ΨΗΦΙΑΚΑ ΣΥΣΤΗΜΑΤΑ HW ΣΕ ΧΑΜΗΛΑ ΕΠΙΠΕΔΑ ΛΟΓΙΚΗΣ ΙΙ

ΑΝΑΦΟΡΑ ΕΡΓΑΣΙΑΣ

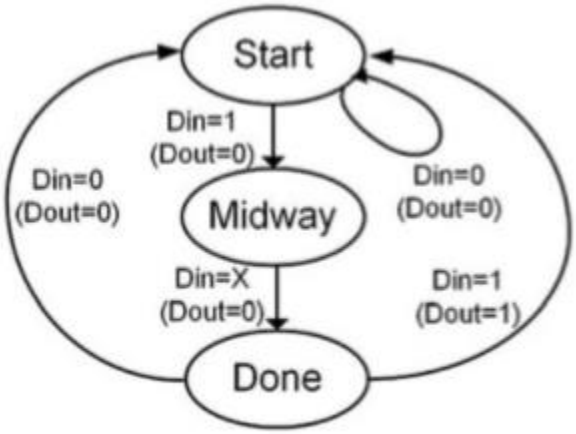
Ονοματεπώνυμο: Ναπολέων Παπουτσάκης

ΑΕΜ: 9170

Email: napoleop@ece.auth.gr

ΑΣΚΗΣΗ 1

Στην πρώτη άσκηση κληθήκαμε να υλοποιήσουμε σε *Verilog* το *FSM* που απεικονίζεται στο Σχήμα 1 καθώς επίσης και να το επαληθεύσουμε με κατάλληλο *testbench* αρχείο. Οι καταστάσεις του *FSM* κωδικοποιούνται με δύο διαφορετικούς τρόπους όπως ορίζεται στην εκφώνηση: α) σαν συνήθεις δυαδικοί αριθμοί, β) με την κωδικοποίηση «*one-hot*».



Σχήμα 1. Διάγραμμα μεταβάσεων για την άσκηση 1.

Περιγραφή λειτουργίας

Στο Σχήμα 1, παρουσιάζονται τρεί διαφορετικές καταστάσεις στις οποίες μπορούμε να βρεθούμε και αυτές είναι οι α) ***Start***, β)***Midway*** , γ)***Done*** καθώς επίσης, οι είσοδοι ***Din*** που εισάγονται κάθε φορά για να μεταβούμε σε μια κατάσταση και η αντίστοιχη έξοδος ***Dout***. Αρχικά, ξεκινάμε απο την κατάσταση *Start*, στην οποία και θα παραμείνουμε εφόσον η είσοδος *Din* είναι 0 με την έξοδο *Dout* να μένει επίσης στο 0. Αντίθετα, εάν η είσοδος γίνει 1, οδηγούμαστε στην κατάσταση *Midway*, και απο εκεί στην κατάσταση *Done* ανεξάρτητα απο το ποία θα είναι η είσοδος μας (***Din = X***) με την έξοδο *Dout* να είναι μηδέν και στις 2 τελευταίες περιπτώσεις. Τέλος, απο την κατάσταση *Done* θα μεταφερθούμε στην κατάσταση *Start* είτε η είσοδος μας είναι 0, είτε είναι 1 με τη διαφορά ότι στην πρώτη περίπτωση η έξοδος *Dout* θα παραμείνει στο 0, ενώ στη δεύτερη η έξοδος *Dout* θα γίνει 1. Σε αυτή την τελευταία μετάβαση γίνεται αντιληπτό ότι πρόκειται για μηχανή πεπερασμένης κατάστασης τύπου ***Mealy***, καθώς η έξοδος δεν εξαρτάται μόνο απο την παρούσα κατάσταση αλλα και απο την είσοδο *Din*.

Α) Υλοποίηση με τις καταστάσεις κωδικοποιημένες σαν συνήθεις δυαδικοί αριθμοί

Κωδικοποίηση Καταστάσεων

Όπως αναφέρθηκε, υπάρχουν τρείς καταστάσεις στο *FSM* μας, επομένως απαιτούνται 2 *Flip Flops* τα οποία υποθέτουμε ότι είναι *D-FFs*. Η κωδικοποίηση των καταστάσεων με απλούς δυαδικούς αριθμός φαίνεται στον Πίνακα 1 που ακολουθεί.

Κατάσταση	Κωδικοποίηση $D_{1:0}$
Start	0 0
Midway	0 1
Done	1 0

Πίνακας 1: Κωδικοποίηση καταστάσεων

Δυο *FF* μπορούν γενικά να κωδικοποιήσουνε μέχρι και 4 καταστάσεις επομένως έχουμε μια παραπάνω κατάσταση που δεν χρησιμοποιείται στη μηχανή μας.

Πίνακας Αληθείας

Ο πίνακας αληθείας του *FSM* παρουσιάζεται στον Πίνακα 2.

Τρέχουσα Κατασταση	Κωδικοποίηση D_1 D_0	Είσοδος Din	Επόμενη Κατάσταση	Κωδικοποίηση D_1 D_0	Έξοδος Dout
Start	0 0	1	Midway	0 1	0
Start	0 0	0	Start	0 0	0
Midway	0 1	X	Done	1 0	0
Done	1 0	0	Start	0 0	0
Done	1 0	1	Start	0 0	1

Πίνακας 2: Πίνακας Αληθείας με απλή κωδικοποίηση.

Λογικές εξισώσεις εξόδων

Με τη βοήθεια του Πίνακα 3 εξάγεται η εξισώση εξόδου του *FSM*, η οποία είναι
Dout = D1*Din.

Τρέχουσα Κατασταση D_1 D_0	Είσοδος Din	Έξοδος Dout
0 0	1	0
0 0	0	0
0 1	X	0
1 0	0	0
1 0	1	1

Πίνακας 3: Πίνακας για την εξαγωγή της λογικής εξίσωσης εξόδου.

Ανάλυση κώδικα *Verilog* για το *FSM*

Για την ανάπτυξη του κώδικα της άσκησης 1, δόθηκε ένας αρχικός ορισμός των θυρών του *FSM* όπου εμπεριέχονται τα σήματα εισόδου και εξόδου όπως φαίνονται στην Εικόνα 1.

```
module fsm1_behavioral
      (output reg Dout,
       input wire, Clock, Reset, Din)
```

Εικόνα 1: Ορισμός θυρών

Αρχικά στο αρχείο ***fsm1_behavioral.v*** ορίζουμε 3 τοπικές παραμέτρους (*localparam*) των 2 bit, τις *start*, *midway* και *done* που θα έχουν η κάθε μια την τιμή της κωδικοποίησης της κάθε κατάστασης αντίστοιχα. Απο εδώ και πέρα θα αναφερόμαστε σε κάθε κατάσταση όχι με την τιμή της κωδικοποίησης της αλλα με το όνομα της αντίστοιχης παραμέτρου. Έπειτα, δηλώνουμε το σήματα τύπου *reg* των 2 *bit currentState* και *nextState* στα οποία θα αποθηκεύεται η τρέχουσα και η επόμενη κατάσταση του *FSM*(για αυτο χρειάζονται πάλι 2 bit, όπως και παραπάνω). Για την υλοποίηση της μηχανής *FSM* χρειάστηκαν 3 *procedural blocks* η λειτουργικότητα των οποίων θα περιγραφεί παρακάτω .

1^{ο} Procedural Block – Αποθήκευσης Κατάστασης

Σε αυτό το *block* - **ακολουθιακής λογικής**- διατηρείται η κατάσταση της μηχανής σύμφωνα με τις μεταβάσεις του σήματος ρολογιού (*Clock*) και επαναφοράς (*Reset*). Συγκεκριμένα, όπως φαίνεται στην Εικόνα 2, έχουμε ένα *always block* όπου στο *sensitivity list* του έχουμε συμπεριλάβει το *posedge Clock* και το *negedge Reset*. Έτσι, το *block* εκτελείται κάθε φορά που

έχουμε κάποια θετική ακμή του ρολογιού, με την τρέχουσα κατάσταση να ανανεώνεται, ή κάποιο πάτημα του *Reset*(ασύγχρονα) το οποίο είναι *active low* (*negedge*) και αρχικοποιεί το *FSM* στην κατάσταση *Start*.

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

Εικόνα 2: Block αποθήκευσης κατάστασης

2^{o} Procedural Block – Επόμενης Κατάστασης

Σε αυτό το *block* - **συνδιαστικής λογικής**- θα βρεθεί η επόμενη κατάσταση που θα μεταβεί το *FSM* με βάση την είσοδο που έχει δοθεί και την τρέχουσα κατάσταση του. Στη ***sensitivity list*** αυτού του *always block* υπάρχει το σήμα ***currentState*** και η είσοδος ***Din***, έτσι το *block* αυτό εκτελείται κάθε φορά που αλλάζει η τρέχουσα κατάσταση ή η είσοδος *Din*. Για τον προσδιορισμό της μετάβασης στην επόμενη κατάσταση χρησιμοποιείται μια ***case*** εντολή, ώστε να γίνουν διαφορετικές αναθέσεις ανάλογα την τρέχουσα κατάσταση και την είσοδο. Για την περίπτωση που η τρέχουσα κατάσταση είναι η *Start*, εάν η είσοδος είναι 1 τότε μεταβαίνουμε στην κατάσταση *Midway* αλλιώς παραμένουμε στην κατάσταση *Start*. Για τις περιπτώσεις όπου η τρέχουσα κατάσταση είναι *Midway* ή *Done* η επόμενη κατάσταση θα είναι *Done* ή *Start*, αντίστοιχα(Εικόνα 3).

```
always @(currentState or Din)
     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
```

Εικόνα 3: Block Επόμενης Κατάστασης

3ο Procedural Block – Έξοδος

Στο τελευταίο αυτό *block* - **συνδιαστικής λογικής** - θα αποφασιστεί ποιά θα είναιη έξοδος. Αφού όπως αναφέρθηκε πρόκειται για μηχανή πεπερασμένης κατάστασης τύπου ***Mealy***, την τιμή της εξόδου καθορίζουν τόσο η τρέχουσα κατάσταση όσο και η είσοδος. Επομένως το *always block* της Εικόνας 4, περιλαμβάνει στο ***sensitivity* list** του το ***currentState*** και την είσοδο ***Din***. Με μια *case* εντολή και πάλι, έαν η τρέχουσα κατάσταση είναι *Done* τότε εάν η είσοδος είναι 1, η έξοδος *Dout* είναι 1. Σε κάθε άλλη περίπτωση(έαν δηλαδή η είσοδος είναι 0 και η τρέχουσα κατάσταση *Done* ή αν η τρέχουσα κατάσταση είναι οποιαδήποτε άλλη) η έξοδος *Dout* είναι 0.

