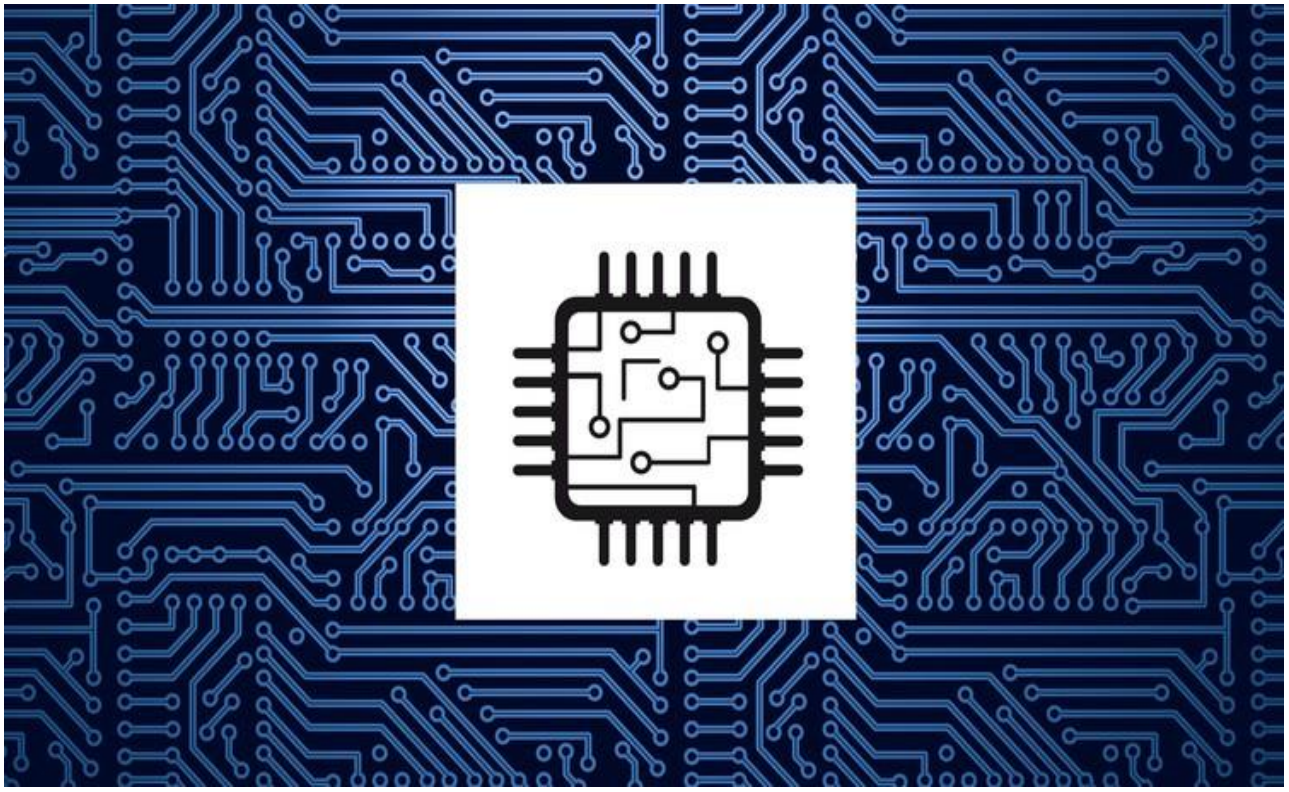
Ψηφιακά Συστήματα HW II

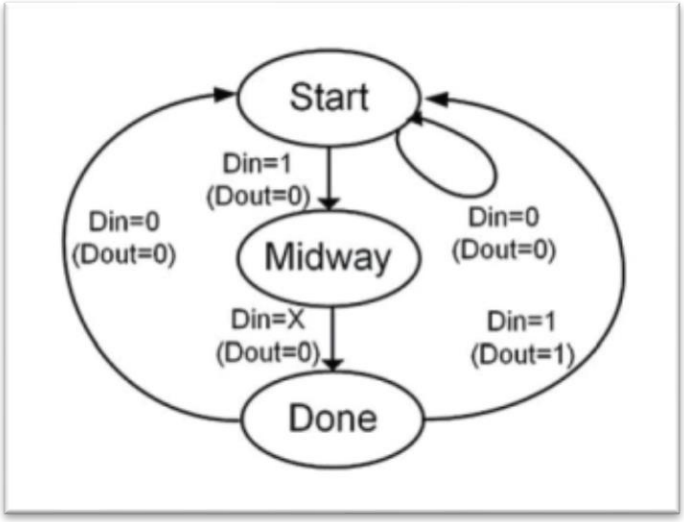
Αναφορά Εργασίας

- Όνομα: **Στασινός Αλκιβιάδης**
- ΑΕΜ: **9214**
- Email: **astasinos@ece.auth.gr**



Άσκηση 1η

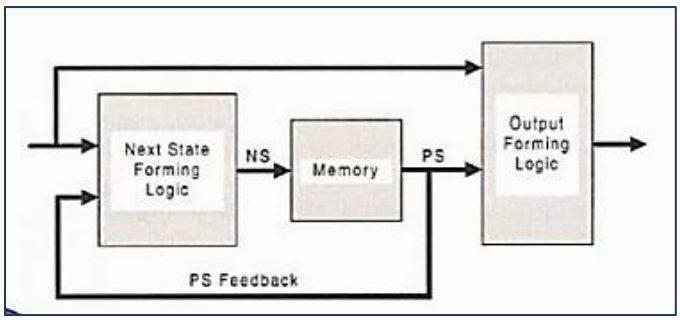
Στη πρώτη άσκηση μας ζητείται η σχεδίαση σε **Verilog** ενός μοντέλου συμπεριφοράς που υλοποιεί το **FSM** του Σχήματος 1, καθώς και του αντίστοιχου **testbench** για την επαλήθευση και διασφάλιση της λειτουργίας του. Επιπλέον η κωδικοποίηση των καταστάσεων του FSM θα πρέπει να γίνει με δυο διαφορετικούς τρόπους και η υλοποίηση του κώδικα Verilog σύμφωνα με τον ορισμό των θυρών που δίνεται στην εκφώνηση.



Σχήμα 1. Διάγραμμα μεταβάσεων για την άσκηση 1.

Από το Σχήμα 1 παρατηρούμε πως όταν το σύστημα μας αρχικοποιείται, ξεκινά από την κατάσταση **Start**. Από αυτή εάν η είσοδος μας **Din** είναι **1** οδηγούμαστε στη κατάσταση **Midway**, ειδάλλως παραμένουμε στη κατάσταση **Start**, ενώ και στις δυο περιπτώσεις η έξοδος μας **Dout** είναι **0**. Αν βρεθούμε στη κατάσταση **Midway** τότε ανεξαρτήτως εισόδου (**Din = X** / **don't care**) θα οδηγηθούμε στη κατάσταση **Done** με την έξοδο μας πάλι στο **0**. Τέλος στη κατάσταση **Done** εάν η είσοδος μας είναι **1** τότε η έξοδος μας γίνεται **1** και μετά επιστρέφουμε στη κατάσταση **Start**, αλλιώς αν η είσοδος μας είναι **0** τότε επιστρέφουμε πάλι στη **Start** αλλά η έξοδος μας παραμένει στο **0.**

Όπως μπορούμε να αντιληφθούμε πρόκειται για μια μηχανή πεπερασμένης κατάστασης τύπου **Mealy.** Αυτό το διαπιστώνουμε από το γεγονός ότι η έξοδος του FSM μας **Dout** εξαρτάται από την τρέχουσα κατάσταση αλλά **και** από την είσοδο μας **Din**. Όταν βρεθούμε στη κατάσταση **Done** η έξοδος εξαρτάται από την είσοδο **Din**.



Σχήμα 2. Mealy Finite State Machine

α) Υλοποίηση με απλή δυαδική κωδικοποίηση των καταστάσεων.

1. Κωδικοποίηση Καταστάσεων.

Το **FSM** μας έχει **τρείς** διακριτές καταστάσεις. Επομένως χρησιμοποιώντας την απλή δυαδική κωδικοποίηση θα χρειαστούμε **2 bit** δηλαδή 2 **FFs** (έστω **D-FFs**) για να περιγράψουμε όλες τις πιθανές καταστάσεις. Εφόσον θα έχουμε δύο **Flip-Flops** μπορούμε να κωδικοποιήσουμε έως και 4 καταστάσεις, άρα έχουμε μία παραπάνω κατάσταση που δε χρησιμοποιείται στη μηχανή αυτή. Στον Πίνακα 1 παρατίθενται όλες οι πιθανές καταστάσεις του FSM χρησιμοποιώντας κωδικοποίηση $\mathbf{D_{1:0}}$.

Κατάσταση	Κωδικοποίηση $D_{1:0}$
Start	00
Midway	01
Done	10

Πίνακας 1. Απλή δυαδική κωδικοποίηση καταστάσεων.

2. Πίνακας Αληθείας.

Στον Πίνακα 2 φαίνεται ο **πίνακας αληθείας του FSM**. Πέρα από τις απαραίτητες πληροφορίες ο πίνακας περιέχει και την επόμενη κατάσταση για κάθε είσοδο καθώς και τη κωδικοποίηση της κάθε κατάστασης.

Τρέχουσα Κατάσταση	Κωδικοποίηση D1	D0	Είσοδος *Din*	Επόμενη Κατάσταση	Κωδικοποίηση D1	D0	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**	**1**	**Start**	**0**	**0**	**1**
Done	**1**	**0**	**0**	**Start**	**0**	**0**	**0**

Πίνακας 2. Πίνακας Αληθείας FSM Σχήματος 1.

3. Λογικές εξισώσεις εξόδου FSM.

Από τον Πίνακα 3 μπορούμε εύκολα να εξάγουμε τη λογική εξίσωση που περιγράφει την έξοδο **Dout**.

Βάσει αυτού είναι

$$\boldsymbol{Dout = D1 * Din}$$

Τρέχουσα Κατάσταση D_1 D_0	Είσοδος *Din*	Έξοδος *Dout*
0 0	1	0
0 0	0	0
0 1	X	0
1 0	1	1
1 0	0	0

Πίνακας 3.

4. Κώδικας Verilog για το FSM.

Αρχικά δημιουργήθηκε ένα νέο project στο ModelSim και προστέθηκαν σε αυτό δύο αρχεία με όνομα fsml_behavioral.v και fsml_behavioral_TB.v για το **testbench** . Στο αρχείο **fsml_behavioral.v** ορίστηκε το **module** fsml_behavioral με τον κατάλληλο ορισμό θυρών όπως ακριβώς δίνεται στην εκφώνηση και φαίνεται στην Εικόνα 1.

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

Εικόνα 1.

Απαραίτητα Σήματα και Μεταβλητές.

Στη συνέχεια ορίζουμε δύο σήματα τύπου **reg [1:0]** (δηλ. 2 bit) τα οποία μοντελοποιούν την παρούσα και την επόμενη κατάσταση. Τα ονόματα των σημάτων αυτών είναι ***current_state*** και ***next_state*** αντίστοιχα. Στη πραγματικότητα οι καταστάσεις της μηχανής μοντελοποιούνται μέσω **Flip-Flops** που αποθηκεύουν την κατάσταση της μηχανής και όπως είδαμε προηγουμένως συγκεκριμένα σε αυτή τη περίπτωση χρειαζόμαστε **2 FFs** (2 bit). Έτσι το σήμα ***current_state*** περιγράφει ουσιαστικά τις εξόδους αυτών των FFs. Από την άλλη το ***next_state*** περιγράφει τις εισόδους αυτών. Επιπλέον είναι φυσικά πιο ευανάγνωστο και εύκολο να προσδώσουμε μέσα στον κώδικα ένα όνομα για κάθε κατάσταση παρά να χρησιμοποιούμε τη κωδικοποίηση της κάθε φορά. Για το λόγο αυτό ορίζονται παράμετροι για κάθε κατάσταση της μηχανής, με κάθε μία να αντιστοιχεί στη κατάλληλη κωδικοποίηση. Οι προαναφερθέντες ορισμοί φαίνονται στην Εικόνα 2.

```
reg [1:0] current_state, next_state;
parameter Start = 2'b00, Midway = 2'b01, Done = 2'b10;
```

Εικόνα 2. Απαραίτητα σήματα και μεταβλητές.

Κύριο μέρος - Περιγραφή της μηχανής.

Για την πλήρη περιγραφή του **FSM** χρησιμοποιήθηκαν **3 procedural blocks**.

