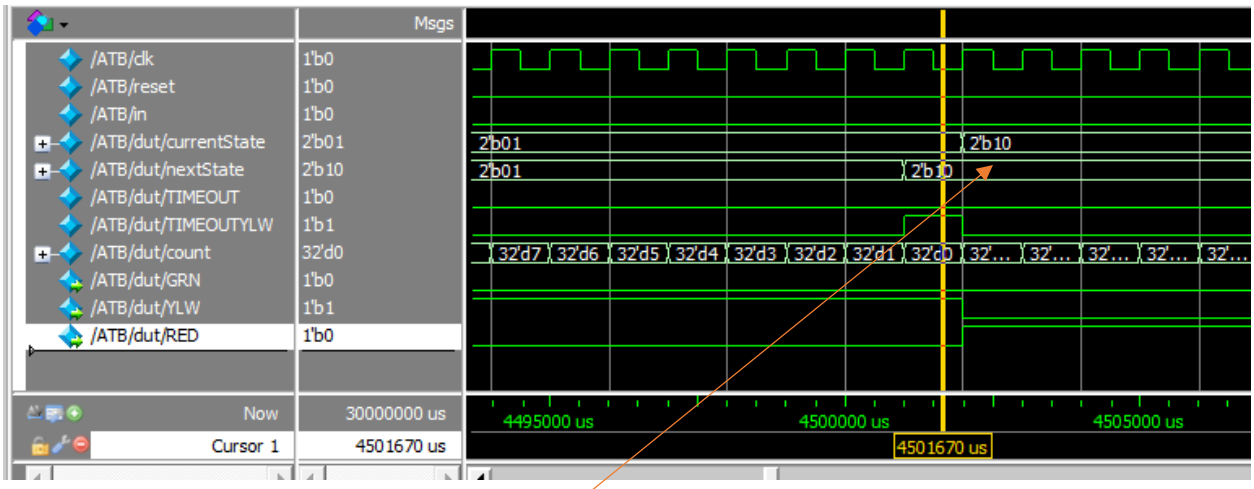
Απο τη στιγμή που θα έρθει λοιπόν το in=1 στα 1,5sec, στον αμέσως επόμενο κύκλο (posedge clock) θα γίνει το currentState = 2'b01->YELLOW, περιμένουμε να τεθεί το count σε 32'd3000 (clock T=1000us, άρα 3000=3sec) και θα αρχίσει να μετράει ο χρόνος αντίστροφα μέχρι να τεθεί το σήμα TIMEOUTYLW σε 1 και να επιτραπεί η μετάβαση απο YELLOW σε REDD:



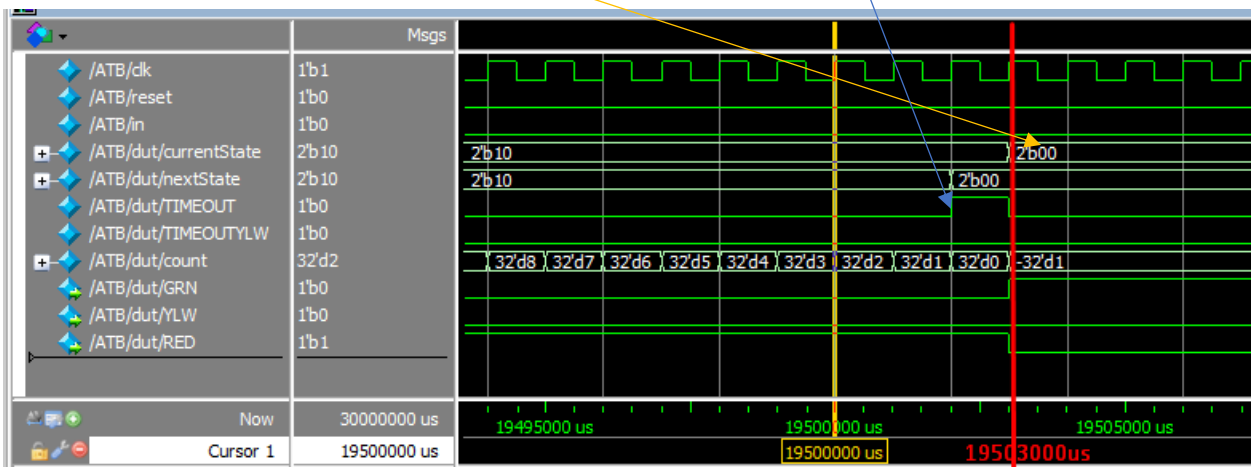
Όντως φαίνεται απο την προσομοίωση ότι στα 1500000us έρχεται το in=1, στο επόμενο posedge clock, 1501000us το currentState γίνεται YELLOW -> 2'b01 και ξεκινάει το delay των 3sec μέχρι να μεταβούμε στην κατάσταση currentState = REDD. Παράλληλα βλέπουμε στο screenshot ότι οι έξοδοι στην currentState = YELLOW είναι {GRN,YLW,RED}={0,1,0} όπως θα έπρεπε.