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

Εικόνα 4: Block Εξόδου

Συνολική Περιγραφή της Μηχανής

```
module fsm1_behavioral(output reg Dout, input wire Clock, Reset, Din);
 //declaration of necessary parameters
        localparam [1:0]
                start = 2'b00,
                midway = 2'b01,
                done = 2'b10;
        reg [1:0] currentState, nextState; // declaration of two signal
```

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

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

```
always @(currentState or Din)
     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
```

Ανάλυση κώδικα *Verilog* για το *Testbench* του *FSM*

Για να επαληθευτεί η λειτουργία του *FSM* δημιουργήθηκε ενα *testbench* αρχείο, ***fsm1_behavioral_TB.v***. Σε αυτό το αρχείο παράγονται όλοι οι συνδυασμοί διανυσμάτων εισόδων για το προς επαλήθευση κύκλωμα, ώστε να παρατηρηθεί αν η έξοδος είναι η αναμενόμενη σε κάθε περίπτωση.

Αρχικά, ορίζουμε το timescale σε *1ns*/*1ns* (κλίμακα 1 ns και ακρίβεια 1ns), καθώς στη συνέχεια θα χρησιμοποιηθούν καθυστερήσεις, καθώς επίσης και τις εισόδους *clk* , *reset*, in ως ***reg*** οι οποίες θα οδηγήσουν τις αντίστοιχες εισόδους του ***DUT*** (***D****esign* ***U****nder* ***T****est*) αλλα και την έξοδο out ως ***wire*** που θα οδηγηθεί απο την έξοδο *Dout* του *DUT*. Αμέσως μετά, ορίζεται το DUT fsm1_behavioral, με την δήλωση των θυρών να γίνεται **ρητά** (Εικόνα 5).

```
`timescale 1ns/1ns
module fsm1_behavioral_TB;
        reg clk;
        reg reset;
        reg in;
        wire out;
        fsm1_behavioral dut(.Clock(clk), .Reset(reset), .Din(in), .Dout(out));
        initial
           begin
                 reset = 1'b0;
            #15  reset = 1'b1;
        end
```

Εικόνα 5: Ορισμός του DUT

Έπειτα, στο *initial block* που ακολουθει και εκτελείται μια φορά, το *reset* τίθεται σε 0 - οπότε και ενεργοποιείται (*active low*), και μετά απο 15ns(#15-μη συνθέσιμη εντολή) γίνεται 1.

```
initial
   begin
         reset = 1'b0;
    #15  reset = 1'b1;
end
```

Εικόνα 6: Initial block για το reset

Αμέσως μετά, υπάρχει ακόμα ένα *initial block* όπου αρχικοποιείται το *clk* στην τιμή 0, ενω στο *always block* που ακολουθεί μοντελοποιείται το ρολόι το οποίο αλλάζει τιμή επ'αόριστον (Εικόνα 7). Βάζοντας #10 αναγκάζουμε το *clk* να αλλάζει τιμή(*low* σε *high* και αντίστροφα) κάθε 10 ns οπότε το clock θα έχει περίοδο 20ns.

```
initial
    begin
        clk = 1'b0;
end
always
    begin
        #10 clk = ~clk;
end
```

Εικόνα 7: Blocks για το Clock

Στο τελευταίο *initial block* (Εικόνα 8)υπάρχει μια σειρά απο διαφορετικές τιμές εισόδου που τίθενται μετά απο δοαφορετικά χρονικά διαστήματα έτσι ώστε να ελεγχθεί κατα πόσο το *FSM* ανταποκρίνεται στις εισόδους μεταβαίνοντας στις σωστές καταστάσεις και βάζοντας σωστές τιμές στην έξοδο *Dout*.

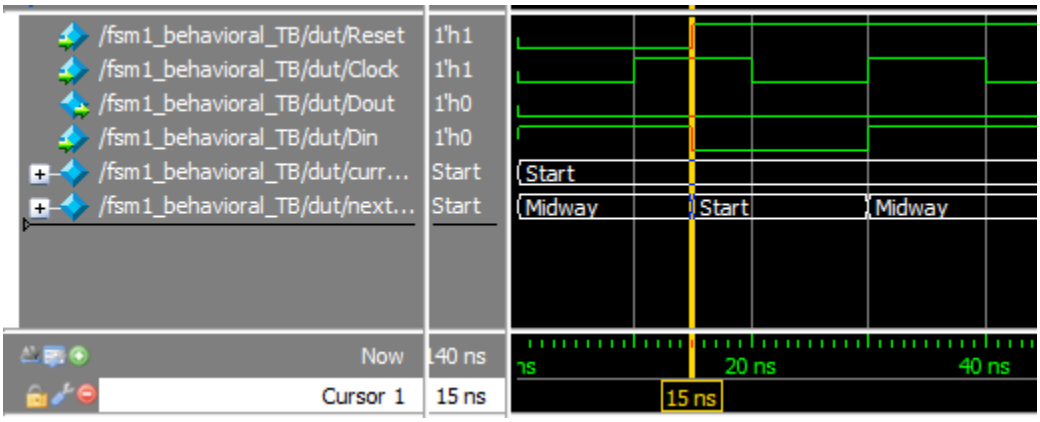
```
initial
   begin
            in = 1'b1;
      #15   in = 0'b0;
      #15   in = 1'b1;
      #20   in = 1'b0;
      #30   in = 1'b1;
      #10   in = 1'b1;
      #20   in = 1'b1;
      #10   in = 1'b0;
   end
```

Εικόνα 7: Block για τις διαφορετικές εισόδους

Προσομοίωση και Επαλήθευση του FSM

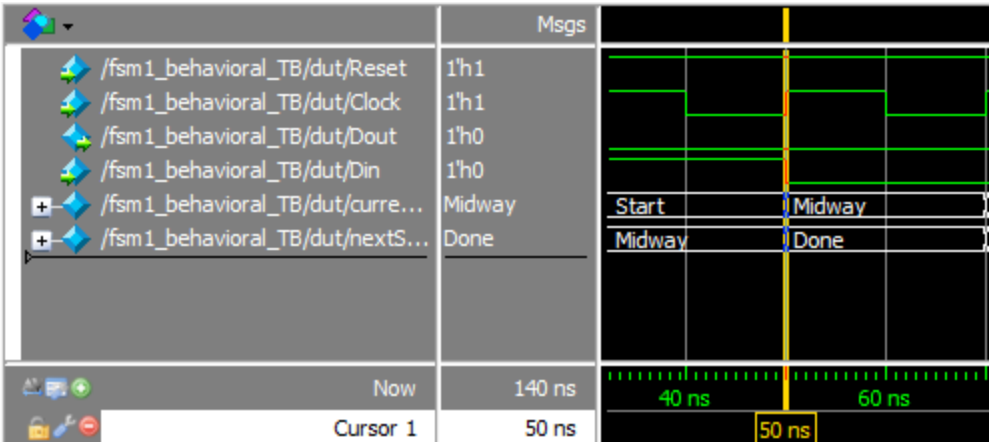
Για την επαλήθευση του *FSM* έγινε προσομοίωση του *testbench* που περιγράφηκε παραπάνω και μέσω *screenshots* της προσομοίωσης σε διάφορα χρονικά διαστήματα θα εξηγηθεί η καλή λειτουργία του.

Αρχικά στο πρώτο *screenshot*(Εικόνα 8) βλέπουμε ότι μεχρι το 15^{o} ns το Reset παραμένει στο 0, δηλαδή είναι ενεργοποιημένο(*active low*), για αυτό και παρόλο που η είσοδος *Din* είναι στο 1 μέχρι το 15^{o} ns και το *nextState* αλλάζει(γίνεται *Midway*), το *currentState* δεν ενημερώνεται και παραμένει στην κατάσταση *Start(εδω φαίνεται και το ασυγχρονο Reset, καθώς η τρέχουσα κατάσταση παίρνει την τιμή start πρίν την πρώτη θετική ακμή του ρολογιού)*. Στο 15^{o} ns, το *Reset* γινεται 1(απενεργοποιείται) όπου και θα παραμείνει για τη όλη τη συνέχεια,η είσοδος *Din* γίνεται 0 και η επόμενη κατάσταση γίνεται *Start*. Στην επόμενη θετική ακμή του ρολογιού, δηλαδή στο 30^{o} ns θα ενημερωθεί το *currentState*, αλλα λόγω του οτι η επόμενη κατάσταση είναι πάλι *Start*, δεν υπάρχει κάποια διαφοροποίηση στην κυματομορφη. Στο ίδιο ns η είσοδος *Din* γίνεται 1, οπότε η επόμενη κατάσταση ενημερώνεται αμέσως και γίνεται *Midway* ενω η έξοδος *Dout* είναι στο 0.