1. Μπλόκ **Αποθήκευσης της κατάστασης** - **STATE_MEMORY**

   Το μπλόκ αυτό μοντελοποιεί την **ακολουθιακή λογική** υπεύθυνη για την ανάθεση της σωστής τιμής στο ***current_state***. Ορίζεται ως ***always*** μπλόκ και στο **sensitivity list** του περιλαμβάνει τα σήματα ***posedge Clock*** και ***negedge Reset***. Με τον τρόπο αυτό έχουμε επιλέξει την ενεργοποίηση του μπλόκ και την ενημέρωση του FSM σε κάθε **ανερχόμενη** ακμή του ρολογιού. Επιπροσθέτως, αφού το σήμα ***Reset*** περιλαμβάνεται στο sensitivity list καταλαβαίνουμε πως πρόκειται για ένα **ασύγχρονο** σήμα επαναφοράς το οποίο όπως θα φανεί και παρακάτω είναι **ενεργά χαμηλό** (active low).

```
always @ (posedge Clock or negedge Reset)
        begin: STATE_MEMORY
                if(!Reset)
                        current_state <= Start;
                else
                        current_state <= next_state;
        end
```

Εικόνα 3. Μπλόκ ακολουθιακής λογικής αποθήκευσης κατάστασης.

   Όπως φαίνεται και στην Εικόνα 3 όσο το σήμα ***Reset*** είναι ***low*** (δηλ. στο **0**) τότε **αν** βρισκόμαστε **ήδη** στη κατάσταση **Start** θα παραμείνουμε σε αυτή, **ειδάλλως** αν βρισκόμαστε σε άλλη κατάσταση τότε στην επόμενη ανερχόμενη ακμή του ρολογιού το σήμα ***current_state*** θα πάρει την τιμή **Start** (θα μεταβούμε δηλαδή στη **Start**).
   Όταν το ***Reset*** αποκτήσει την τιμή **1** και για όσο παραμείνει ***high*** τότε σε κάθε ανερχόμενη ακμή του ρολογιού το ***current_state*** θα τίθεται ίσο με την τιμή του σήματος ***next_state***. Η τιμή αυτή εξαρτάται και παράγεται από το **2ο** μπλόκ που περιγράφεται παρακάτω.

2. Μπλόκ **Επόμενης Κατάστασης** – **NEXT_STATE_LOGIC**

   Το μπλόκ αυτό μοντελοποιεί την **συνδυαστική** λογική μέσω της οποίας θα λάβει τιμή το σήμα ***next_state*** άρα τελικά και το ***current_state***. Ορίζεται ως ***always*** μπλόκ και στη ***sensitivity list*** του περιλαμβάνεται το σήμα ***current_state*** και το σήμα εισόδου ***Din***. Έτσι κάθε φορά που η παρούσα κατάσταση αλλάζει ή η είσοδος μας μεταβάλλεται το μπλόκ θα ενημερώνει τη τιμή του ***next_state***.
   Για τον προσδιορισμό της επόμενης κατάστασης χρησιμοποιείται μια εντολή **case** η οποία ανάλογα την παρούσα κατάσταση και την είσοδο μας θα δώσει την κατάλληλη τιμή στο ***next_state***. Το μπλόκ φαίνεται στην Εικόνα 4.

```
always @ (current_state or Din)
        begin: NEXT_STATE_LOGIC
                case(current_state)
                        Start   :  if(Din == 1'b1)
                                        next_state = Midway;
                                   else
                                        next_state = Start;
                        Midway  :  next_state = Done;
                        Done    :  next_state = Start;
                        default :  next_state = Start;
                endcase
        end
```

Εικόνα 4. Μπλόκ συνδυαστικής λογικής υπολογισμού επόμενης κατάστασης.

3. Μπλόκ **Εξόδου** – **OUTPUT_LOGIC**

Το μπλόκ αυτό μοντελοποιεί την **συνδυαστική** λογική μέσω της οποίας θα λάβει τιμή το σήμα εξόδου ***Dout***. Το FSM μας είναι τύπου **Mealy** επομένως στο sensitivity list πέρα από το σήμα ***current_state*** περιλαμβάνεται και η είσοδος ***Din***.

Όπως και τα προηγούμενα ορίζεται ως ***always*** μπλόκ και μέσω της εντολής **case** δίνεται η κατάλληλη τιμή στην έξοδο ανάλογα την κατάσταση στην οποία βρισκόμαστε και την τρέχουσα είσοδο. Το μπλόκ απεικονίζεται στην Εικόνα 5.

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

Εικόνα 5. Μπλόκ συνδυαστικής λογικής υπολογισμού εξόδου.

Συνολική Περιγραφή της Μηχανής.

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

        reg [1:0] current_state, next_state;
        parameter Start = 2'b00, Midway = 2'b01, Done = 2'b10;
```

```
always @ (posedge Clock or negedge Reset)
        begin: STATE_MEMORY
                if(!Reset)
                        current_state <= Start;
                else
                        current_state <= next_state;
        end
```

Τα μπλόκ που παρατίθενται στη σελίδα αυτή περιγράφουν στο σύνολο τους το FSM της άσκησης 1.

```
always @ (current_state or Din)
        begin: NEXT_STATE_LOGIC
                case(current_state)
                        Start  :  if(Din == 1'b1)
                                        next_state = Midway;
                                  else
                                        next_state = Start;
                        Midway  :  next_state = Done;
                        Done    :  next_state = Start;
                        default :  next_state = Start;
                endcase
        end
```

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

5. Κώδικας Verilog για το Testbench του FSM.

Για την επαλήθευση της λειτουργίας του FSM δημιουργήθηκε το κατάλληλο **testbench**. Το testbench αυτό με όνομα **fsml_behavioral_TB.v** παράγει τα διανύσματα εισόδου για το προς επαλήθευση κύκλωμα. Το προς επαλήθευση κύκλωμα συχνά ονομάζεται **DUT** (Design Under Test) και ορίζεται σαν ένα module στο εσωτερικό του testbench. Σκοπός του testbench είναι να ελέγξει εάν οι έξοδοι του DUT είναι οι σωστές για όλες τις πιθανές εισόδους.

Αρχικά ορίζεται το **timescale** για το testbench σε **1ns/1ns** δηλαδή σε κλίμακα 1 nanosecond με ακρίβεια 1 nanosecond. Ο τύπος **reg** χρησιμοποιείται για τις εισόδους ***Clock_TB Reset_TB Din_TB*** ώστε να "οδηγήσει" τις αντίστοιχες εισόδους του DUT που έχουν οριστεί σαν **wire** σε αυτό (Clock Reset Din), ενώ ο τύπος **wire** χρησιμοποιείται για την έξοδο ***Dout_TB*** που θα "οδηγηθεί" από την αντίστοιχη έξοδο του DUT που ορίστηκε σαν **reg** (Dout). Επιπλέον η αντιστοίχιση των θυρών στον ορισμό του DUT έγινε **χωροταξικά**. Τα παραπάνω φαίνονται στην Εικόνα 6.

```
`timescale 1ns/1ns

module fsml_behavioral_TB();

        wire Dout_TB;
        reg Clock_TB, Reset_TB, Din_TB;

        fsml_behavioral DUT(Dout_TB, Clock_TB, Reset_TB, Din_TB);
```

Εικόνα 6. Ορισμός του Design Under Test (DUT).

Στη συνέχεια χρησιμοποιείται ένα ***initial*** μπλόκ για να οδηγήσει το σήμα ***Reset***. Ένα μπλόκ αυτού του τύπου θα εκτελεστεί **μία** μόνο φορά και στο εσωτερικό του το ***Reset*** αρχικά τίθεται σε **0** (δηλαδή είναι ενεργό αφού είναι active-low) και μετά από μια καθυστέρηση 25 ns τίθεται στο **1**. Η καθυστέρηση μοντελοποιείται με το **#25** το οποίο αποτελεί μια μη συνθέσιμη εντολή γι'αυτό χρησιμοποιείται μόνο στο testbench.

```
initial
        begin
                Reset_TB = 1'b0;
            #25 Reset_TB = 1'b1;
end
```

Εικόνα 7. Μοντελοποίηση του σήματος Reset.

Για τη μοντελοποίηση του ***Clock*** χρησιμοποιήθηκε ένα ***initial*** και ένα ***always*** μπλόκ. Στο initial μπλόκ που εκτελείται μία φορά η τιμή του σήματος αρχικοποιείται σε **0** και στο always μπλόκ το σήμα θα παίρνει κάθε φορά μετά από μία καθυστέρηση 10 ns την αντίθετη τιμή από την τιμή που έχει εκείνη τη στιγμή. Έτσι έχουμε δημιουργήσει ένα ρολόι με περίοδο **20 ns** αφού βρίσκεται σε κάθε κατάσταση (high ή low) 10 ns.

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

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

Εικόνα 8. Μοντελοποίηση σήματος ρολογιού.

Τέλος το σήμα εισόδου μοντελοποιήθηκε με ένα ***initial*** μπλόκ στο οποίο του δόθηκαν οι κατάλληλες τιμές ανά τα κατάλληλα χρονικά διαστήματα ώστε κατά την προσομοίωση του testbench να μπορέσουμε να δούμε όλες τις πιθανές περιπτώσεις για το FSM μας.

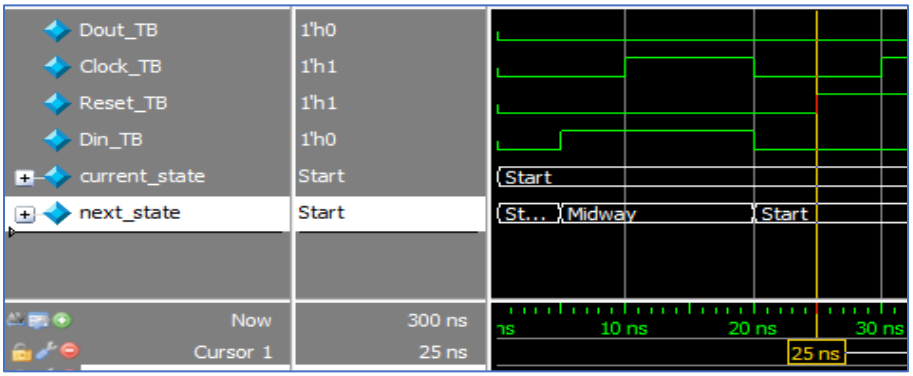
```
initial
        begin
                Din_TB = 1'b0;
          #5    Din_TB = 1'b1;
          #15   Din_TB = 1'b0;
          #35   Din_TB = 1'b1;
          #10   Din_TB = 1'b0;
          #25   Din_TB = 1'b1;
          #55   Din_TB = 1'b0;
          #45   Din_TB = 1'b1;
          #20   Din_TB = 1'b0;