Σύμφωνα με τις καθυστερήσεις που επιλέχθηκαν στο initial block που καθορίζει τις εισόδους στο testbench, περιμένουμετη στιγμή 2,5sec το in να γίνει 0. Όμως στην κατάσταση YELLOW το FSM ήρθε τη στιγμή 1.5sec, άρα αφού θέλουμε το FSM να μείνει στην YELLOW για 3 δευτερόλεπτα, περιμένουμε να μεταβούμε στην κατάσταση RED τη στιγμή 4,5sec -> 4.500.000us. (Ουσιαστικά προσομοιώνουμε την περίπτωση του να περνάει το αυτοκίνητο με πορτοκαλί φανάρι, και δείχνουμε ότι παρόλο που πέρασε το αυτοκίνητο, το FSM (φανάρι) θα μπει στην κατάσταση REDD αφότου κρατηθεί στη YELLOW για 3 δευτερόλεπτα, και μετά, αφότου περάσουν τα 15sec παραμονής στο REDD, θα μεταβεί πάλι στην κατάσταση GREEN.) Αυτό λοιπόν φαίνεται και από την προσομοίωση:



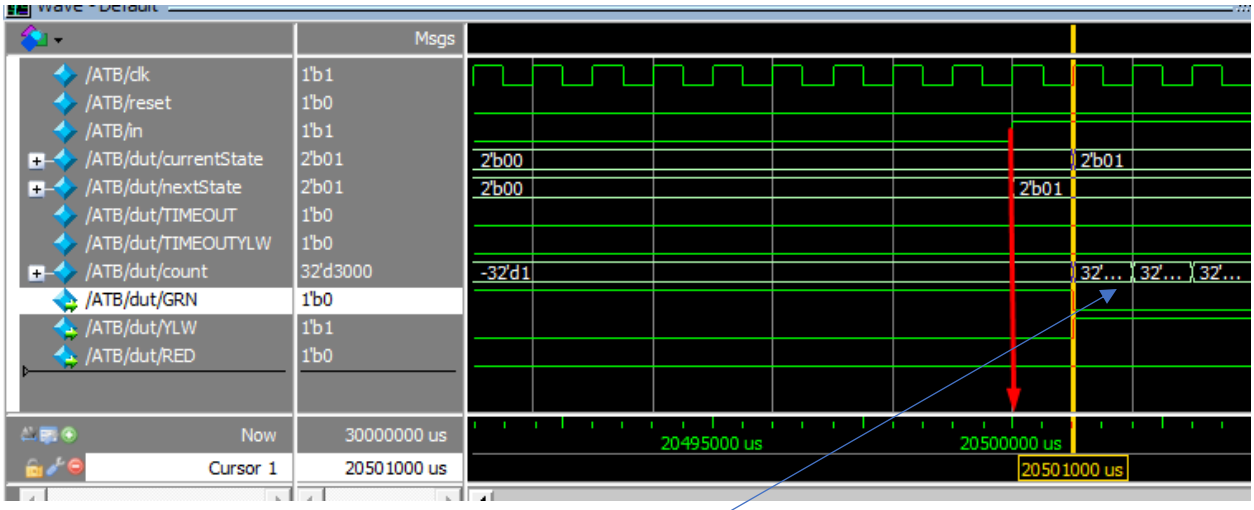


Στο παραπάνω screenshot βλέπουμε ότι τη στιγμή 4501000us που πέρασαν τα 3sec παραμονής στην κατάσταση YELLOW (στην κατάσταση YELLOW μπηκαμε στα 1501000us) τέθηκε το σήμα TIMEOUTYLW σε 1, και έτσι στο επόμενο posedge clock (4502000us) το FSM μεταβαίνει στην κατάσταση 2'b10 όπου οι έξοδοι γίνονται {GRN,YLW,RED}={0,0,1}. Στην κατάσταση REDD θέλουμε να μείνει το FSM για 15 δευτερόλεπτα, οπότε περιμένουμε τη στιγμή 19.503.000us να μεταβούμε στην κατάσταση currentState=GREEN. (15.000.000 το delay για να τεθεί το σήμα TIMEOUT σε ένα + ένας κύκλος για να γίνει η μετάβαση-> 15.001.000. Ξεκίνησε η αντίστροφη μέτρηση των 15sec τη στιγμή 4.502.000us άρα 15.001.000+4.502.000=19.503.000us)



Στην εικόνα απο πάνω βλέπουμε ότι τη στιγμή 19.503.000us η έξοδος γίνεται: {GRN,YLW,RED}={1,0,0} όπως θα έπρεπε καθώς μπήκε το FSM στην κατάσταση currentState=2'b00->GREEN.

Τώρα σύμφωνα με το initial block του testbench που θέτει τις εισόδους, περιμένουμε τη στιγμή 20,5sec να γίνει το in=1, άρα το FSM θα μείνει στην κατάσταση currentState=GREEN μέχρι τη στιγμή 20.501.000us:



Όντως φαίνεται απο το screenshot ότι την στιγμη 20.500.000us->20.5sec «έρχεται αυτοκίνητο στον κάθετο δρόμο» (in=1) και στον posedge clock που ακολουθεί μπαίνει το FSM στην κατάσταση YELLOW (2'b01) και ξεκινάει το countdown των 3 δευτερολέπτων κτλ απο την αρχή.

Με βάση την προσομοίωση διαπιστώνουμε πως όταν το FSM βρίσκεται στην κατάσταση GREEN, με reset=0, όταν το in (CAR) γίνει 1, μεταβαίνει στην κατάσταση YELLOW, μένει σε αυτήν (ανεξάρτητα φυσικά από την είσοδο CAR) για 3 δευτερόλεπτα, μετά μεταβαίνει στην κατάσταση REDD, παραμένει σε αυτήν για 15sec και ύστερα μεταβαίνει στην κατάσταση GREEN και μένει σε αυτήν μέχρι να διαπιστώσει ότι ήρθε πάλι αυτοκίνητο στον κάθετο δρόμο (CAR=1) όπου ξεκινάει όλος αυτός ο κύκλος από την αρχή.