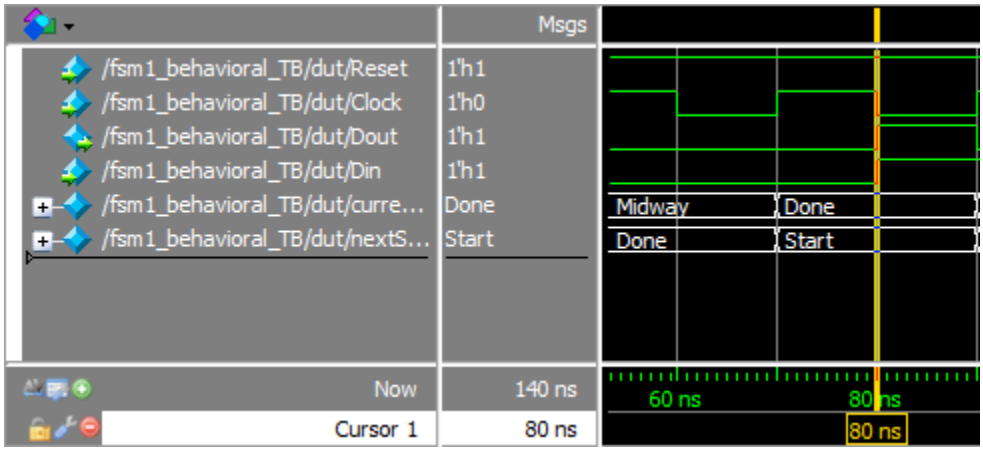
Εικόνα 8: Κυματομορφές για τα πρώτα 40 ns

Όπως φαίνεται στην Εικόνα 9, στην αμέσως επόμενη θετική ακμη του ρολογιού(50ns) θα ενημερωθεί η τρέχουσα κατάσταση και θα γίνει *Midway* και η είσοδος αλλάζει και πάλι παίρνοντας την τιμή 0. Εφόσον βρισκόμαστε στην *Midway* κατάσταση η τιμή της εισόδου δεν καθορίζει την μετάβαση, αφού όπως είδαμε, θα μεταβούμε έτσι κι αλλιώς στην κατάσταση *Done*, με έξοδο 0. Οπότε, παρατηρούμε όντως ότι η επόμενη κατάσταση, έχει γίνει *Done* και η έξοδος συνεχίζει να είναι 0.



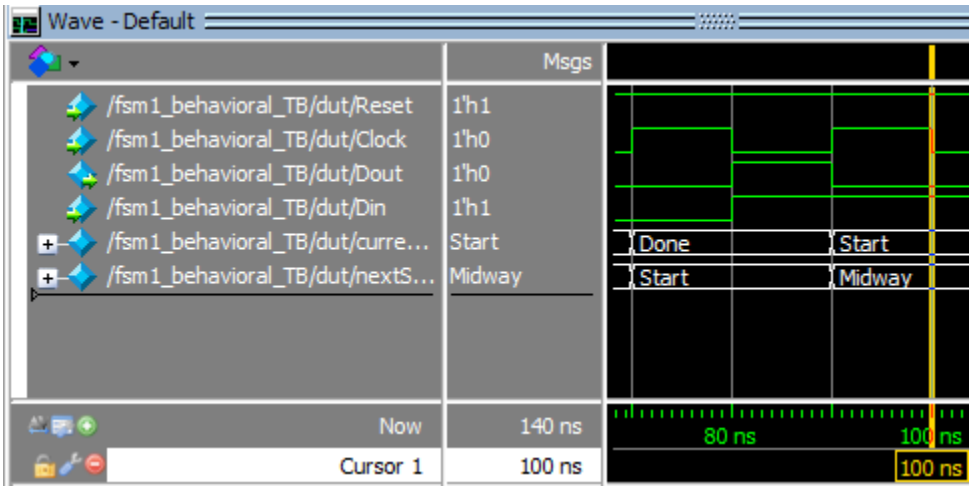
Εικόνα 9: Κυματομορφές για 40 - 60 ns

Η επόμενη θετική ακμή του ρολογίου έρχεται στα 70 ns, οπότε και η τρεχουσα κατάσταση γίνεται *Done*. Η επόμενη κατάσταση θα είναι ανεξαρτήτως εισόδου η *Start*, καθώς το αν το Din είναι 1 ή 0 θα καθορίσει μόνο την έξοδο. Επομένως σε αυτό το σημείο η αλλαγή της επόμενης κατάστασης οφείλεται αποκλειστικά στην αλλαγή της τρέχουσας κατάστασης (το *currentState* βρίσκεται στη sensitivity list του *next state logic block*). Μέχρι και τα 80ns η έξοδος παραμένει στο 0. Τότε όμως η είσοδος γίνεται 1, και η έξοδος μεταβαίνει επίσης στο 1 ασύγχρονα. Η συμπεριφόρά αυτή είναι σωστή αν αναλογιστούμε ότι η είσοδος *Din* (και το *currentState*) είναι στο *sensitivity list* του *block* εξόδου μας και η τρέχουσα κατάσταση είναι Done .



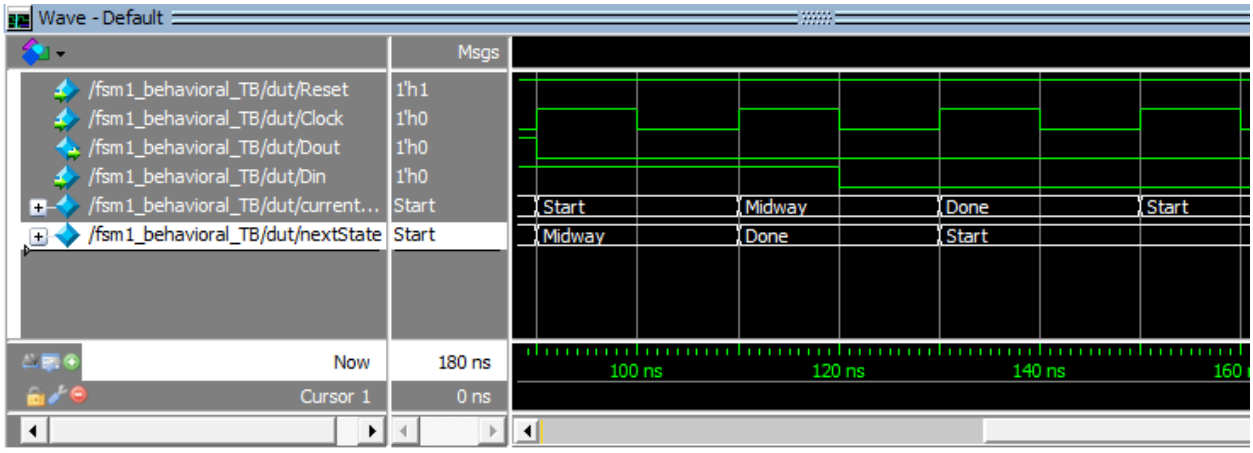
Εικόνα 10: Κυματομορφές για 60 - 80 ns

Στο 90^{o} ns έρχεται το επόμενο *positive edge* του *clock*, η τρέχουσα κατάσταση μεταβαίνει στη *Start*, η είσοδος παραμένει στο 1 άρα η επόμενη κατάσταση σωστα γίνεται *Midway* και η έξοδος *Dout* πέφτει ξανά στο μηδέν(Εικόνα 11).



Εικόνα 11: Κυματομορφές για 80 - 100 ns

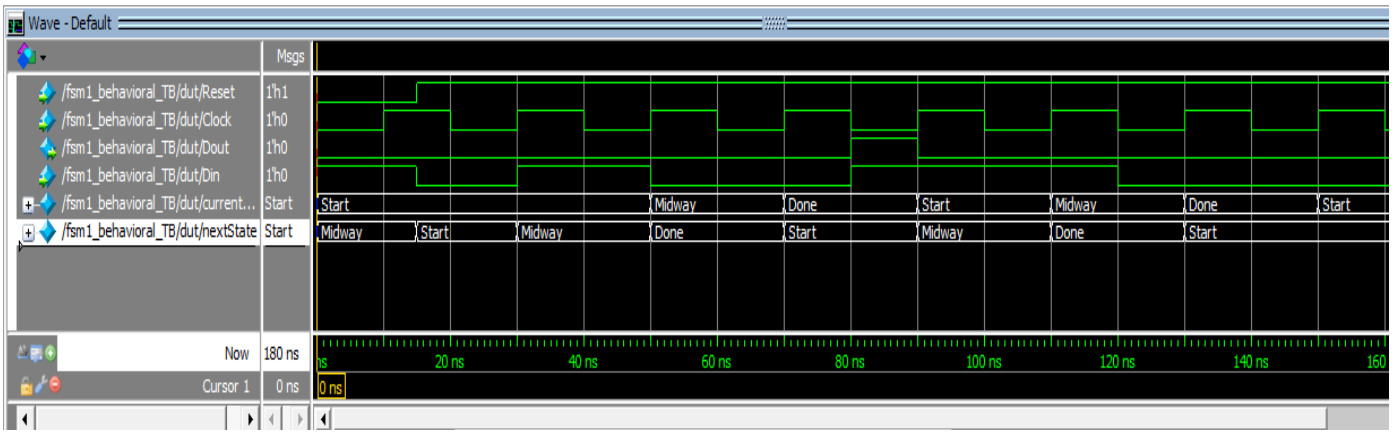
Στα 110ns(Εικόνα 12) ενημερώνεται η τρέχουσα κατάσταση σε *Midway*, καθώς έχουμε θετική ακμή του ρολογιού και η επόμενη κατάσταση γίνεται *Done*. Βλέπουμε ότι η επόμενη κατάσταση έγινε *Done* με την είσοδο *Din* να είναι στο 1, σε αντίθεση με πρίν που μεταβήκαμε απο τη *Midway* στη *Done* με το *Din* στο 0, κάτι που αποδεικνύει ότι η μετάβαση αυτή είναι ανεξάρτητη της εισόδου(όπως απαιτούσε η εκφώνηση). Η έξοδος *Dout* παραμένει στο 0.