end
```

Εικόνα 9. Μοντελοποίηση σήματος εισόδου.

6. Προσομοίωση και επαλήθευση.

Στο σημείο αυτό πραγματοποιήθηκε προσομοίωση του testbench που περιγράφεται πιο πάνω. Παρακάτω παρατίθενται στιγμιότυπα οθόνης από τις κυματομορφές που προκύπτουν βήμα-βήμα σύμφωνα με τη ροή της προσομοίωσης. Τα σήματα που φαίνονται είναι αυτά που ορίστηκαν στο testbench και με το **port mapping** που γίνεται στην αρχή, αντιστοιχούν στις κατάλληλες εισόδους και εξόδους του DUT.

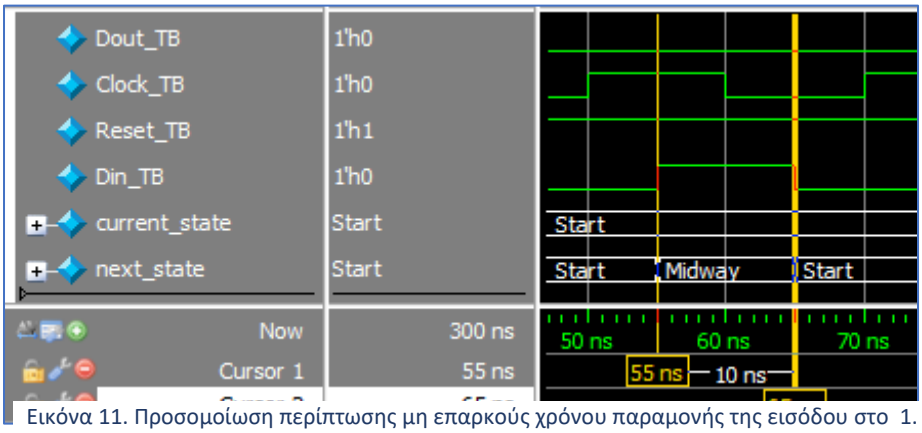
Αρχικά παρατηρούμε πώς το σήμα ***Reset*** παραμένει στο λογικό **0** από την αρχή του χρόνου έως τα **25 ns** όπου γίνεται **1** όπως ακριβώς έχει οριστεί στο testbench. Από **0** έως **25 ns** δηλαδή όσο το ***Reset*** είναι **low** το FSM επαναφέρεται και παραμένει στην κατάσταση **Start** ανεξαρτήτως εισόδου. Η είσοδος ***Din*** προκαλεί την ενεργοποίηση του μπλόκ **NEXT_STATE_LOGIC** αλλά στον κύκλο ρολογιού



Εικόνα 10. Επαλήθευση λειτουργίας σήματος Reset.

που έρχεται εντός των **25 ns** αυτών, το σήμα ***current_state*** δε θα επηρεαστεί αφού το σήμα επαναφοράς μας είναι ενεργοποιημένο και έτσι θα παραμείνουμε σίγουρα στη κατάσταση **Start**.

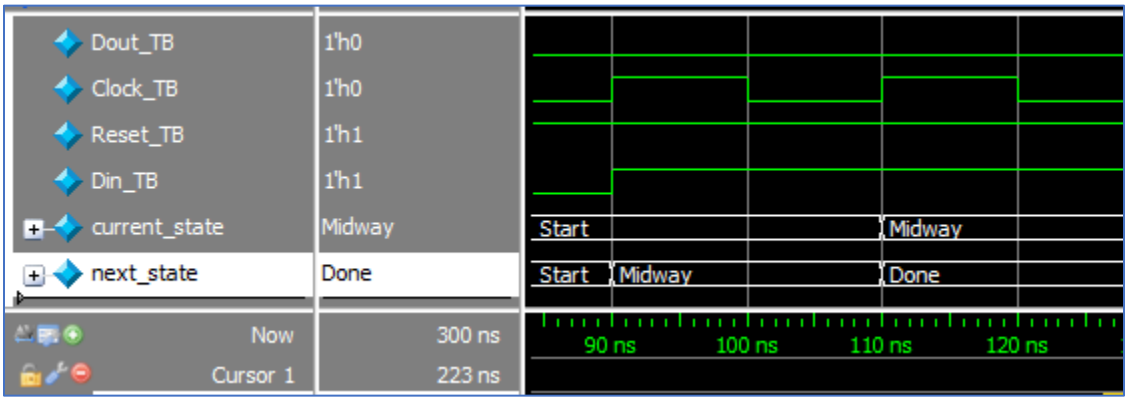
Στη συνέχεια και αφού το σήμα επαναφοράς μας (Reset) έχει απενεργοποιηθεί, κατά το διάστημα **55 ns** – **65 ns** η είσοδος ***Din*** γίνεται **1**. Το σήμα εισόδου ***Din*** περιλαμβάνεται στο **sensitivity list** των μπλόκ **OUTPUT_LOGIC** και **NEXT_STATE_LOGIC** . Στο μπλόκ υπεύθυνο για τη λογική εξόδου η αλλαγή του ***Din*** δεν έχει καμία επίδραση αφού αν δε βρισκόμαστε στη κατάσταση **Done** η έξοδος ανεξαρτήτως εισόδου είναι **0**. Το μπλόκ υπεύθυνο για την επόμενη κατάσταση από την άλλη, μόλις άλλαξε η ***Din*** σε **1** αμέσως υπολόγισε την τιμή του σήματος ***next_state*** σε **Midway**. Όμως το σήμα ***current_state*** λαμβάνει τη τιμή του ***next_state*** σε **κάθε ανερχόμενη ακμή του ρολογιού** που όπως βλέπουμε **δε πρόλαβε να έρθει πριν** η είσοδος ***Din*** πέσει πάλι στο **0**. Όταν πέσει στο **0** η είσοδος , το μπλόκ **NEXT_STATE_LOGIC** θα υπολογίσει πάλι το ***next_state*** σε **Start** και στον επόμενη ανερχόμενη ακμή που θα έρθει, η παρούσα κατάσταση δεν επηρεάζεται και σωστά παραμένει σε **Start**.



Εικόνα 11. Προσομοίωση περίπτωσης μη επαρκούς χρόνου παραμονής της εισόδου στο 1.

Ύστερα η είσοδος μας επανέρχεται στο **1** τη χρονική στιγμή **90 ns** και παραμένει εκεί έως τη **145 ns**.

Μόλις γίνει **1** άμεσα το **NEXT_STATE_LOGIC** μπλόκ σωστά υπολογίζει πως η επόμενη κατάσταση θα πρέπει να είναι η **Midway** αφού από **Start** (Din = 1) → **Midway**. Το σήμα ***current_state*** θα πάρει τη τιμή **Midway** στην επόμενη ανερχόμενη ακμή του ρολογιού όπως σωστά απεικονίζεται στη χρονική στιγμή **110 ns** . Εφόσον το σήμα αυτό περιέχεται στη sensitivity list του **NEXT_STATE_LOGIC** άμεσα το μπλόκ αυτό θα υπολογίσει πως η επόμενη κατάσταση θα είναι η **Done** . Από τη **Midway** η επόμενη κατάσταση είναι πάντα η **Done** ανεξαρτήτως εισόδου. Στην Εικόνα 12 φαίνεται το χρονικό διάστημα **90 ns** με **120 ns**.



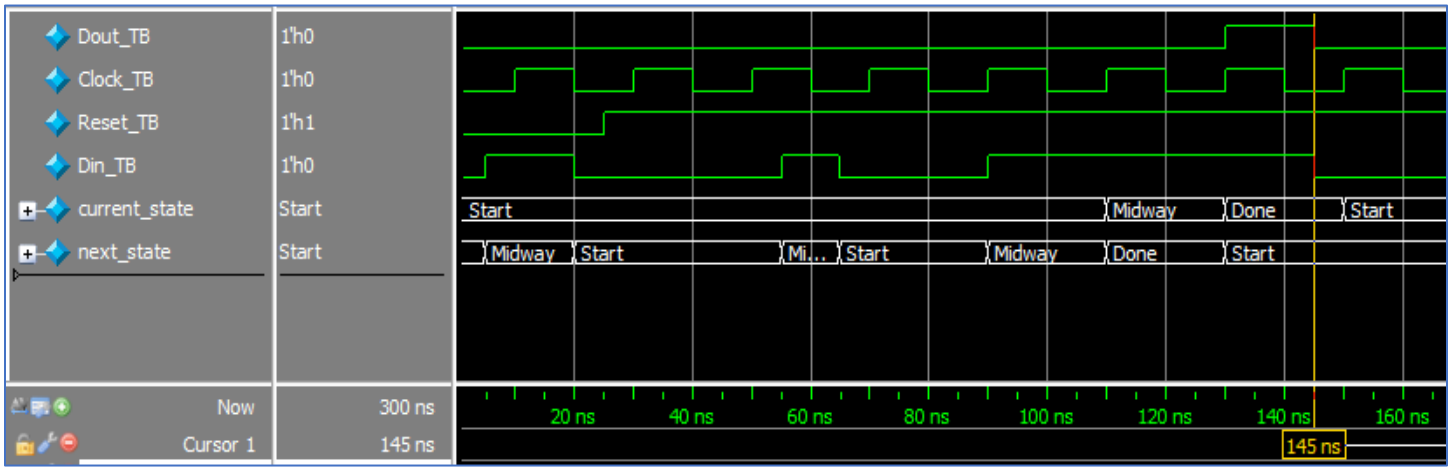
Εικόνα 12.

Στα **130 ns** δηλαδή στην επόμενη ανερχόμενη ακμή του ρολογιού το σήμα ***current_state*** παίρνει τη τιμή **Done**. Επιπλέον γνωρίζουμε ότι το ***current_state*** και η έισοδος ***Din*** βρίσκονται στη sensitivity list του μπλόκ **OUTPUT_LOGIC .** Άρα αφού μεταβήκαμε στη κατάσταση **Done** πυροδοτείται το μπλόκ αυτό ενώ παράλληλα είναι και ***Din* = 1**. Έτσι η έξοδος γίνεται ***Dout = 1*** όπως αναμένοταν και περιγράφεται και στον κώδικα Verilog.

Την ίδια στιγμή λόγω της μετάβασης της παρούσας κατάστασης ενεργοποιείται και το μπλόκ **NEXT_STATE_LOGIC** και υπολογίζεται η νέα κατάσταση δηλαδή η **Start**.

Η έξοδος παραμένει στο **1** όσο η είσοδος ***Din*** παραμένει στο **1** ή δεν έχει έρθει ο επόμενος κύκλος ρολογιού ώστε το ***current_state*** να μεταβεί σε **Start**.

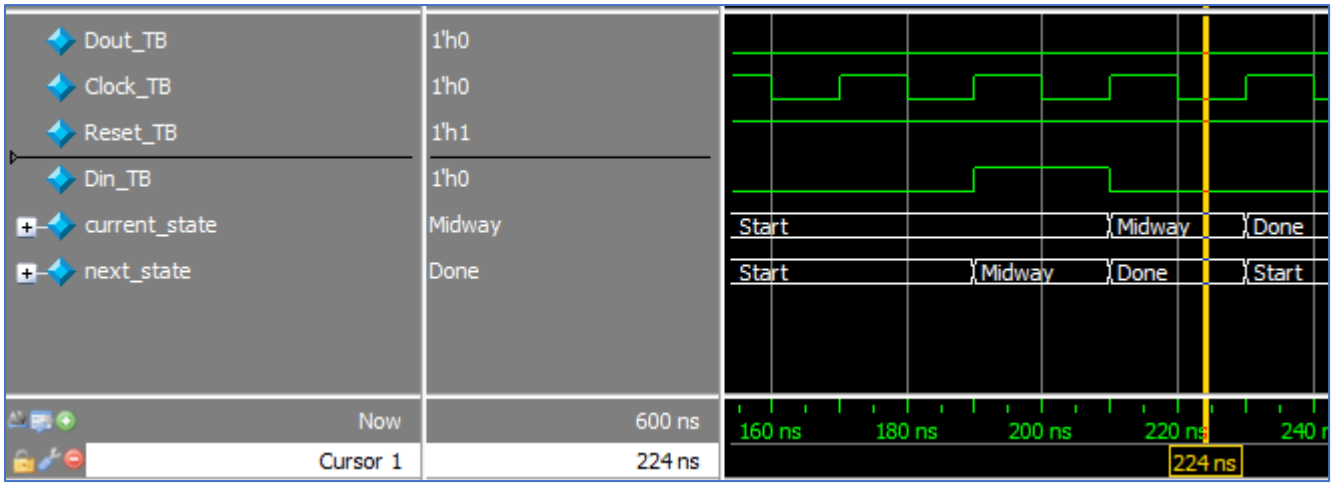
Στη περίπτωση μας η είσοδος ***Din*** μεταβαίνει στο **0** πριν την επόμενη ανερχόμενη ακμή του ρολογιού και συγκεκριμένα τη στιγμή **145 ns** . Λόγω αυτού την ίδια στιγμή και η έξοδος μας μηδενίζεται. Ύστερα τη στιγμή **150 ns** έρχεται η ανερχόμενη ακμή του ρολογιού και το FSM επιστρέφει στη κατάσταση **Start**. Όλα τα παραπάνω φαίνονται στην Εικόνα 13.



Εικόνα 13.

190 ns μετά την αρχή της προσομοίωσης η είσοδος ***Din*** γίνεται και πάλι **1**. Έτσι αφού άλλαξε το ***Din*** ενεργοποιείται το μπλόκ **NEXT_STATE_LOGIC** και υπολογίζεται η τιμή του ***next_state*** σε **Midway**. Σωστά αφού βρισκόμασταν στη κατάσταση **Start**.

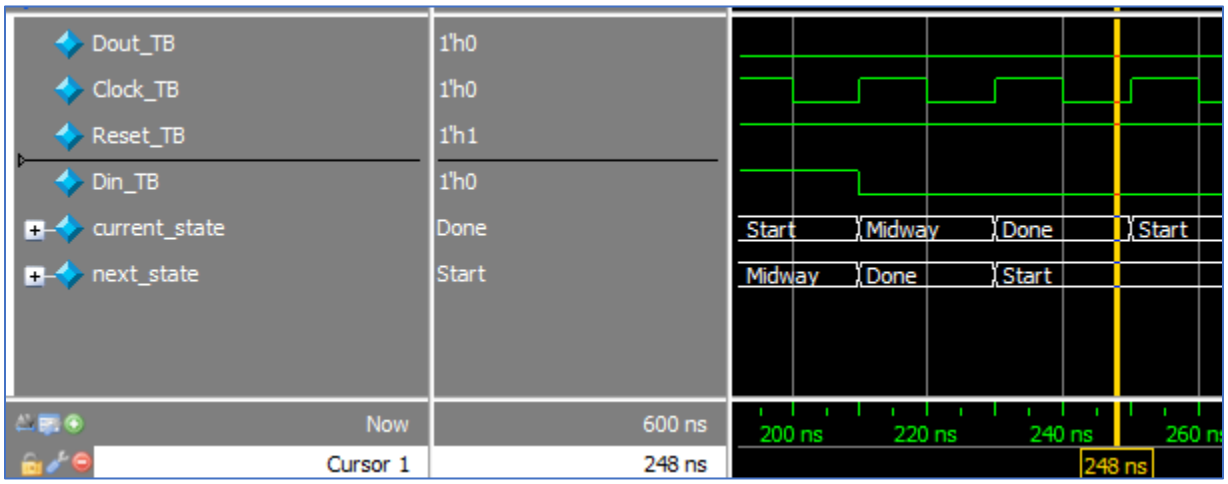
Στην επόμενη ανερχόμενη ακμή του ρολογιού,δηλαδή στα **210 ns**, η τιμή του ***next_state*** εκχωρείται στο ***current_state*** και μεταβαίνουμε στη κατάσταση **Midway**. Εφόσον το σήμα ***current_state*** άλλαξε, ενεργοποιείται πάλι το μπλόκ υπεύθυνο για την επόμενη κατάσταση και το σήμα ***next_state*** παίρνει τη τιμή **Done**. Παράλληλα την ίδια στιγμή λόγω των καθυστερήσεων που έχουμε ορίσει στο testbench τυχαίνει και η είσοδος ***Din*** να πέφτει στο **0**.



Εικόνα 14.

Στην επόμενη ανερχόμενη ακμή του ***Clock*** βλέπουμε πως το σήμα ***current_state*** λαμβάνει τη τιμή που είχε το ***next_state*** και μεταβαίνουμε στη κατάσταση **Done**. Τη στιγμή που συμβαίνει αυτό η είσοδος ***Din*** είναι στο **0** σε αντίθεση με τη προηγούμενη φορά που έγινε η μετάβαση **Midway → Done** στα **130 ns**. Έτσι αντιλαμβανόμαστε ότι πράγματι ανεξαρτήτως εισόδου (***Din = X / Don't care***) όταν βρισκόμαστε στη κατάσταση **Midway** στην επόμενη ανερχόμενη ακμή θα βρεθούμε στη **Done**. Όπως ακριβώς έχει οριστεί από τις προδιαγραφές του FSM.

Επιπλέον όπως γνωρίζουμε το μπλόκ **OUTPUT_LOGIC** περιλαμβάνει στη sensitivity list του τα σήματα ***Din*** και ***current_state*** και ενεργοποιείται όταν συμβαίνει οποιαδήποτε αλλαγή σε αυτά. Σε περίπτωση όμως που δε βρισκόμαστε στη κατάσταση **Done** η έξοδος θα είναι by default **0** ενώ αν βρισκόμαστε σε αυτή, η έξοδος θα είναι **1** μόνο αν και η είσοδος είναι **1**. Τη χρονική στιγμή **230 ns** θα βρεθούμε στη κατάσταση **Done** με την είσοδο ***Din*** ωστόσο να βρίσκεται στο **0**. Έτσι το σήμα εξόδου ***Dout*** δε θα γίνει **1** και θα παραμείνει στο **0** ενώ στη συνέχεια θα επανέλθουμε στη κατάσταση **Start**. Τα παραπάνω φαίνονται στην Εικόνα 15.



Εικόνα 15.

Από τη στιγμή **250 ns** και μετά η προσομοίωση έχει λάβει τέλος. Η λειτουργία του FSM έχει επαληθευτεί πλήρως, όπως ακριβώς υποδεικνύεται και από το διάγραμμα καταστάσεων της εκφώνησης.

β) Υλοποίηση με κωδικοποίηση One-hot.

Στο δεύτερο μέρος της άσκησης 1 ζητείται η υλοποίηση του ίδιου FSM με κωδικοποίηση One-hot. Αυτό σημαίνει πώς κάθε κατάσταση κωδικοποιείται με τέτοιο τρόπο ώστε **ένα** και μόνο bit θα βρίσκεται στο **1** ενώ τα υπόλοιπα στο **0**.

1. Κωδικοποίηση Καταστάσεων.

Οι τρείς πιθανές καταστάσεις του FSM **Start, Midway** και **Done** κωδικοποιούνται με One-hot κωδικοποίηση όπως φαίνεται στον Πίνακα 4.

Κατάσταση	Κωδικοποίηση $D_{2:0}$
Start	001
Midway	010
Done	100

Πίνακας 4.

2. Πίνακας Αληθείας.

Ο Πίνακας 5 είναι ο πίνακας αλήθειας του FSM σύμφωνα με την κωδικοποίηση One-hot.

Τρέχουσα Κατάσταση	Κωδικοποίηση D2	D1	D0	Είσοδος *Din*	Επόμενη Κατάσταση	Κωδικοποίηση D2	D1	D0	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**	**1**	**Start**	**0**	**0**	**1**	**1**
Done	**1**	**0**	**0**	**0**	**Start**	**0**	**0**	**1**	**0**

Πίνακας 5.

3. Λογικές εξισώσεις εξόδου FSM.

Βάσει του Πίνακα 6 μπορούμε εύκολα να εξάγουμε τη λογική εξίσωση εξόδου

$$\boldsymbol{Dout = D2 * Din}$$

Τρέχουσα Κατάσταση 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**	**1**	**1**
1	**0**	**0**	**0**	**0**

Πίνακας 6.

4. Κώδικας Verilog για το FSM.

Για την υλοποίηση του FSM με κωδικοποίηση One-hot δημιουργήθηκε ένα νέο αρχείο **fsml_behavioral_onehot.v** το οποίο περιείχε τον κώδικα που είχε και το αντίστοιχο αρχείο για την απλή δυαδική κωδικοποίηση με πολύ λίγες αλλαγές. Συγκεκριμένα η μόνη αλλαγή ήταν στον ορισμό των καταστάσεων όπως φαίνεται στην Εικόνα 16. Τα σήματα ***current_state*** και ***next_state*** ορίζονται ως **reg [2:0]** αφού τώρα χρειαζόμαστε 3 bit για να κωδικοποιήσουμε τις καταστάσεις.