Εικόνα 12: Κυματομορφές για 100 - 150 ns

Στα 120 ns, βλέπουμε απο την Εικόνα 12, ότι η είσοδος γίνεται 0, η έξοδος συνεχίζει να είναι 0, όμως και η επόμενη και η τρέχουσα κατάσταση ακόμα παραμένουν στις προηγούμενες τιμές τους περιμένοντας την θετική ακμή του ρολογιού. Αυτή θα έρθει, στα 130 ns, όπου η τρέχουσα κατάσταση θα γίνει *Done* και καθώς η είσοδος παραμένει στο 0, η έξοδος δεν αλλάζει τιμή και μένει και αυτή στο 0, ενώ ενημερώνεται και η επόμενη κατάσταση η οποία γίνεται *Start*. Στην επόμενη θετική ακμή του clock η τρέχουσα κατάσταση κατάσταση θα πάρει την τιμή *Start* και εδώ τελειώνει η προσομοίωση.

Στην παρακάτω φωτογραφία(Εικόνα 13) φαίνεται και η ολόκληρη η προσομοίωση.



Εικόνα 13: Πλήρης προσομοίωση

Β) Υλοποίηση με τις καταστάσεις κωδικοποιημένες με την κωδικοποίηση “one-hot”

Κωδικοποίηση Καταστάσεων

Η κωδικοποίηση των καταστάσεων με κωδικοποίηση “one-hot” φαίνεται στον Πίνακα 4 που ακολουθεί.

Κατάσταση	Κωδικοποίηση $D_{2:0}$
Start	0 0 1
Midway	0 1 0
Done	1 0 0

Πίνακας 4: Κωδικοποίηση one-hot

Πίνακας Αληθείας

Ο πίνακας αληθείας του *FSM* παρουσιάζεται στον Πίνακα 5.

Τρέχουσα Κατασταση	Κωδικοποίηση D_2 D_1 D_0	Είσοδος Din	Επόμενη Κατάσταση	Κωδικοποίηση D_2 D_1 D_0	Έξοδος Dout
Start	0 0 1	1	Midway	0 1 0	0
Start	0 0 1	0	Start	0 0 1	0
Midway	0 1 0	X	Done	1 0 0	0
Done	1 0 0	0	Start	0 0 1	0
Done	1 0 0	1	Start	0 0 1	1

Πίνακας 5: Πίνακας αλήθειας με κωδικοποίηση One-hot

Λογικές εξισώσεις εξόδων

Με τη βοήθεια του Πίνακα 6 εξάγεται η εξισώση εξόδου του *FSM*, η οποία είναι

Dout = Din*D2

Τρέχουσα Κατασταση D_2 D_1 D_0	Είσοδος Din	Έξοδος Dout
0 0 1	1	0
0 0 1	0	0
0 1 0	X	0
1 0 0	0	0
1 0 0	1	1

Πίνακας 6: Πίνακας για την εξαγωγή της λογικής εξίσωσης εξόδου

Ανάλυση κώδικα *Verilog* για το *FSM*

Ο κώδικας ως προς τη λειτουργικότητα είναι ακριβώς ίδιος με τον αντίστοιχο για την απλή κωδικοποίηση που αναλύθηκε παραπάνω. Οι μόνες αλλαγές έγιναν στα σήματα *currentState* και *nextState* καθώς επίσης και στις παραμέτρους *start*,*midway* και *done* που δηλώθηκαν να παίρνουν 3-bit. Έτσι, οι παράμετροι πήραν τις τιμές 001, 010, 100 αντίστοιχα. Ο κώδικας παρατίθεται παρακάτω.