```
reg [2:0] current_state, next_state;
parameter Start = 3'b001, Midway = 3'b010, Done = 3'b100;
```

Εικόνα 16.

5. Κώδικας Verilog για το Testbench του FSM.

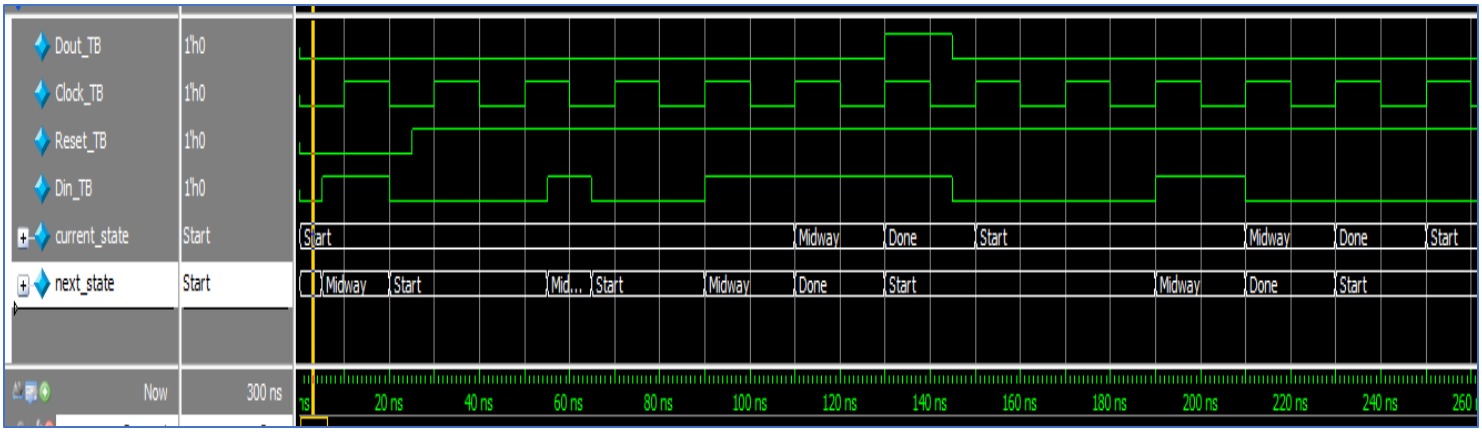
Για την επαλήθευση της σχεδίασης όταν χρησιμοποιούμε One-hot κωδικοποίηση χρησιμοποιήθηκε το ίδιο testbench με αυτό του ερωτήματος **α)** δηλαδή της απλής δυαδικής κωδικοποίησης. Η μόνη διαφορά ήταν στον ορισμό του **DUT** όπως φαίνεται στην Εικόνα 17.