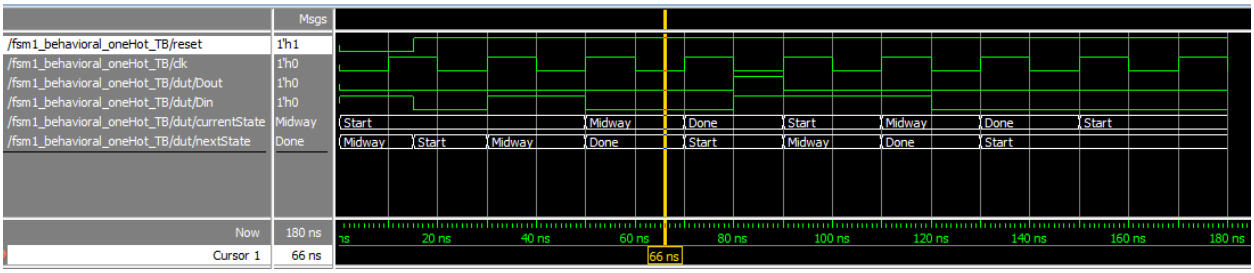
```
module fsm1_behavioral_oneHot(output reg Dout, input wire Clock, Reset, Din);

        localparam [2:0]
                start = 3'b001,
                midway = 3'b010,
                done = 3'b100;
        reg [2:0] currentState, nextState;

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

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

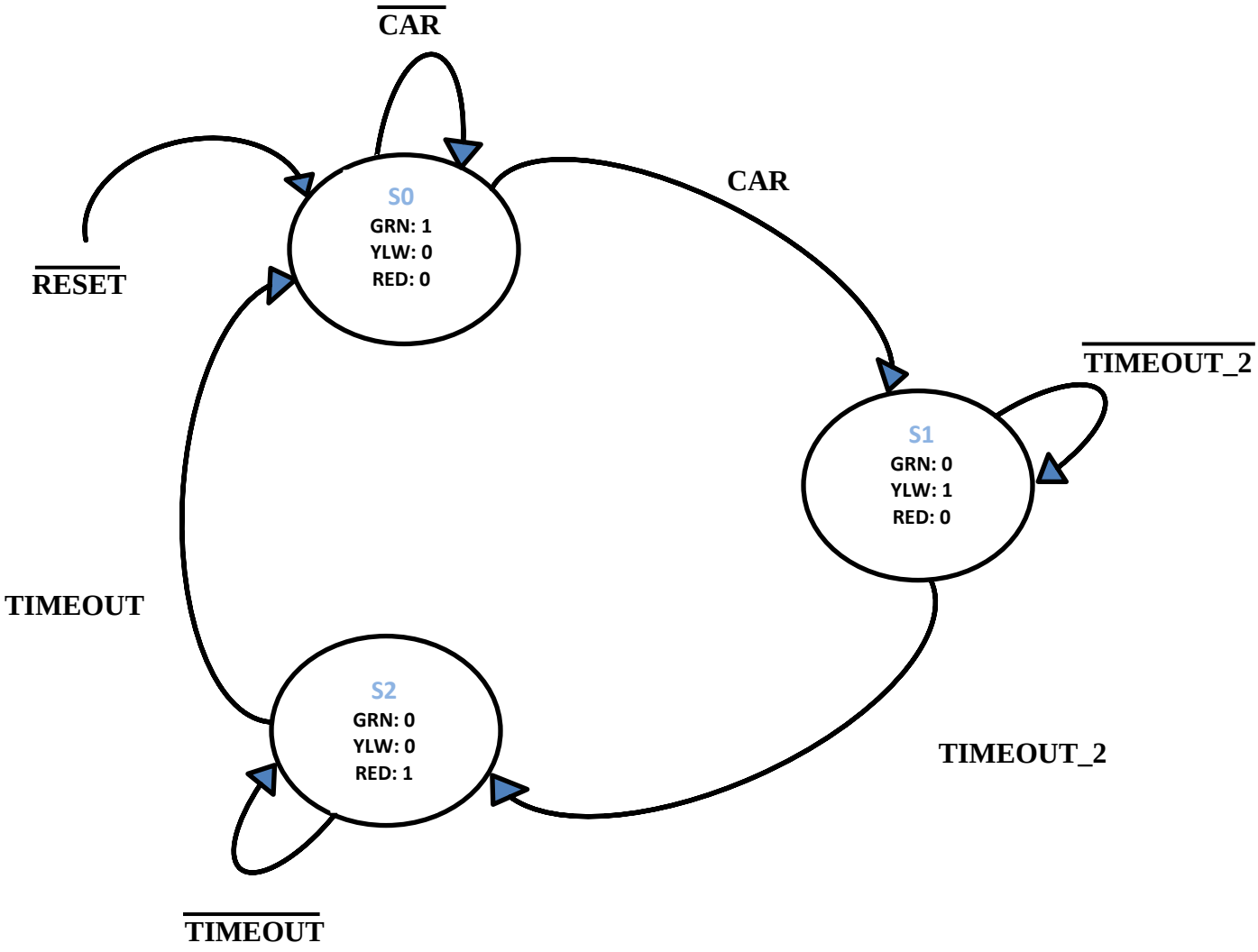
Προσομοίωση και Επαλήθευση

Για την επαλήθευση του FSM έγινε προσομοίωση του ίδιου *testbench* μεσω του οποίου επαληθεύτηκε το πρώτο FSM με μόνη ουσιαστική αλλαγή το *DUT*. Τα αποτελέσματα είναι τα ίδια και στην παρακάτω εικόνα φαίνεται ολόκληρη η προσομοίωση.



ΑΣΚΗΣΗ 2

Στη δεύτερη άσκηση κληθήκαμε να υλοποιήσουμε σε Verilog ένα μοντέλο συμπεριφοράς ενός *FSM*, για έναν ελεγκτή φανών σήμανσης που ελέγχει την κίνηση ενός δρόμου ταχείας κυκλοφορίας καθώς επίσης και να το επαληθεύσουμε με το κατάλληλο *testbench*. Το διάγραμμα μεταβάσεων των καταστάσεων φαίνεται στο Σχήμα 2.



Σχήμα 2: Διάγραμμα μεταβάσεων καταστάσεων

Απο το Σχήμα 2, βλέπουμε ότι κατα το Reset(όταν πάρει την τιμή 0) το *FSM* ξεκινάει απο την κατάσταση S0, στην οποία το φανάρι του δρόμου είναι πράσινο. Σε αυτήν την κατάσταση παραμένουμε όσο το σήμα *CAR* είναι 0, μόλις το σήμα αυτό τεθεί στο 1, σημαίνει ότι ο αισθητήρας που βρίσκεται στον κάθετο δρόμο είδε κάποιο αμάξι, οπότε μεταβαίνουμε στην κατάσταση S1 όπου το φανάρι είναι πορτοκαλί στην οποία θα μείνουμε μέχρι να τεθεί στο 1 το

σήμα *TIMEOUT_2* κάτι που θα γίνει μετά απο 3 δευτερόλεπτα οπότε και μεταβαίνουμε στην κατάσταση *S2* με το φανάρι να γίνεται κόκκινο. Αφού περάσουν 15 δευτερόλεπτα(τα οποία μετρά ένας ενσωματωμένος μετρητής) τίθεται το σήμα *TIMEOUT* και το φανάρι γίνεται ξανά πράσινο(κατάσταση S0) ενω προτού περάσουν τα 15 δευτερόλεπτα παραμένουμε στην καταστασηη όπου το φανάρι είναι κόκκινο(S2). Απο τα παραπάνω φαίνεται ότι η μηχανή πεπερασμένης κατάστασης που μας ζητείται να υλοποιήσουμε είναι τύπου ***Moore***, καθώς η έξοδος εξαρτάται αποκλειστικά απο την τρέχουσα κατάσταση.

Κωδικοποίηση Καταστάσεων

Όπως αναφέρθηκε, υπάρχουν τρείς καταστάσεις στο *FSM* μας, επομένως απαιτούνται 2 *Flip Flops* τα οποία υποθέτουμε ότι είναι *D-FFs*. Η κωδικοποίηση των καταστάσεων με απλούς δυαδικούς αριθμός φαίνεται στον Πίνακα 7 που ακολουθεί.

Κατάσταση	Κωδικοποίηση D_1 D_0
S0	0 0
S1	0 1
S2	1 0

Πίνακας 7: Κωδικοποίηση καταστάσεων με απλούς δυαδικούς

Δυο *FF* μπορούν γενικά να κωδικοποιήσουνε μέχρι και 4 καταστάσεις επομένως έχουμε μια παραπάνω κατάσταση που δεν χρησιμοποιείται στη μηχανή μας.

Πίνακας Αληθείας

Ο πίνακας αληθείας του *FSM* παρουσιάζεται στον Πίνακα 8.

Τρέχουσα Κατασταση	Κωδικοποίηση D1 D0	Έξοδος RED	Έξοδος GRN	Έξοδος YLW
S0	0 0	0	1	0
S1	0 1	0	0	1
S2	1 0	1	0	0

Πίνακας 8: Πίνακας Αληθείας Ασκησης 2

Λογικές εξισώσεις εξόδων

Με τη βοήθεια του Πίνακα 8 εξάγονται οι εξισώσεις εξόδου του *FSM*, οι οποίες είναι

$$GRN = \overline{D1} * \overline{D0}$$

$$YLW = \overline{D1} * D0$$

$$RED = D1 * \overline{D0}$$

Τρέχουσα Κατάσταση D1	D0	Έξοδοι GRN	YLW	RED
0	0	1	0	0
0	1	0	1	0
1	0	0	0	1

Πίνακας 8: Πίνακας για την εξαγωγή της λογικής εξίσωσης εξόδου.

Ανάλυση κώδικα Verilog για το FSM

Αρχικά στο αρχείο ***fsm2_Lights.v*** , ορίζουμε 3 τοπικές παραμέτρους (*localparam*) των 2 *bit*, τις *S0*, *S1* και *S2* που θα έχουν η κάθε μια την τιμή της κωδικοποίησης της κάθε κατάστασης αντίστοιχα. Έτσι, απο εδώ και πέρα θα αναφερόμαστε σε κάθε κατάσταση με το όνομα της αντίστοιχης παραμέτρου και όχι με την τιμή της κωδικοποίησης της. Αμέσως μετά, δηλώνουμε δύο σήματα τύπου *reg* των 2 *bit currentState* και *nextState* στα οποία θα αποθηκεύεται η τρέχουσα και η επόμενη κατάσταση του *FSM* (για αυτο και χρησιμοποιούμε 2 *bit* όπως και παραπάνω). Επίσης δηλώνουμε τέσσερα ακόμα *reg* σήματα *counter*,και *counter*_2 των 32 bit και *TIMEOUT*, *TIMEOUT*_2 ενός bit (η χρησιμότητα των οποίων θα εξηγηθεί παρακάτω). Για την υλοποίηση της μηχανής *FSM* χρειάστηκαν 4 *procedural blocks* η λειτουργικότητα του κάθενός θα περιγραφεί παρακάτω.

1^{ο} Procedural Block – Αποθήκευσης Κατάστασης

Σε αυτό το *block* - **ακολουθιακής λογικής**- διατηρείται η κατάσταση της μηχανής σύμφωνα με τις μεταβάσεις του σήματος ρολογιού (*Clock*) και επαναφοράς (*Reset*). Συγκεκριμένα, όπως φαίνεται στην Εικόνα 15 , έχουμε ένα *always block* όπου στο *sensitivity list* του έχουμε συμπεριλάβει το *posedge Clock* και το *negedge Reset*. Έτσι, το *block* εκτελείται κάθε φορά που έχουμε κάποια θετική ακμή του ρολογιού,- με την τρέχουσα κατάσταση να ανανεώνεται-, ή κάποιο πάτημα του *Reset*(ασύγχρονα) το οποίο είναι *active low* (*negedge*) και αρχικοποιεί το

FSM στην κατάσταση S0, τους 2 counters στις τιμές 15000 και 3000 και τα σήματα *TIMEOUT* και *TIMEOUT*_2 στο 0.

```
always @(posedge Clock or negedge Reset)
     begin: STATE_MEMORY
        if(!Reset)
          begin
             currentState <= S0; // if reset is active, store S0 in current state..
            // initialization of counters and timeout dignals
             Counter <= 15000;
             Counter_2 <= 3000;
             TIMEOUT <= 0;
             TIMEOUT_2 <= 0;
          end
        else
           currentState <= nextState; // ..else store nextstate in current state
     end
```

Εικόνα 15: Block Αποθήκευσης Κατάστασης

2ο Procedural Block – Επόμενης Κατάστασης

Σε αυτό το *block* - **συνδιαστικής λογικής**- θα βρεθεί η επόμενη κατάσταση που θα μεταβεί το *FSM* με βάση την είσοδο που έχει δοθεί, την τρέχουσα κατάσταση του καθώς επίσης και τα σήματα *CAR*, *TIMEOUT* και *TIMEOUT_2*. Στη *sensitivity list* αυτού του *always block* υπάρχει το σήμα *currentState*, το σήμα *CAR*(είσοδος) και το *posedge* των *TIMEOUT* και *TIMEOUT_2.* Το σήμα *TIMEOUT* όπως υποδεικνύει η εκφώνηση έρχεται όταν ένας ενσωματωμένος μετρητής μετρήσει 15 δευτερόλεπτα,(χρόνος μεταξύ της μετάβασης απο το κόκκινο στο πράσινο) οπότε και το σήμα *TIMEOUT* γίνεται 1. Αντίστοιχα, το σήμα *TIMEOUT_2* γίνεται 1 όταν ένας ενσωματωμένος μετρητής μετρήσει 3 δευτερόλεπτα, χρόνος που θα μεσολαβεί μεταξύ της μετάβασης απο το πορτοκαλί στο κόκκινο για να γίνει ρεαλιστικότερη η προσέγγιση μας. Ο λόγος για τον οποίο μπήκαν αυτά τα σήματα στη *sensitivity list* και μάλιστα ως *posedge TIMEOUT* και *posedge TIMEOUT_2*, είναι διότι θέλουμε να εκτελεστεί το *block*, και να αλλάξουν οι αντίστοιχες καταστάσεις μόνο κατα τη θετική ακμή αυτών των δύο σημάτων, όταν δηλαδή παρέλθει ο απαιτούμενος χρόνος για το καθένα και το σήμα γίνει απο 0 σε 1(και όχι όταν θα επαναφερθούν τα σήματα στο 0).Για τον προσδιορισμό της μετάβασης στην επόμενη κατάσταση χρησιμοποιείται μια ***case*** εντολή, στην οποία ελέγχεται σε ποιά κατάσταση είμαστε κάθε φορά. Εαν είμαστε στην κατάσταση *S0* (φανάρι πράσινο), τότε ελέγχουμε την τιμή του σήματος *CAR*, εαν αυτό είναι 1 η επόμενη κατάσταση γίνεται η *S1*, αλλιώς παραμένουμε στην *S0*. Αν η τρέχουσα κατάσταση είναι η *S1*, τότε μόνο εαν το σήμα *TIMEOUT_2* είναι 1 - που σημαίνει οτι πέρασαν 3 δευτερόλεπτα απο όταν μεταβήκαμε στο πορτοκαλί - η *nextState* παίρνει την τιμή *S2*, αλλιώς δεν γίνεται καμία αλλαγή και παραμένουμε στην κατάσταση *S1*. Τελευταία περίπτωση είναι η τρέχουσα κατάσταση να έχει την τιμή *S2*, οπότε ελέγχουμε την

τιμή του *TIMEOUT*, τότε μόνο εαν το σήμα *TIMEOUT* είναι 1 η επόμενη κατάσταση παίρνει την τιμή *S0,* ειδάλλως παραμένουμε στην κατάσταση *S2* με το φανάρι στο κόκκινο(Εικόνα 16).