```
fsml_behavioral_onehot DUT(Dout_TB, Clock_TB, Reset_TB, Din_TB);
```

Εικόνα 17.

6. Προσομοίωση και επαλήθευση.

Εφόσον χρησιμοποιήθηκε το ίδιο testbench και οι μόνες αλλαγές ήταν στον τρόπο κωδικοποίησης των καταστάσεων, ο τρόπος προσομοίωσης και επαλήθευσης είναι ο ίδιος με αυτόν που ακολουθήθηκε στο ερώτημα α) . Έτσι, όπως ακριβώς και στην απλή δυαδική κωδικοποίηση γνωρίζουμε πως το FSM μας έχει υλοποιηθεί σωστά. Παρατίθεται ενδεικτικά και η κυματομορφή της προσομοίωσης στην Εικόνα 18.



Εικόνα 18.

Άσκηση 2η

Στη δεύτερη άσκηση ζητείται η σχεδίαση ενός ελεγκτή φανού σήμανσης σε ένα δρόμο ταχείας κυκλοφορίας. Ο δρόμος αυτός τέμνεται κάθετα και από έναν μικρότερο δρόμο. Το FSM θα είναι υπεύθυνο για τον έλεγχο των σημάτων για το κόκκινο, πορτοκαλί και πράσινο του φαναριού στο δρόμο ταχείας κυκλοφορίας.

1. Διάγραμμα μεταβάσεων των καταστάσεων.

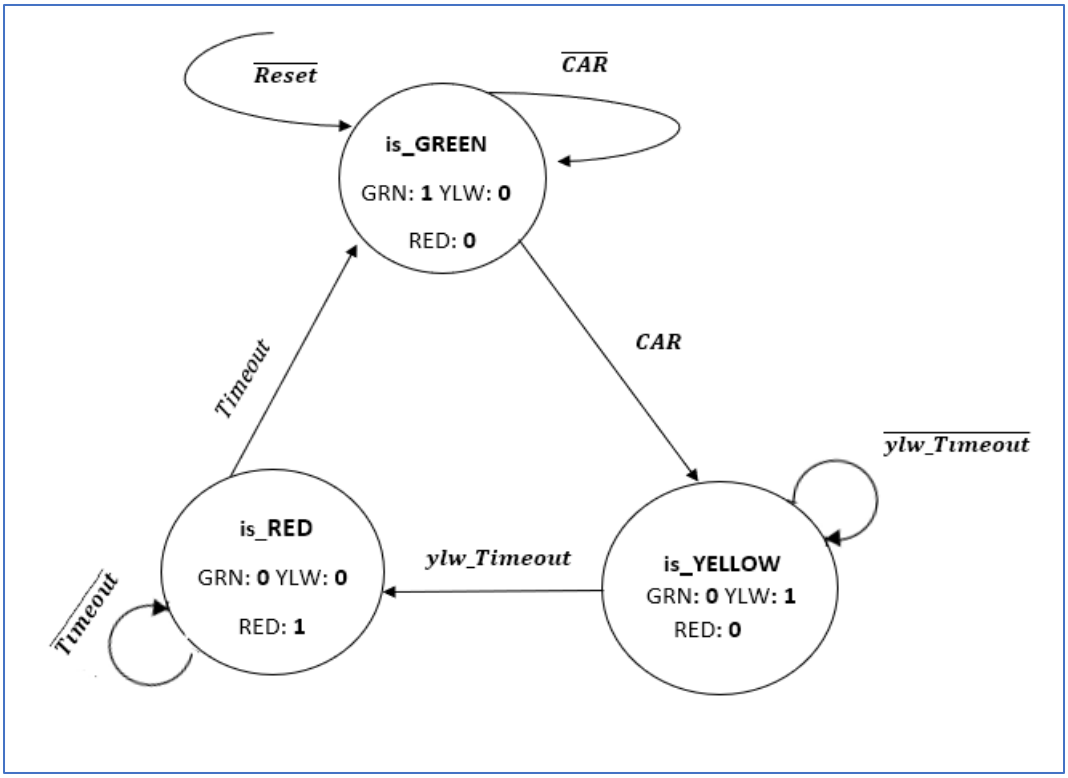
Στο πρώτο ερώτημα της άσκησης 2 ζητείται η κατασκευή του διαγράμματος καταστάσεων. Για την κωδικοποίηση των καταστάσεων χρησιμοποιήθηκε η απλή δυαδική κωδικοποίηση. Συγκεκριμένα ορίστηκαν οι καταστάσεις **is_GREEN = 2'b00** , **is_YELLOW = 2'b01** , **is_RED = 2'b10**.

Όπως ορίζεται και από την εκφώνηση, υπό κανονικές συνθήκες το φανάρι είναι πράσινο. Έτσι, επιλέχθηκε η κατάσταση **is_GREEN** ως αρχική κατάσταση του FSM ενώ σε αυτή θα γυρίσουμε και αν ενεργοποιηθεί το σήμα επαναφοράς (***Reset***).

Όσο το σήμα ***CAR*** είναι **0** , δηλαδή όσο δεν έχει εμφανιστεί όχημα στον κάθετο δρόμο τότε το φανάρι θα παραμένει πράσινο. Μόλις έρθει ένα σήμα ***CAR = 1*** τότε θα μεταβούμε στην επόμενη κατάσταση **is_YELLOW** και το φανάρι θα γίνει κίτρινο.

Μόλις το φανάρι γίνει κίτρινο τότε αν και δεν αναφέρεται στην εκφώνηση περιμένουμε **2** δευτερόλεπτα για να μεταβούμε στη κατάσταση **is_RED** και να γίνει κόκκινο. Αυτό γίνεται ώστε να προσομοιώσουμε όσο το δυνατόν πιο πραγματικές συνθήκες και όπως γνωρίζουμε η μετάβαση ενός φαναριού από κίτρινο σε κόκκινο δεν είναι ακαριαία. Όταν περάσουν τα 2 δευτερόλεπτα ένα σήμα με όνομα ***ylw_Timeout*** σηκώνεται από **0** σε **1** και μεταβαίνουμε στη κατάσταση όπου το φανάρι είναι κόκκινο.

Όταν το φανάρι γίνει κόκκινο τότε όπως υποδεικνύεται, μετά από μια καθυστέρηση 15 δευτερολέπτων παράγεται ένα **εσωτερικό** σήμα **TIMEOUT** και το FSM δηλαδή το φανάρι θα γίνεται και πάλι πράσινο (μετάβαση σε **is_GREEN**). Με βάση τα παραπάνω παρατίθεται στο Σχήμα 3 το διάγραμμα καταστάσεων του προς υλοποίηση FSM. Σε κάθε κατάσταση φαίνονται και οι έξοδοι του FSM σε εκείνο το state όπως αυτές δίνονται στην εκφώνηση.



Σχήμα 3. Διάγραμμα καταστάσεων FSM

2. Κωδικοποίηση των καταστάσεων.

Για την κωδικοποίηση των τριών πιθανών καταστάσεων χρησιμοποιήθηκε η απλή δυαδική κωδικοποίηση όπως φαίνεται στον Πίνακα 7.

Κατάσταση	Κωδικοποίηση $D_{1:0}$
is_GREEN	00
is_YELLOW	01
is_RED	10

Πίνακας 7. Απλή δυαδική κωδικοποίηση FSM ασκ.2.

3. Πίνακας Αλήθειας.

Ο Πίνακας 8 είναι ο πίνακας αλήθειας του FSM. Όπως γνωρίζουμε το FSM έχει ένα σήμα εισόδου , το σήμα ***CAR*** και τρείς εξόδους ***RED YLW GRN*** οι οποίες αντιστοιχούν στο χρώμα του φανού σήμανσης. Για παράδειγμα αν ***RED = 1*** και ***YLW = 0 GRN = 0*** τότε το φανάρι είναι κόκκινο. Ωστόσο το FSM μας είναι

τύπου **Moore** και επομένως οι έξοδοι δεν εξαρτώνται από το σήμα εισόδου γι' αυτό και η είσοδος ***CAR*** δε περιλαμβάνεται στον πίνακα αλήθειας.

Τρέχουσα Κατάσταση	Τρέχουσα Κατάσταση D1	D0	Έξοδος *GRN*	Έξοδος *YLW*	Έξοδος *RED*
is_GREEN	0	0	1	0	0
is_YELLOW	0	1	0	1	0
is_RED	1	0	0	0	1

Πίνακας 8.

4. Λογικές εξισώσεις εξόδου FSM.

Από τον Πίνακα 8 μπορούμε εύκολα να εξάγουμε τις εξισώσεις $\boldsymbol{GRN = \overline{D1} * \overline{D2}}$ $\boldsymbol{RED = D1 * \overline{D0}}$

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

5. Κώδικας Verilog για το FSM.

Αρχικά δημιουργήθηκε ένα νέο project στο ModelSim με όνομα **fanari** και προστέθηκαν δύο αρχεία, το **fanari.v** και το **fanari_TB.v** για το testbench. Στο αρχείο fanari.v δημιουργήθηκε το module fanari με τρείς εξόδους τύπου **reg** για τα **RED GRN YLW** και τρία σήματα εισόδου τύπου **wire** για τα **Clock Reset CAR** όπως φαίνεται στην Εικόνα 19.

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

Εικόνα 19.

Απαραίτητα Σήματα και Μεταβλητές.

Στη συνέχεια ορίζουμε δύο σήματα τύπου **reg [1:0]** (δηλ. 2 bit) τα οποία μοντελοποιούν την παρούσα και την επόμενη κατάσταση. Τα ονόματα των σημάτων αυτών είναι ***current_state*** και ***next_state*** αντίστοιχα. Επιπλέον όπως και για το FSM της πρώτης άσκησης είναι πιο ευανάγνωστο αντί να χρησιμοποιούμε συνεχώς την κωδικοποίηση των καταστάσεων να προσδώσουμε ένα όνομα σε καθεμία. Έτσι στον κώδικα ορίζονται παράμετροι για κάθε κατάσταση με κάθε παράμετρο να αντιστοιχεί στην κατάλληλη κατάσταση.

Επιπροσθέτως ορίζονται δύο σήματα τύπου **reg** , ένα με 15 bit και όνομα **counter** το οποίο θα παίρνει ώς μέγιστη τιμή το 15000 (δυαδικό 11101010011000) και χρησιμεύει όπως θα δούμε στην υλοποίηση της

καθυστέρησης 15 δευτερολέπτων όταν το φανάρι γίνεται από κόκκινο πράσινο και άλλο ένα σήμα 12 bit με όνομα **ylw_counter** και μέγιστη τιμή το 2000 (δυαδικό 11111010000) που χρησιμεύει αντίστοιχα στην υλοποίηση της καθυστέρησης όταν το φανάρι γίνεται από κίτρινο κόκκινο.

Τέλος ορίζονται δύο σήματα τύπου **wire.** Το πρώτο είναι το σήμα ***Timeout*** όπως αναφέρεται και στην εκφώνηση. Με ανάθεση **assign** η τιμή του ***Timeout*** συνδέεται με το αποτέλεσμα της Boolean έκφρασης **(counter == 0)** η οποία είναι αληθής και παίρνει την τιμή **1** μόνο όταν η τιμή του counter γίνει **0**. Άρα τότε θα σηκωθεί στο λογικό **1** και το σήμα ***Timeout*** . Όταν ο counter θα επαναφερθεί στη τιμή 15000 το σήμα ***Timeout*** θα πέσει πάλι στο **0**. Η τιμή του counter αρχίζει και μειώνεται όταν το φανάρι γίνει κόκκινο και φτάνει στη τιμή **0** όταν παρέλθουν 15 δευτερόλεπτα, οπότε και το φανάρι γίνεται πράσινο. Αντίστοιχα με την ίδια λογική ορίζεται και το σήμα ***ylw_Timeout*** που θα σηκωθεί στο λογικό **1** όταν μηδενίσει ο **ylw_counter**. Τα προαναφερθέντα φαίνονται όλα στην Εικόνα 20.

```
reg [1:0] current_state, next_state;

parameter is_GREEN = 2'b00, is_YELLOW = 2'b01, is_RED = 2'b10;

reg [14:0] counter = 15000;
reg [11:0] ylw_counter = 2000;
wire Timeout, ylw_Timeout;

assign Timeout = (counter == 0);
assign ylw_Timeout = (ylw_counter == 0);
```

Εικόνα 20. Απαραίτητα σήματα για το FSM.

Κύριο μέρος - Περιγραφή της μηχανής.

Για την πλήρη περιγραφή του FSM χρησιμοποιήθηκαν 5 **procedural blocks**.

1. Μπλόκ **Αποθήκευσης της κατάστασης** - **STATE_MEMORY**

   Όπως και το πρώτο μπλόκ του FSM της πρώτης άσκησης που φαίνεται στην Εικόνα 3, το μπλόκ αυτό μοντελοποιεί την **ακολουθιακή λογική** υπεύθυνη για την ανάθεση της σωστής τιμής στο ***current_state***. Ορίζεται ως ***always*** μπλόκ και στο **sensitivity list** του περιλαμβάνει τα σήματα ***posedge Clock*** και ***negedge Reset***. Με τον τρόπο αυτό έχουμε επιλέξει την ενεργοποίηση του μπλόκ και την ενημέρωση του FSM σε κάθε **ανερχόμενη** ακμή του ρολογιού και ένα

```
always @ (posedge Clock or negedge Reset)
        begin: STATE_MEMORY
                if(!Reset)
                        current_state <= is_GREEN;
                else
                        current_state <= next_state;
        end
```

Εικόνα 21. . Μπλόκ ακολουθιακής λογικής αποθήκευσης κατάστασης FSM

ασύγχρονο ενεργά χαμηλό ***Reset***. Όπως φαίνεται και από το διάγραμμα μεταβάσεων κατάστασης όταν το σήμα επαναφοράς ***Reset*** είναι ενεργό επιστρέφουμε και παραμένουμε στην κατάσταση **is_GREEN** , δηλαδή το φανάρι είναι πράσινο.

2. Μπλόκ **Επόμενης Κατάστασης** – **NEXT_STATE_LOGIC**

Όπως και στο προηγούμενο FSM που υλοποιήθηκε, το μπλόκ αυτό μοντελοποιεί την **συνδυαστική** λογική μέσω της οποίας θα λάβει τιμή το σήμα ***next_state*** άρα τελικά και το ***current_state*** (παρούσα κατάσταση). Ορίζεται ως ***always*** μπλόκ και στη ***sensitivity list*** του περιλαμβάνεται το σήμα ***current_state***, το σήμα εισόδου ***CAR*** και τα σήματα ***Timeout*** , ***ylw_Timeout*** (μόνο σε posedge) . Αφού τα σήματα αυτά περιλαμβάνονται στη sensitivity list του μπλόκ, επιδρούν ασύγχρονα και ενεργοποιούν το μπλόκ κάθε φορά που κάποιο από αυτά αλλάζει. Έτσι κάθε φορά που η παρούσα κατάσταση αλλάζει ή η είσοδος μεταβάλλεται ή **σηκώνεται** κάποιο σήμα ***Timeout*** το μπλόκ θα ενημερώνει τη τιμή του ***next_state***.
Για τον προσδιορισμό της επόμενης κατάστασης χρησιμοποιείται μια εντολή **case** η οποία ανάλογα την παρούσα κατάσταση και τα υπόλοιπα σήματα της sensitivity list μας θα δώσει την κατάλληλη τιμή στο ***next_state***. Συγκεκριμένα, όταν το φανάρι είναι πράσινο αν η είσοδος ***CAR*** είναι **1** τότε η επόμενη κατάσταση είναι το φανάρι να γίνει κίτρινο ***next_state = is_YELLOW*** , ειδάλλως αν ***CAR = 0*** τότε παραμένουμε στην **is_GREEN**. Αν βρισκόμαστε στην κατάσταση όπου το φανάρι είναι κίτρινο τότε αν το σήμα **ylw_Timeout** γίνει **1** (όταν παρέλθουν 2 δευτερόλεπτα) η επόμενη κατάσταση είναι αυτή που το φανάρι είναι κόκκινο (**is_RED**) , αλλιώς παραμένουμε στην **is_YELLOW**. Τέλος αν βρισκόμαστε στην κατάσταση **is_RED** και έχουν παρέλθει 15 δευτερόλεπτα από τη στιγμή που το φανάρι άναψε κόκκινο, άρα και το σήμα ***Timeout*** γίνεται **1** , η επόμενη κατάσταση είναι η **is_GREEN** , αλλιώς παραμένουμε στην **is_RED**. Το μπλόκ φαίνεται στην Εικόνα 22.