```
always @(currentState or CAR or posedge TIMEOUT or posedge TIMEOUT_2)
     begin: NEXT_STATE_LOGIC
//check which is the current state
        case(currentState)
           S0: if(CAR) nextState = S1;// in case of S0 check the CAR si
               else nextState = S0;// ..else set S0 in next state
           S1:if(TIMEOUT_2)// in case of S1 check the TIMEOUT_2 signal,
                 begin
                   nextState = S2;
                 end
           S2: if(TIMEOUT)// in case of S2 check the TIMEOUT signal, it
                  begin
                    nextState = S0;

                  end
               else nextState = S2;
           default: nextState = S0; // in any other case, set S0 in nex
        endcase
end
```

Εικόνα 16: Block Επόμενης Κατάστασης

3ο Procedural Block – Απαριθμητής

Σε αυτό το *block* - **ακολουθιακής λογικής**- υλοποιείται ο *down counter* που θα μας βοηθήσει να μετρήσουμε του χρόνους μεταξύ των μεταβάσεων απο το πορτοκαλι στο κόκκινο και απο το κόκκινο στο πράσινο(οπότε και τίθονται τα σήματα *TIMEOUT*, *TIMEOUT*_2). Στο *sensitivity list* αυτού του *always block* υπάρχει μόνο η θετική ακμή του *clock*, όπως αναμενόταν αφού σκοπός μας είναι να μετρήσουμε χρόνο. Αρχικά, ελέχγουμε αν η τρέχουσα κατάσταση είναι η S2, όπου το φανάρι είναι κόκκινο καθώς τότε μόνο θέλουμε να ξεκινήσει η μέτρηση. Εαν είναι, τότε μειώνουμε κατα 1 τον πρώτο *Counter*, και ακριβώς απο κάτω ελέγχουμε αν η τιμή του *counter* είναι 0, που σημαίνει ότι παρήλθε ο χρόνος που θέλαμε και το σήμα *TIMEOUT* πρέπει να γίνει 1. Στο ίδιο *block* υλοποιείται και ο απαριθμητής που θα μετρήσει το χρόνο μεταξύ της μετάβασης απο το πορτοκαλί στο κόκκινο(απο την *S1* στην *S2* κατάσταση). Έτσι, παρακάτω ελέγχουμε αν η τρέχουσα κατάσταση είναι η *S1*, με το φανάρι πορτοκαλί, καθώς μόνο τότε

```
always @(posedge Clock)
     begin: DOWN_COUNTER
        if(currentState == S2) // check
          begin
             Counter = Counter - 1; //de
             if(Counter == 0)//check if
                begin
                 TIMEOUT  = 1;//change t

                end
          end

        if(currentState == S1) // check
           begin
             Counter_2 = Counter_2 - 1;
             if(Counter_2 == 0) //check
                begin
                        TIMEOUT_2 = 1;//

                end
          end
     end
```

Εικόνα 17: Block Down Counter

θέλουμε ο απαριθμητής να ξεκινήσει να μετρά. Εαν ισχύει η συνθήκη, μειώνουμε κατα ένα τον *Counter*_2, και ελέγχουμε εαν ο *Counter*_2 είναι 0, όμοια με παραπάνω. Αν είναι 0, τότε τίθεται στο ένα και το σήμα *TIMEOUT*_2(Εικόνα 17).

4ο Procedural Block – Εξόδων

Στο τελευταίο αυτό *block* - **συνδιαστικής λογικής** - θα αποφασιστεί ποιες θα είναι οι έξοδοι. Αφού όπως αναφέρθηκε πρόκειται για μηχανή πεπερασμένης κατάστασης τύπου ***Moore***, τις τιμές των εξόδων τις καθορίζει μόνο η τρέχουσα κατάσταση. Έτσι, στο *sensitivity list* του *always block* υπάρχει μόνο το *currentState*. Με μια *case* εντολή ελέγχουμε σε ποία κατάσταση βρισκόμαστε(ποιά είναι η τιμή του *currentState* δηλαδή). Εαν, η τρέχουσα κατάσταση είναι η *S0,* τότε οι έξοδοι *YLW* και *RED* γίνονται 0 και η έξοδος ***GRN*** 1(φανάρι πράσινο) ενω επαναρχικοποιουνται στο 0 το TIMEOUT και ο Counter στην τιμή, 15000 για να είναι έτοιμα για την επόμενη φορά που θα φτάσουμε στην κατάσταση S2 με το φανάρι κόκκινο(οπότε και θα πρέπει να περιμένουμε εκ νέου 15 δευτερόλεπτα προτού μεταβούμε στην κατάσταση S0-φανάρι πράσινο). Εαν είναι *S1*, τότε οι έξοδοι *GRN* και *RED* γίνονται 0 και η ***YLW*** 1(φανάρι πορτοκαλί), ενω τέλος αν είμαστε στην κατάσταση *S2*, η έξοδος ***RED*** γίνεται 1 και οι *GRN* και *YLW* 0 ενω

```
always @(currentState)
    begin:OUTPUT_LOGIC
       //check which is the current st
       case(currentState)
         S0: begin // in case of S0, s
               GRN = 1'b1;
               YLW = 1'b0;
               RED = 1'b0;
               TIMEOUT = 0;
               Counter = 15000; //rese
             end
         S1:begin // in case of S1, se
               GRN = 1'b0;
               YLW = 1'b1;
               RED = 1'b0;
            end
         S2: begin //in case of S2, se
               GRN = 1'b0;
               YLW = 1'b0;
               RED = 1'b1;
               TIMEOUT_2 = 0;
               Counter_2 = 3000; // re
               end
         default: begin // in any othe
                    GRN = 1'b1;
                    YLW = 1'b0;
                    RED = 1'b0;
                   end
       endcase
    end
```

Εικόνα 18: Block Εξόδων

επαναφέρονται οι αρχικές τιμές στα σήματα *TIMEOUT*_2(στην τιμή 0) και *Counter*_2(στην τιμή 3000) για να είναι έτοιμα για την επόμενη φορά που θα φτάσουμε στην κατάσταση *S1* (φανάρι πορτοκαλί)(Εικόνα 18).

Συνολική Περιγραφή της Μηχανής

```
module fsm2_Lights(output reg GRN, YLW, RED, input wire Clock, Reset, CAR);
        // declaration of necessary local parameters
        localparam [1:0]
                S0 = 2'b00,
                S1 = 2'b01,
                S2 = 2'b10;
        reg [1:0] currentState, nextState; // declaration of two signals fo
        reg [31:0] Counter, Counter_2; //declaration of two 32bit counters
        reg TIMEOUT, TIMEOUT_2; // declare the TIMEOUT and TIMEOUT_2 signal
```

```
always @(currentState)
     begin:OUTPUT_LOGIC
        //check which is the current
        case(currentState)
          S0: begin // in case of S0
                GRN = 1'b1;
                YLW = 1'b0;
                RED = 1'b0;
                TIMEOUT = 0;
                Counter = 15000; //
              end
          S1:begin // in case of S1,
                GRN = 1'b0;
                YLW = 1'b1;
                RED = 1'b0;
             end
          S2: begin //in case of S2,
                GRN = 1'b0;
                YLW = 1'b0;
                RED = 1'b1;
                TIMEOUT_2 = 0;
                Counter_2 = 3000; //
                end
          default: begin // in any c
                     GRN = 1'b1;
                     YLW = 1'b0;
                     RED = 1'b0;
                    end
        endcase
     end
```

```
always @(posedge Clock or negedge Reset)
     begin: STATE_MEMORY
        if(!Reset)
          begin
             currentState <= S0; // if r
            // initialization of counter
             Counter <= 15000;
             Counter_2 <= 3000;
             TIMEOUT <= 0;
             TIMEOUT_2 <= 0;
          end
        else
           currentState <= nextState; //
     end
```

```
always @(posedge Clock)
     begin: DOWN_COUNTER
        if(currentState == S2) // check
          begin
             Counter = Counter - 1; //d
             if(Counter == 0)//check if
                begin
                 TIMEOUT  = 1;//change

                end
          end

        if(currentState == S1) // check
           begin
             Counter_2 = Counter_2 - 1;
             if(Counter_2 == 0) //check
                begin
                        TIMEOUT_2 = 1;/

                end
          end
     end
```

```
always @(currentState or CAR or posedge TIMEOUT or posedge TIMEOUT_2)
     begin: NEXT_STATE_LOGIC
//check which is the current state
        case(currentState)
           S0: if(CAR) nextState = S1;// in case of S0 check the CAR si
               else nextState = S0;// ..else set S0 in next state
           S1:if(TIMEOUT_2)// in case of S1 check the TIMEOUT_2 signal,
                 begin
                   nextState = S2;
                 end
           S2: if(TIMEOUT)// in case of S2 check the TIMEOUT signal, it
                  begin
                    nextState = S0;

                  end
               else nextState = S2;
           default: nextState = S0; // in any other case, set S0 in nex
        endcase
end
```

Ανάλυση κώδικα *Verilog* για το *Testbench* του *FSM*

Για να επαληθευτεί η λειτουργία του *FSM* δημιουργήθηκε ενα *testbench* αρχείο, ***fsm2_Lights_TB.v***. Σε αυτό το αρχείο παράγονται όλοι οι συνδυασμοί διανυσμάτων εισόδων για το προς επαλήθευση κύκλωμα, ώστε να παρατηρηθεί αν οι έξοδοι είναι οι αναμενόμενες σε κάθε περίπτωση.

Αρχικά, ορίζουμε το timescale σε *100us*/*10us* (κλίμακα 100 us και ακρίβεια 10us), καθώς στη συνέχεια θα χρησιμοποιηθούν καθυστερήσεις, καθώς επίσης και τις εισόδους *Clock_TB*, *Reset_TB*, *CAR_TB* ως ***reg*** οι οποίες θα οδηγήσουν τις αντίστοιχες εισόδους του ***DUT*** (***D****esign* ***U****nder* ***T****est*) αλλα και τις εξόδους *GRN_TB, YLW_TB, RED_TB* ως ***wire*** που θα οδηγηθούν απο τις εξόδους *GRN, YLW* και *RED* του *DUT*. Αμέσως μετά, ορίζεται το *DUT fsm2_Lights*, με την δήλωση των θυρών να γίνεται **ρητά** (Εικόνα 19).