```
always @ (current_state or CAR or posedge Timeout or posedge ylw_Timeout)
        begin: NEXT_STATE_LOGIC
                case(current_state)
                        is_GREEN  :
                                        if(CAR == 1'b1)
                                          next_state = is_YELLOW;
                                          else
                                          next_state = is_GREEN;
                        is_YELLOW  :  if(ylw_Timeout == 1'b1)
                                          next_state = is_RED;
                                        else
                                          next_state = is_YELLOW;
                        is_RED    :  if(Timeout == 1'b1)
                                          next_state = is_GREEN;
                                    else
                                          next_state = is_RED;
                        default :  next_state = is_GREEN;
                endcase
        end
```

Εικόνα 22.Μπλόκ συνδυαστικής λογικής υπολογισμού επόμενης κατάστασης.

3. Μπλόκ **Εξόδου** – **OUTPUT_LOGIC**

   Το μπλόκ αυτό μοντελοποιεί την **συνδυαστική** λογική μέσω της οποίας θα λάβουν τιμή τα σήματα εξόδου ***RED YLW GRN***. Το FSM μας είναι τύπου **Moore** επομένως στο sensitivity list πέρα από το σήμα ***current_state*** **δε** περιλαμβάνεται και η είσοδος ***CAR***.

   Όπως και τα προηγούμενα ορίζεται ως ***always*** μπλόκ και μέσω της εντολής **case** δίνεται η κατάλληλη τιμή στην έξοδο ανάλογα την κατάσταση στην οποία βρισκόμαστε. Το μπλόκ απεικονίζεται στην Εικόνα 23.

```
always @ (current_state)
        begin:  OUTPUT_LOGIC
                case(current_state)
                        is_GREEN  :  {RED,YLW,GRN} = 3'b001;
                        is_YELLOW :  {RED,YLW,GRN} = 3'b010;
                        is_RED    :  {RED,YLW,GRN} = 3'b100;
                        default:  {RED,YLW,GRN} = 3'b001;
                endcase
        end
```

Εικόνα 23. Μπλόκ συνδυαστικής λογικής υπολογισμού εξόδου.

4. Μπλόκ Απαριθμητή για το σήμα **TIMEOUT – RED_DOWN_COUNTER**

   Όπως ορίζεται στην εκφώνηση από τη στιγμή που το φανάρι γίνει κόκκινο τότε θα πρέπει να περιμένουμε 15 δευτερόλεπτα και ύστερα να γίνει πράσινο. Το μπλόκ αυτό εξυπηρετεί ακριβώς αυτό το σκοπό. Αρχικά ελέγχεται εάν βρισκόμαστε στην κατάσταση **is_RED** και όσο αυτό ισχύει τότε σε κάθε ανερχόμενη ακμή του ρολογιού η τιμή του **counter** που ορίστηκε στην αρχή του αρχείου μειώνεται κατά ένα. Ο **counter** αυτός ξεκίνησε από την τιμή 15000 και έτσι θα πάρει 15000 κύκλους να φτάσει στο 0. Στο testbench που θα δημιουργήσουμε οι ανερχόμενες ακμές του ρολογιού έρχονται κάθε **1 ms** επομένως ο counter θα φτάσει στο 0 σε 15 δευτερόλεπτα. Μόλις φτάσει στο **0** ο counter , τότε το σήμα ***Timeout*** θα γίνει **1** και στην επόμενη ανερχόμενη ακμή θα μεταβούμε στη κατάσταση **is_GREEN** ενώ ο counter θα επαναφερθεί στη τιμή 15000. Ο απαριθμητής φαίνεται στην Εικόνα 24.

```
always @ (posedge Clock)
        begin: RED_DOWN_COUNTER
                if(current_state == is_RED)
                        begin
                                if(!Timeout)
                                        counter <= counter - 1;
                                else
                                        counter = 15000;
                        end

        end
```

Εικόνα 24.

5. Μπλόκ Απαριθμητή για το σήμα **ylw_TIMEOUT – YLW_DOWN_COUNTER**

   Ακριβώς στην ίδια λογική με το από πάνω μπλόκ, το μπλόκ αυτό μοντελοποιεί την καθυστέρηση των 2 δευτερολέπτων που εισάγουμε όταν το φανάρι γίνει πορτοκαλί έως να γίνει κόκκινο. Όταν ο **ylw_counter** γίνει **0** τότε θα έχουν περάσει 2 δευτερόλεπτα, το σήμα ***ylw_Timeout*** θα γίνει **1** και θα μεταβούμε στη κατάσταση **is_RED**. Το μπλόκ φαίνεται στην Εικόνα 25.

```
always @ (posedge Clock)
        begin: YELLOW_DOWN_COUNTER
                if(current_state == is_YELLOW)
                        begin
                                if(!ylw_Timeout)
                                        ylw_counter <= ylw_counter - 1;
                                else
                                        ylw_counter = 2000;
                        end
        end
```

Εικόνα 25.

Έτσι ολοκληρώνεται η περιγραφή του FSM.

6 . Κώδικας Verilog για το Testbench του FSM.

Για την επαλήθευση της λειτουργίας του FSM δημιουργήθηκε το κατάλληλο **testbench**. Το testbench αυτό με όνομα **fanari_TB.v** παράγει τα διανύσματα εισόδου για το προς επαλήθευση κύκλωμα (**DUT** / Design Under Test) . Το DUT ορίζεται σαν ένα module στο εσωτερικό του testbench. Σκοπός του testbench είναι να ελέγξει εάν οι έξοδοι του DUT είναι οι σωστές για όλες τις πιθανές εισόδους.

Αρχικά ορίζεται το **timescale** για το testbench σε **100us/10us** δηλαδή σε κλίμακα 100 microsecond με ακρίβεια 10 microsecond. Αυτό σημαίνει πώς αν δοθεί καθυστέρηση **#1** τότε αυτή αντιστοιχεί σε 1 * 100 = 100 us . Ο τύπος **reg** χρησιμοποιείται για τις εισόδους ***Clock_TB Reset_TB CAR_TB*** ώστε να "οδηγήσει" τις αντίστοιχες εισόδους του DUT που έχουν οριστεί σαν **wire** σε αυτό (Clock Reset CAR), ενώ ο τύπος **wire** χρησιμοποιείται για τις εξόδους ***RED_TB , YLW_TB*** και ***GRN_TB*** που θα "οδηγηθούν" από τις αντίστοιχες εξόδους του DUT που ορίστηκαν σαν **reg** (RED YLW GRN). Επιπλέον η αντιστοίχιση των θυρών στον ορισμό του DUT έγινε **χωροταξικά**. Τα παραπάνω φαίνονται στην Εικόνα 26.

```
`timescale 100us/10us

module fanari_TB();

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

        fanari DUT(RED_TB, YLW_TB, GRN_TB, Clock_TB, Reset_TB, CAR_TB);
```

Εικόνα 26. Design Under Test.

Στη συνέχεια όπως και για το testbench της άσκησης 1 , χρησιμοποιείται ένα ***initial*** μπλόκ για να οδηγήσει το σήμα ***Reset***. Ένα μπλόκ αυτού του τύπου θα εκτελεστεί **μία** μόνο φορά και στο εσωτερικό του το ***Reset*** αρχικά τίθεται σε **0** (δηλαδή είναι ενεργό αφού είναι active-low) και μετά από μια καθυστέρηση 20 * 100 = 2000 us = **2 ms** τίθεται στο **1**. Η καθυστέρηση **#20** αντιστοιχεί σε **2 ms** λόγω του timescale που ορίστηκε.

```
initial
        begin
                Reset_TB = 1'b0;
              #20 Reset_TB = 1'b1;
end
```

Εικόνα 27. Μοντελοποίηση σήματος Reset.

Ακριβώς όπως στην άσκηση 1, για τη μοντελοποίηση του ***Clock*** χρησιμοποιήθηκε ένα ***initial*** και ένα ***always*** μπλόκ. Στο initial μπλόκ που εκτελείται μία φορά η τιμή του σήματος αρχικοποιείται σε **0** και στο always μπλόκ το σήμα θα παίρνει κάθε φορά μετά από μία καθυστέρηση 500 us την αντίθετη τιμή από την τιμή που έχει εκείνη τη στιγμή. Έτσι έχουμε δημιουργήσει ένα ρολόι με περίοδο **1000 us = 1ms** αφού βρίσκεται σε κάθε κατάσταση (high ή low) 500 us. Δηλαδή έχουμε συχνότητα ρολογιού 1 KHz.

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

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

Εικόνα 28. Μοντελοποίηση Clock συχνότητας 1 KHz.

Τέλος το σήμα εισόδου ***CAR*** μοντελοποιήθηκε με ένα ***initial*** μπλόκ στο οποίο του δόθηκαν οι κατάλληλες τιμές ανά τα κατάλληλα χρονικά διαστήματα ώστε κατά την προσομοίωση του testbench να μπορέσουμε να δούμε όλες τις πιθανές περιπτώσεις για το FSM μας.

```
initial
        begin
                CAR_TB = 1'b0;
                #10 CAR_TB = 1'b1;
                #20 CAR_TB = 1'b0;
                #50 CAR_TB = 1'b1;
                #40 CAR_TB = 1'b0;
end
```

Εικόνα 29. Μοντελοποίηση σήματος εισόδου *CAR*.

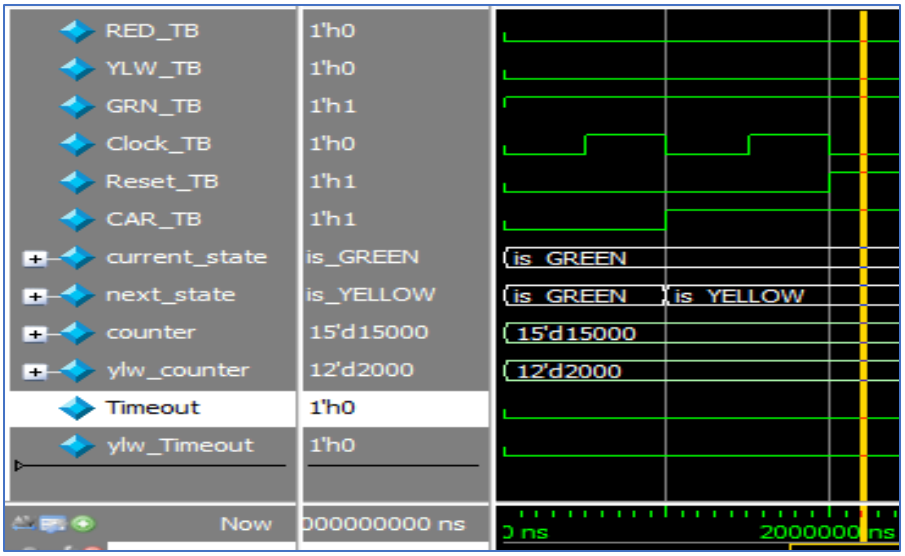
7 . Προσομοίωση και επαλήθευση.

Στο σημείο αυτό πραγματοποιήθηκε προσομοίωση του testbench που περιγράφεται πιο πάνω. Παρατίθενται στιγμιότυπα οθόνης από τις κυματομορφές που προκύπτουν βήμα-βήμα σύμφωνα με τη ροή της προσομοίωσης. Τα σήματα που φαίνονται είναι αυτά που ορίστηκαν στο testbench και με το **port mapping** που γίνεται στην αρχή, αντιστοιχούν στις κατάλληλες εισόδους και εξόδους του DUT.

Προσομοιώθηκαν συνολικά 30 δευτερόλεπτα λειτουργίας.

Αρχικά παρατηρούμε πώς το σήμα ***Reset*** παραμένει στο λογικό **0** από την αρχή του χρόνου έως τα **2 ms** (2000000 ns) όπου γίνεται **1** όπως ακριβώς έχει οριστεί στο testbench. Από **0** έως **2 ms** δηλαδή όσο το ***Reset*** είναι **low** το FSM επαναφέρεται και παραμένει στην κατάσταση **is_GREEN** ανεξαρτήτως εισόδου. Η είσοδος ***CAR*** προκαλεί την ενεργοποίηση του μπλόκ **NEXT_STATE_LOGIC** αλλά στην ανερχόμενη ακμή ρολογιού που έρχεται εντός των **2 ms** αυτών, το σήμα ***current_state*** δε θα επηρεαστεί αφού το σήμα επαναφοράς μας είναι ενεργοποιημένο και έτσι θα παραμείνουμε σίγουρα

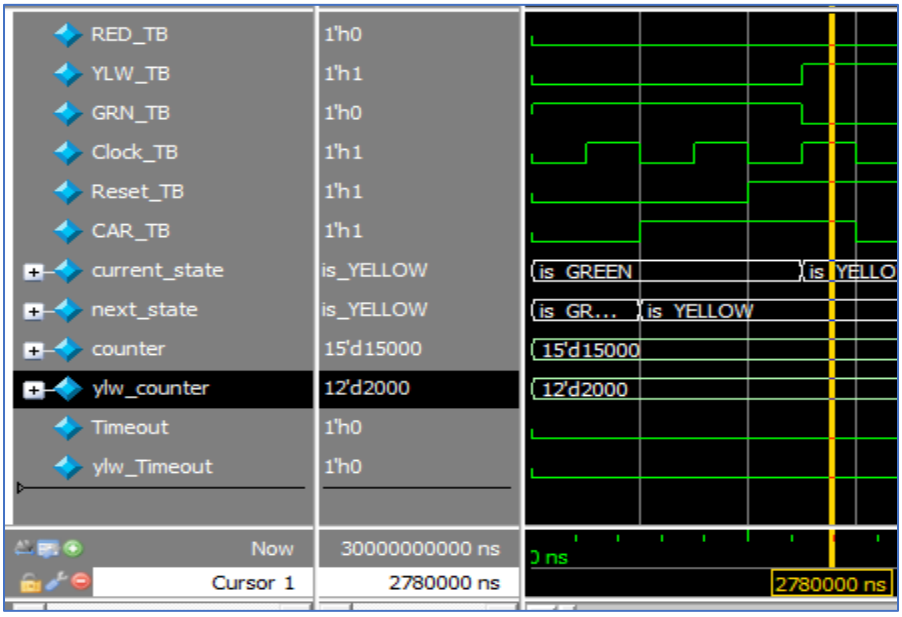
στη κατάσταση **is_GREEN**. Οι παραπάνω διαπιστώσεις επαληθεύονται από την Εικόνα 30 όπου φαίνεται το χρονικό διάστημα **0 – 2 ms .**



Εικόνα 30. Κυματομορφή σήματος *Reset*.

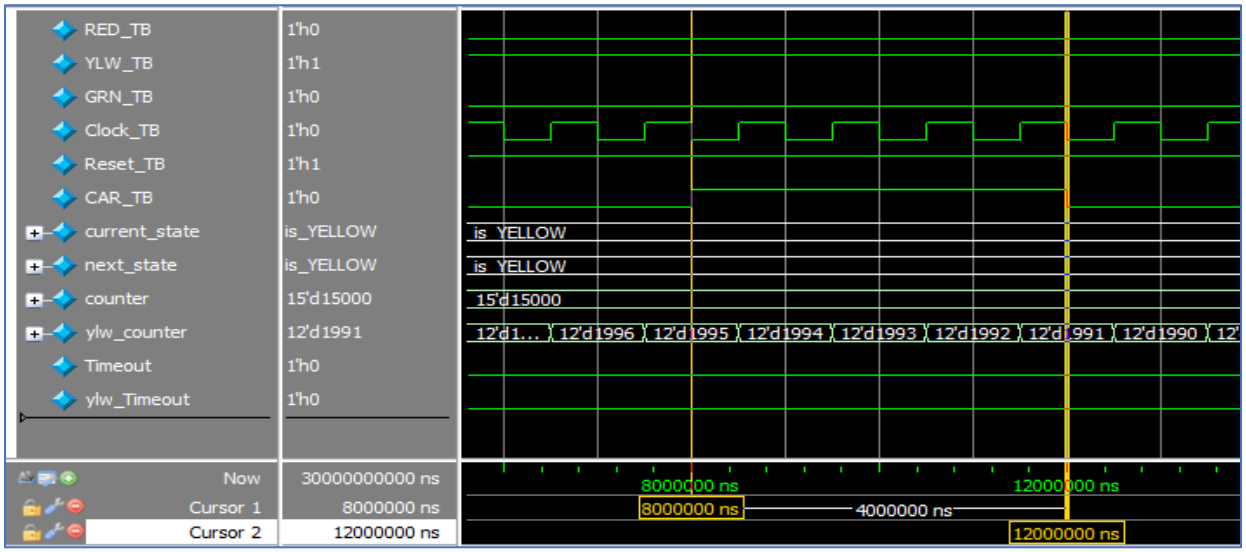
Επιπλέον παρατηρούμε πως οι απαριθμητές έχουν αρχικοποιηθεί σωστά με τις κατάλληλες τιμές (15000 κ 2000).

Στη συνέχεια το σήμα ***Reset*** γίνεται **1** και απενεργοποιείται τη στιγμή **t = 2 ms** . Έτσι αφού το σήμα ***CAR*** συνεχίζει να είναι ενεργό τη στιγμή που έρχεται η επόμενη ανερχόμενη ακμή ρολογιού, η τιμή **is_YELLOW** εκχωρείται στο σήμα ***current_state*** και έτσι μεταβαίνουμε στη κατάσταση όπου το φανάρι είναι κίτρινο. Στην Εικόνα 31 φαίνεται πως το ***current_state*** έχει πάρει την κατάλληλη τιμή , ενώ η έξοδος **YLW_TB** έχει γίνει **1** και οι άλλες **0**. Το σήμα ***CAR*** πέφτει όταν φτάσουμε στα **3 ms** όπως αναμέναμε και βάσει του testbench.



Εικόνα 31. Μετάβαση του φανού σε κίτρινο.

Σημειώνεται πώς το σήμα εισόδου ***CAR*** γίνεται πάλι **1** στα **8 ms** και πέφτει στο **0** στα **12 ms** . Αυτή τη φορά το σήμα δε θα επηρεάσει καθόλου το FSM μας καθώς ήδη το φανάρι έχει γίνει κίτρινο. Αυτό φαίνεται στην Εικόνα 32.



Εικόνα 32.

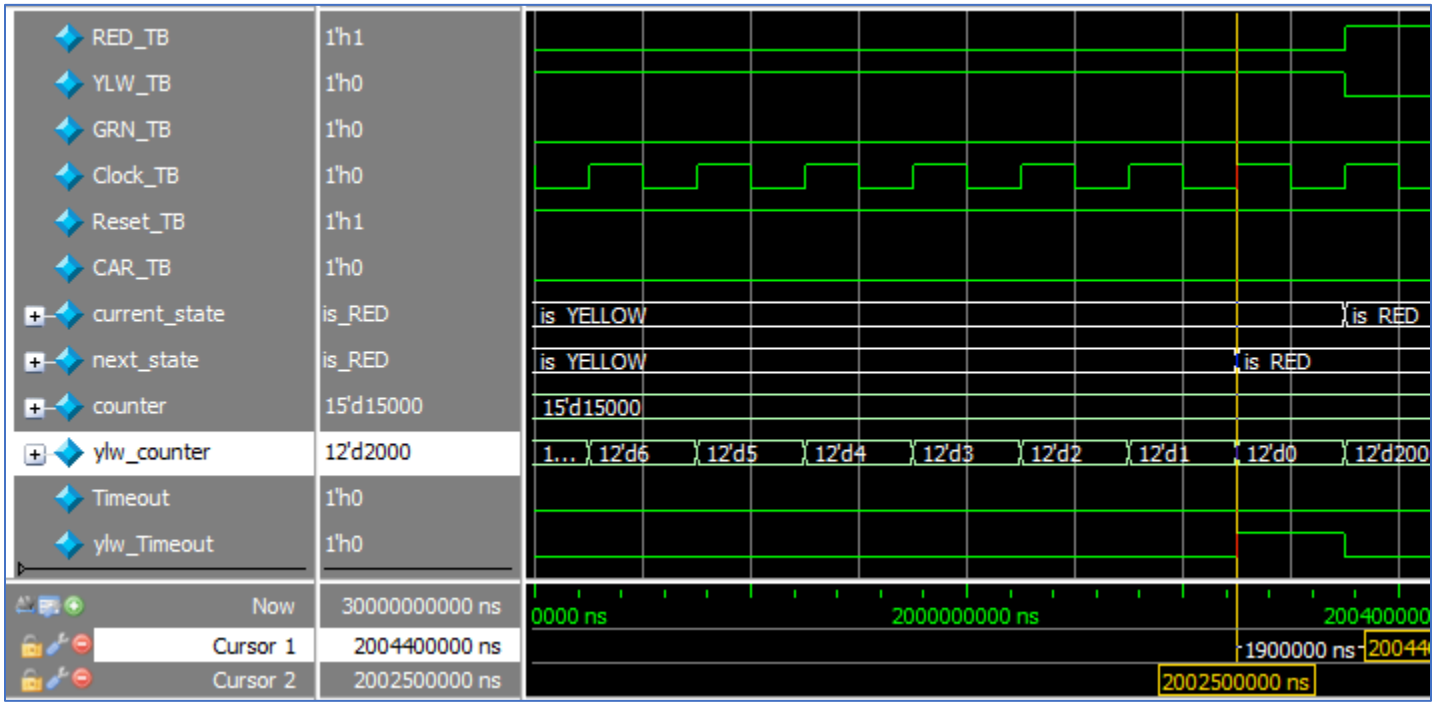
Από τη στιγμή που το φανάρι γίνεται κίτρινο δηλαδή τη στιγμή **2500 us** (2500000 ns / 2.5 ms) τότε σε κάθε επόμενη ανερχόμενη ακμή ρολογιού ο **ylw_counter** θα μειώνεται κατά **1**. Δηλαδή κάθε **1 ms** . Όσο ο **ylw_counter** δεν έχει φτάσει στο **0** το σήμα ***next_state*** άρα και το σήμα ***current_state*** θα παραμένει στη τιμή **is_YELLOW**. Προφανώς για να φτάσει ο counter αυτός στο **0** πρέπει να περάσουν **2000 ms** δηλαδή **2** δευτερόλεπτα.

Στην Εικόνα 33 φαίνεται η χρονική στιγμή λίγο πριν μηδενίσει ο **ylw_counter** . Τη στιγμή που γίνεται **0** σηκώνεται το σήμα ***ylw_Timeout*** . Ακριβώς μόλις γίνει αυτό, αφού η ανερχόμενη ακμή του σήματος ***ylw_Timeout*** περιλαμβάνεται στο sensitivity list του **NEXT_STATE_LOGIC** μπλόκ, η επόμενη κατάσταση υπολογίζεται άμεσα σε **is_RED** δηλαδή το φανάρι να γίνει κόκκινο. Εφόσον στην κατάσταση **is_YELLOW** μεταβήκαμε στα **2.5 ms** , το σήμα ***ylw_Timeout*** σηκώνεται στα **2002.5 ms** .