```
`timescale 100us/10us
module fsm2_Lights_TB;

        reg Clock_TB;
        reg Reset_TB;
        reg CAR_TB;
        wire GRN_TB;
        wire YLW_TB;
        wire RED_TB;

        fsm2_Lights dut(.Clock(Clock_TB),
                .Reset(Reset_TB), .CAR(CAR_TB),
                .YLW(YLW_TB), .RED(RED_TB), .GRN(GRN_TB));
```

Εικόνα 19: Ορισμός του DUT

Έπειτα, στο *initial block* που ακολουθει και εκτελείται μια φορά, το *Reset_TB*- οπότε και ενεργοποιείται (*active low*) - και το *CAR_TB* τίθονται στο 0, και μετά απο 1000us(#10-μη συνθέσιμη εντολή) το Reset_TB γίνεται 1, ώστε το σύστημα να ξεκινήσει να λειτουργεί(Εικόνα 20).

```
initial
   begin

        Reset_TB = 1'b0;
        CAR_TB = 0;
   #10  Reset_TB = 1'b1;
```

Εικόνα 20: Initial block για το Reset

Αμέσως μετά, υπάρχει ακόμα ένα *initial block* όπου αρχικοποιείται το *Clock_TB* στην τιμή 0, ενω στο *always block* που ακολουθεί μοντελοποιείται το ρολόι το οποίο αλλάζει τιμή επ'αόριστον. Βάζοντας #5 αναγκάζουμε το *Clock_TB* να αλλάζει τιμή(*low* σε *high* και αντίστροφα) κάθε 500 us οπότε το clock θα έχει περίοδο 1000us(Εικόνα 21).

```
initial
     begin
         Clock_TB = 1'b0;
 end

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

Εικόνα 21: Initial και Always Blocks για το Clock

Στο τελευταίο initial block (Εικόνα 22)υπάρχει μια σειρά απο διαφορετικές τιμές εισόδου CAR που τίθενται μετά απο διαφορετικά χρονικά διαστήματα έτσι ώστε να ελεγχθεί κατα πόσο το FSM ανταποκρίνεται στις εισόδους μεταβαίνοντας στις σωστές καταστάσεις και βάζοντας σωστές τιμές στις εξόδους GRN, YLW, RED.

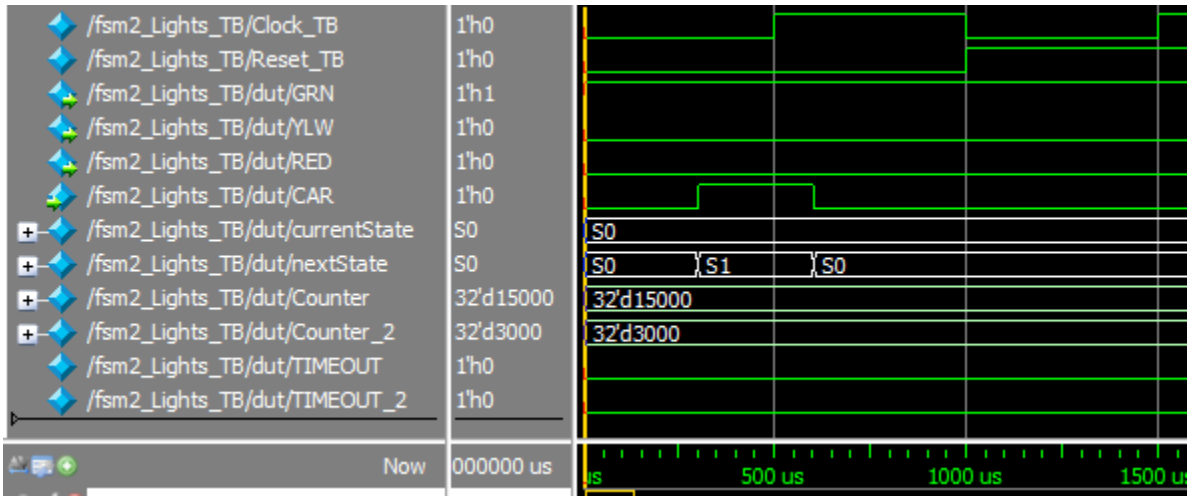
```
initial
   begin
     #3  CAR_TB = 1'b1;//300us
     #3  CAR_TB = 1'b0;//600us
     #19 CAR_TB = 1'b1;//2.5ms
     #75 CAR_TB = 1'b0;//10ms
     #190000 CAR_TB = 1'b1;//19.01 sec
     #80 CAR_TB = 1'b0;

   end
```

Εικόνα 22: Initial Blocks για τις διαφορετικές εισόδους

Προσομοίωση και Επαλήθευση

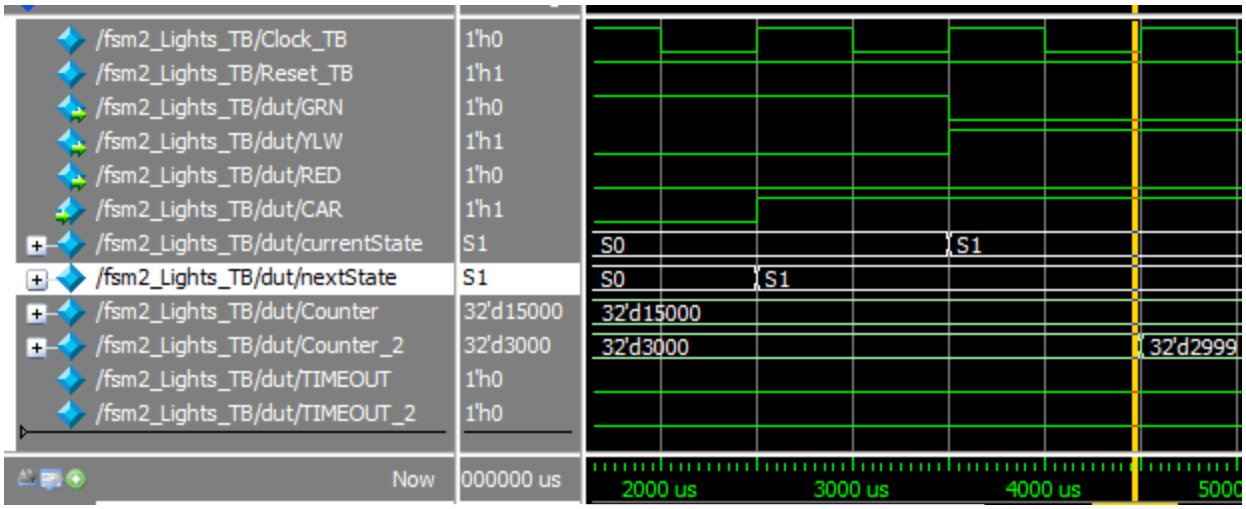
Για την επαλήθευση του *FSM* έγινε προσομοίωση του *testbench* που περιγράφηκε παραπάνω και μέσω *screenshots* της προσομοίωσης σε διάφορα χρονικά διαστήματα θα εξηγηθεί η καλή λειτουργία του.

Αρχικά, στο πρώτο *screenshot*(Εικόνα 23) βλέπουμε ότι μέχρι το 1000^{o} us το *Reset* παραμένει στο 0, δηλαδή είναι ενεργοποιημένο (active low), για αυτό παρόλο που η είσοδος *CAR* στο 300^{o} us γινεται 1 και η επόμενη κατάσταση αλλάζει σε *S1*, στην θετική ακμή του ρολογιού που έρχεται στο 500^{o} us ,η τρέχουσα κατάσταση δεν ενημερώνεται και παραμένει στην S0 και οι έξοδοι *GRN*, *YLW*, *RED* έχουν τιμές 1,0,0 αντίστοιχα όπως και θα έπρεπε αφού το φανάρι είναι πράσινο. Στο 600^{o} us η είσοδος *CAR* πέφτει πάλι στο 0 και στο 1000^{o} us το *Reset* γίνεται 1(απενεργοποιείται) επιτρέποντας στο σύστημα να λειτουργήσει σωστά απο εδω και στο εξής. Επίσης παρατηρούμε οτι οι 2 μετρητές και τα δύο σήματα *TIMEOUT* και *TIMEOUT_2* έχουν πάρει τις αρχικές τους τιμές, με τα δύο τελευταία να περιμένουμε να αλλάξουν αφού οι αντίστοιχοι μετρητές φτάσουν στον μηδέν.



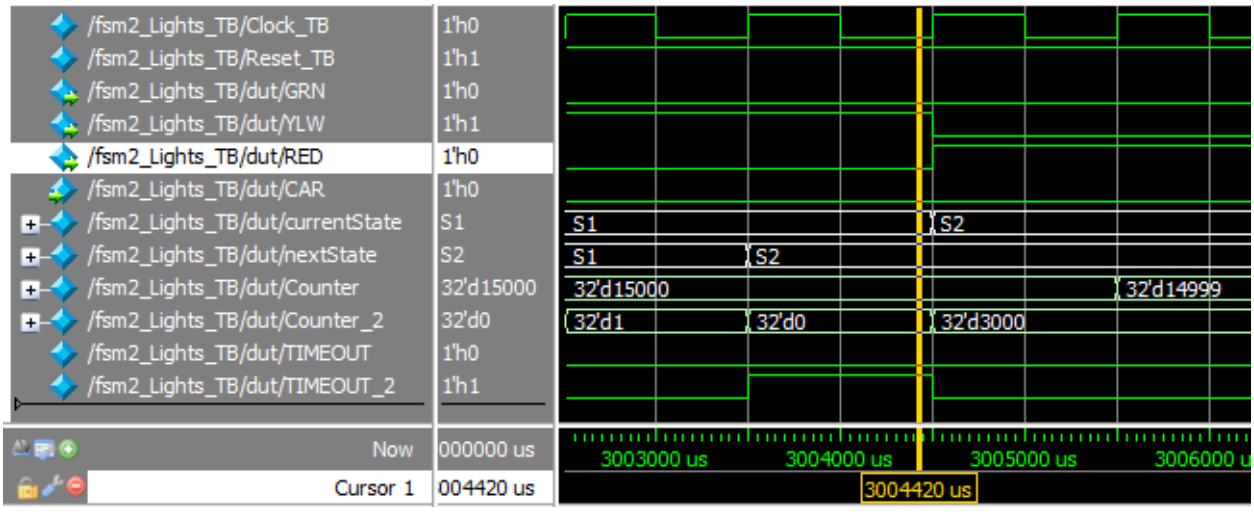
Εικόνα 23: Κυματομορφές για τα πρώτα 1000 us

Η τιμή της εισόδου *CAR* αλλάζει στα 2500 us και γίνεται 1, αμέσως ενημερώνεται η επόμενη κατάσταση σε *S1*, ενω στην επόμενη θετική ακμή του ρολογιού στα 3500us θα ενημερωθεί και η τρέχουσα κατάσταση και οι έξοδοι θα αλλάξουν τιμές με το *YLW* να γίνεται 1 και τις *GRN* και *RED* να είναι 0. Στο επόμενο *posedge* του *clock*, βλέπουμε οτι ο *Counter_2* έχει γίνει 2999 καθώς παρήλθε ένας κύκλος ρολογιού απο την στιγμή που το σύστημα βρέθηκε στην κατάσταση που το φανάρι ήταν πορτοκαλί και επομένως ξεκίνησε η αντίστροφη μέτρηση των τριών δευτερολέπτων έως ότου μεταβούμε στην κατάσταση *S2* και το κόκκινο φανάρι (Εικόνα 24).



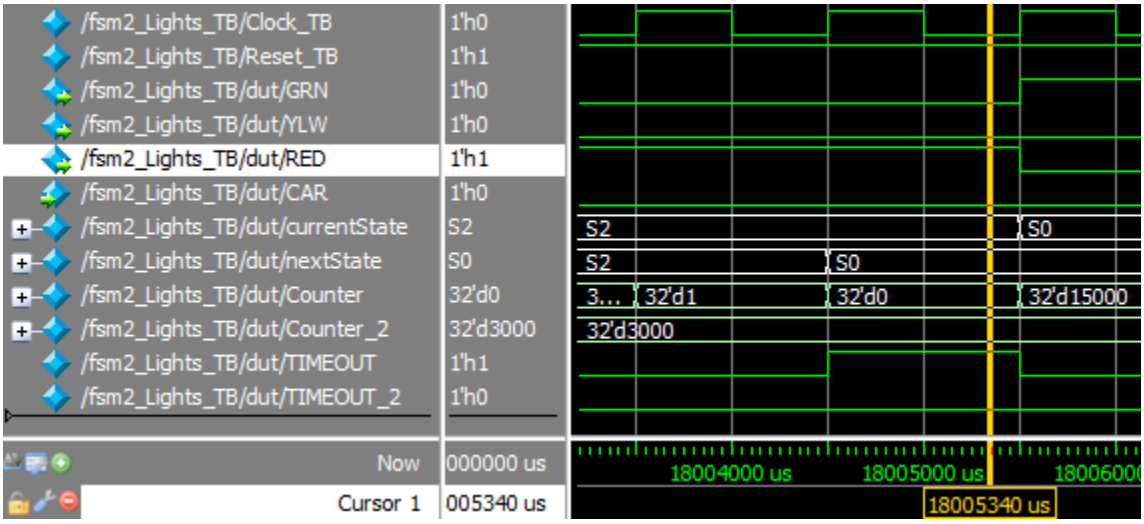
Εικόνα 24: Κυματομορφές για 2500 - 4500 us

Αφού παρέλθουν τα 3 δευτερόλεπτα, οταν δηλαδή φτάσουμε στα 3.003.500 us ο *Counter*_2 γίνεται 0 ,η επόμενη κατάσταση γίνεται *S2*(φανάρι κόκκινο) και το *TIMEOUT*_2 γίνεται 1 οπότε στην επόμενη θετική ακμή του ρολογιού(3.004.500 us) ενημερώνεται η τρέχουσα κατάσταση σε *S2*, οι έξοδοι *GRN*, *YLW*, *RED* 0,0,1 αντίστοιχα, το σήμα TIMEOUT_2 επανέρχεται στο 0 και η τιμή του *Counter*_2 γίνεται ξανά 3000. Ουσιαστικά δηλαδή δεν μεσολλάβησαν 3 δευτερόλεπτα μεταξύ του πορτοκαλί και του κόκκινου, αλλα 3 δευτερόλεπτα και ένας ακόμα κύκλος ρολογιού(1ms), χρόνος ο οποίος μπορεί να θεωρηθεί αμελητέος. Έναν ακόμα κύκλο αργότερα και στην επόμενη θετική ακμή του ρολογιού (3.005.500 us) παρατηρούμε ότι ο *Counter* έχει γίνει 14999 καθώς παρήλθε ένας κύκλος ρολογιού απο τη στιγμή που η τρέχουσα κατάσταση έγινε S2- φανάρι κόκκινο- και ξεκίνησε η αντίστροφη μέτρηση των 15 δευτερολέπτων έως ότου μεταβούμε στην κατάσταση S0 και το πράσινο φανάρι(Εικόνα 25).



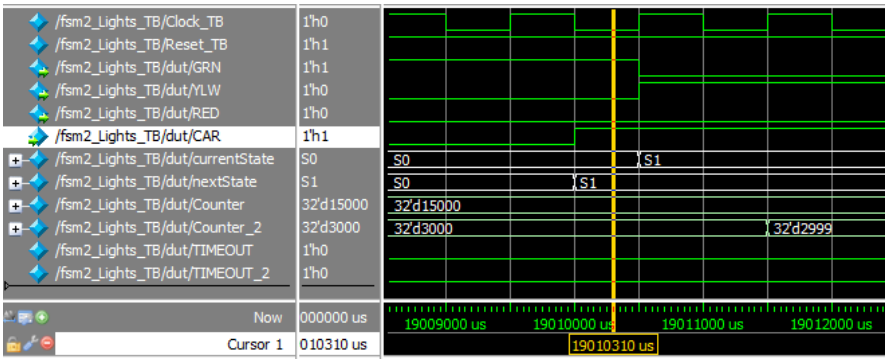
Εικόνα 25: Κυματομορφές για 3.003.500 - 3.005.500 us

Αφού περάσουν 15 δευτερόλεπτα απο όταν ξεκίνησε η αντίστροφη μέτρηση του *Counter*, οπότε και θα έχουμε φτάσει περίπου στα 18 δευτερόλεπτα και 4.5ms(18.004.500us) ο *Counter* γίνεται 0, η επόμενη κατάσταση ενημερώνεται και παίρνει την τιμή S0 και το σήμα *TIMEOUT* γίνεται 1. Στην επόμενη θετική ακμή του ρολογιού η τρέχουσα κατάσταση θα πάρει την τιμή S0 ο μετρητής (Counter) θα αρχικοποιηθεί στο 15000, το σήμα *TIMEOUT* θα επανέλθει στο 0 και οι έξοδοι *GRN*, *YLW*, *RED* θα γίνουν 1,0,0 καθώς το φανάρι στην κατάσταση S0 πρέπει να είναι πράσινο.Παρατηρούμε ότι πρίν μηδενιστεί ο Counter(καθώς και ενα κύκλο ρολογιού μετά το μηδενισμό) οι έξοδοι GRN, YLW, RED παρέμειναν-σωστά- στις τιμές 0,0,1 αντίστοιχα, καθώς η κατάσταση του συστήματος ήταν ακόμα η S2-φανάρι κόκκινο. Και εδώ υπάρχει η ίδια απόκλιση του ενός παραπάνω κύκλου ρολογιού (1ms) στη διάρκεια που μένει στο κόκκινο το φανάρι αλλα θεωρείται επίσης αμελητέα(Εικόνα 26). Μέχρι εδώ το σύστημα μας έχει κάνει έναν σωστό κύκλο ξεκινώντας απο την κατάσταση S0 και το φανάρι πράσινο και καταλήγοντας πάλι εκεί.



Εικόνα 26: Κυματομορφές για 18.004.500 - 18.005.500 us

Στο 19^{ο} δευτερόλεπτο και κάτι(στα 19.010.000 us) το *CAR* γίνεται πάλι 1, η επόμενη κατάσταση ενημερώνεται σε S1 με το φανάρι πορτοκαλί και στο επόμενο *posedge* του *Clock* η τρέχουσα κατάσταση γίνεται S1 με τον μετρητή *Counter*_2 να ξεκινά ξανά την αντίστροφη μέτρηση των 3 δευτερολέπτων και τη διαδικασία να επαναλαμβάνεται όπως ακριβώς και παραπάνω(Εικόνα 27).



Εικόνα 27: Κυματομορφές για 19.010.000 - 19.011.500 us