Στην επόμενη ανερχόμενη ακμή του ρολογιού το σήμα ***ylw_Timeout*** πέφτει και πάλι στο **0** , ο **ylw_counter** επαναφέρεται στη τιμή **2000** και το σήμα ***current_state*** παίρνει την τιμή **is_RED** . Το φανάρι έχει γίνει πλέον κόκκινο. Αυτό διαπιστώνεται και από το ότι η έξοδος **RED_TB** είναι **1** ενώ οι άλλες **0**.

Έτσι καταλαβαίνουμε πως πράγματι από όταν το φανάρι έγινε κίτρινο μέχρι να γίνει κόκκινο έχουμε επιτυχώς θέσει μια καθυστέρηση **δυο δευτερολέπτων**. Υπενθυμίζεται πως η καθυστέρηση αυτή δεν ζητείται από την εκφώνηση αλλά τοποθετήθηκε με σκοπό την καλύτερη προσομοίωση ενός φανού σήμανσης.

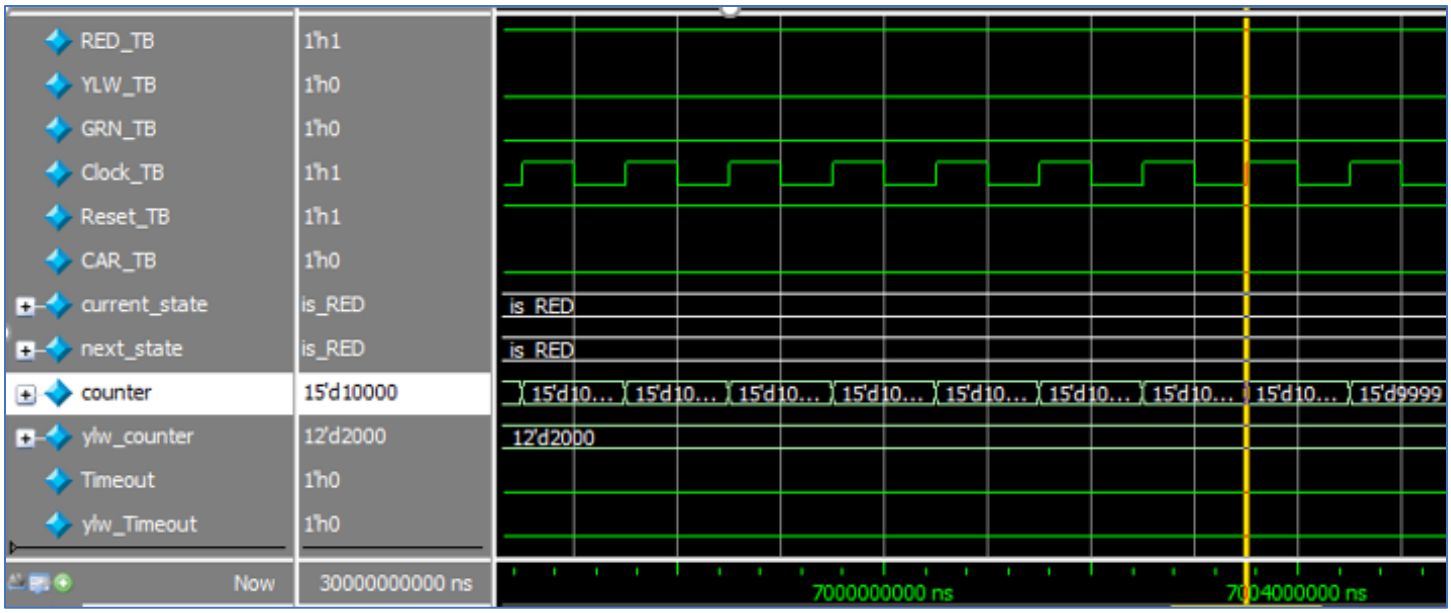
Σημειώνεται επίσης πως όταν περάσουν τα δυο δευτερόλεπτα το σήμα ***next_state*** θα πάρει τιμή αλλά το σήμα ***current_state*** θα τεθεί στην επόμενη ανερχόμενη ακμή ρολογιού. Δηλαδή θα μεταβούμε στην κατάσταση όπου το φανάρι είναι κόκκινο μετά από 2 sec + 1 ms δηλαδή 2.001 seconds . Αυτή η απόκλιση μπορεί να αποφευχθεί αν θέσουμε την τιμή του counter να ξεκινά από **1999** και όχι από **2000** αλλά η απόκλιση αυτή είναι αμελητέα για τον άνθρωπο που περιμένει το φανάρι.



Εικόνα 33. Μετάβαση από κίτρινο σε κόκκινο μετά από 2 δευτερόλεπτα καθυστέρηση.

Με την ίδια λογική όπως πριν , από τη στιγμή που το φανάρι γίνει κόκκινο τότε σε κάθε επόμενη ανερχόμενη ακμή του ρολογιού ο **counter** θα μειώνεται κατά **1** . Ο counter αυτός ξεκινά από τη τιμή **15000** επομένως με βάση το timescale που έχει οριστεί θα μηδενίσει μετά από **15** δευτερόλεπτα και μετά το φανάρι θα γίνει πράσινο. Όπως υποδεικνύεται και στην εκφώνηση.

Συγκεκριμένα για τον ίδιο λόγο που αναφέρθηκε και πριν το φανάρι θα γίνει πράσινο στη πραγματικότητα μετά από 15.001 second . Ουσιαστικά 15 δευτερόλεπτα. Ομοίως θα μπορούσαμε να αρχίσουμε τον counter από το 14999 αντί του 15000 για να έχουμε ακριβώς 15 sec . Στην Εικόνα 34 φαίνεται μια ενδιάμεση κατάσταση όπου έχουν περάσει 5 δευτερόλεπτα (άρα counter = 10000) και μένουμε ακόμα στην **is_RED** (φανάρι κόκκινο) .

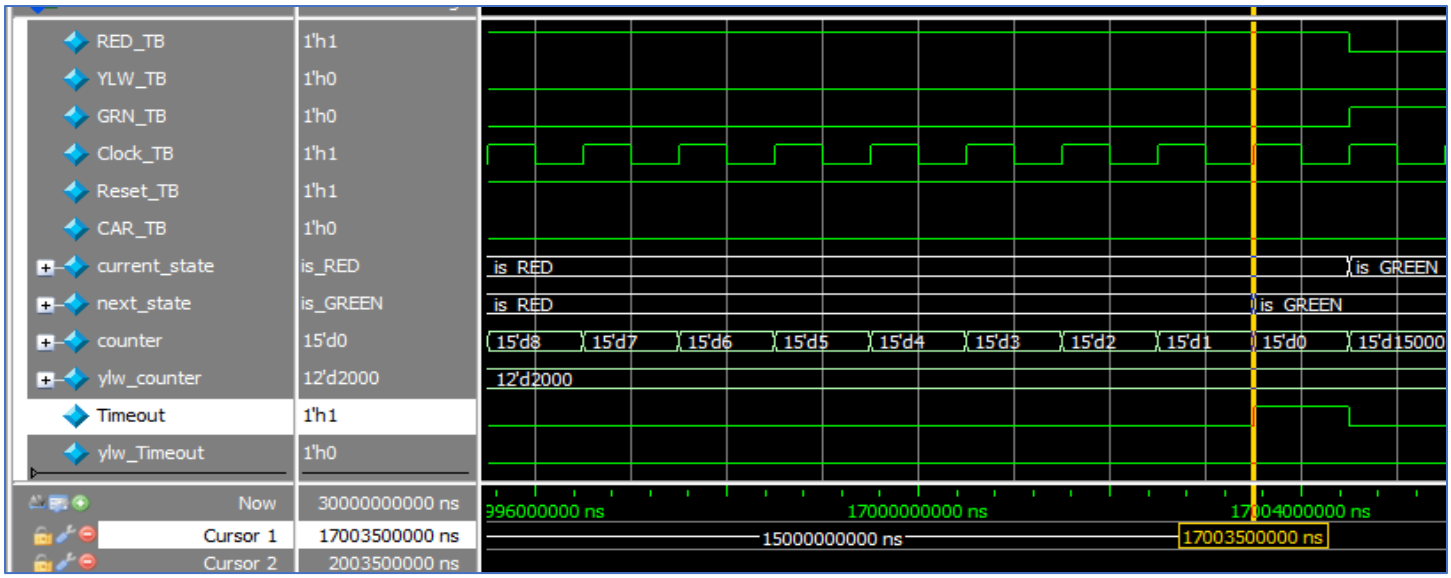


Εικόνα 34. Ενδιάμεση φάση όπου έχουν περάσει 5 δευτερόλεπτα απο όταν έγινε κόκκινο.

Όταν έχουν παρέλθει τα 15 δευτερόλεπτα θα σηκωθεί το σήμα ***Timeout*** και την ίδια στιγμή το σήμα ***next_state*** θα υπολογιστεί σε **is_GREEN .**

Στην επόμενη ανερχόμενη ακμή του ρολογιού το σήμα ***current_state*** θα πάρει την τιμή **is_GREEN** δηλαδή το φανάρι θα γίνει και πάλι πράσινο και το σήμα ***Timeout*** θα γίνει **0.** Επίσης ο counter θα επανέλθει στη τιμή 15000. Αυτά φαίνονται στην Εικόνα 35 .

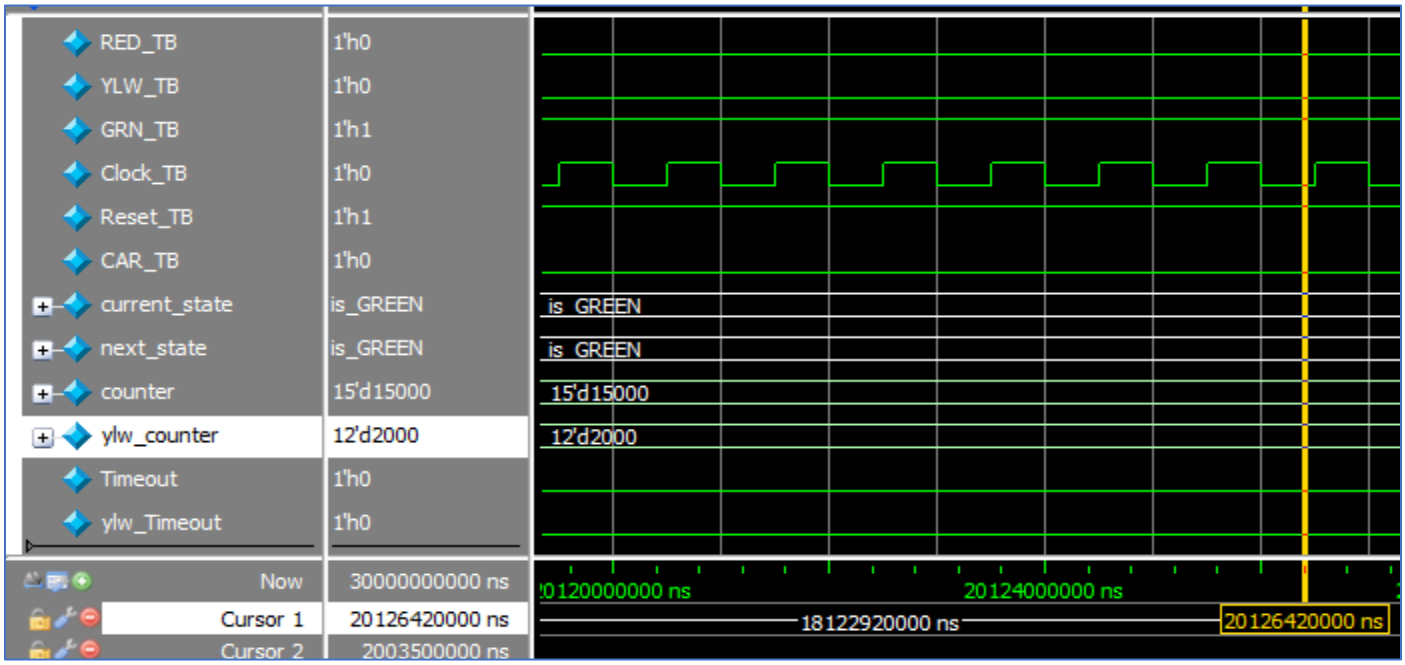
Όπως φαίνεται στην Εικόνα 35 το φανάρι έγινε κόκκινο τη στιγμή 2003.5 ms και το σήμα ***Timeout*** σηκώθηκε τη στιγμή 17003.5 ms , ακριβώς 15 δευτερόλεπτα μετά. Το φανάρι έγινε πράσινο τη στιγμή 17004.5 ms, δηλαδή 15.001 sec αφότου έγινε κόκκινο. Πράγματι δηλαδή , όπως ορίζεται από την εκφώνηση από τη στιγμή που το φανάρι γίνει κόκκινο, μετά από μία καθυστέρηση 15 δευτερολέπτων θα ξαναγίνει πράσινο.



Εικόνα 35. Μετάβαση και πάλι σε πράσινο.

Τέλος από τη στιγμή που το φανάρι γίνει πράσινο θα παραμείνει σωστά έτσι μέχρι το τέλος της προσομοίωσης καθώς στο testbench το σήμα ***CAR*** δεν ξαναγίνεται **1** .

Επιλέγεται μια τυχαία χρονική στιγμή λίγο μετά από 20 δευτερόλεπτα λειτουργίας. Στην Εικόνα 36 βλέπουμε πως το φανάρι παραμένει πράσινο μέχρι να εμφανιστεί αυτοκίνητο στον κάθετο δρόμο (όταν δηλαδή ***CAR = 1***).



Εικόνα 36.

Έτσι ολοκληρώνεται το κομμάτι της προσομοίωσης και της επαλήθευσης. Η λειτουργία του FSM έχει επαληθευτεί πλήρως, όπως ακριβώς υποδεικνύεται και από το διάγραμμα καταστάσεων που δημιουργήθηκε αλλά και από την εκφώνηση.