Γλώσσες Περιγραφής Υλικού Ι

Μηχανές πεπερασμένων καταστάσεων

Νικόλαος Καββαδίας
nkavv@uop.gr

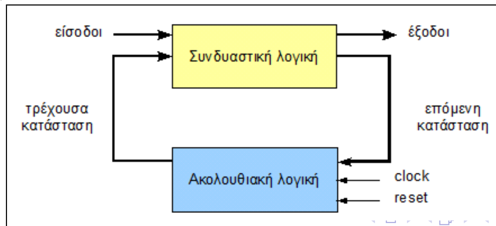
24 Απριλίου 2012

Σκιαγράφηση της διάλεξης

- Μηχανές πεπερασμένων καταστάσεων (FSM: Finite-State Machine)
  - Ορισμός της FSM
  - FSM κατά Mealy και κατά Moore - Μικτοί τύποι
  - Τρόποι καταγραφής της λειτουργίας μιας FSM
  - Δομή της FSM
  - Επανατοποθέτηση (αρχικοποίηση) μιας FSM
  - Κωδικοποίηση της τρέχουσας και της επόμενης κατάστασης
  - Τεχνικές και διαφορετικά στυλ για την περιγραφή μιας FSM
  - Καταχωρημένες έξοδοι στις FSM (registered outputs)
  - Παραδείγματα: απαριθμητής BCD, ελεγκτής ταχύτητας οχήματος (car speed controller)

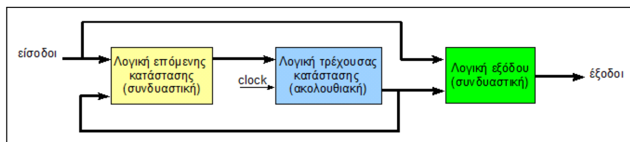
Μηχανές Πεπερασμένων Καταστάσεων: Εισαγωγή και ορισμός

- Οι μηχανές πεπερασμένων καταστάσεων αποτελούν έναν τύπο περιγραφής ακολουθιακών λογικών κυκλωμάτων ο οποίος είναι κατάλληλος για τη μοντελοποίηση κυκλωμάτων που πραγματοποιούν μια σειρά από λειτουργίες
- **Ορισμός:** FSM αποτελεί οποιοδήποτε κύκλωμα ειδικά σχεδιασμένο να διέρχεται με ακολουθιακό τρόπο από ένα σύνολο καταστάσεων
- Ο λόγος που σχεδιάζονται FSM στη Verilog HDL είναι ότι μια υλοποίηση σε υλικό είναι κατά κανόνα πολύ ταχύτερη σε χρόνο επεξεργασίας από την αντίστοιχη υλοποίηση σε λογισμικό ενός μικροεπεξεργαστή
- Γενικό διάγραμμα μιας FSM



Δομή μιας FSM

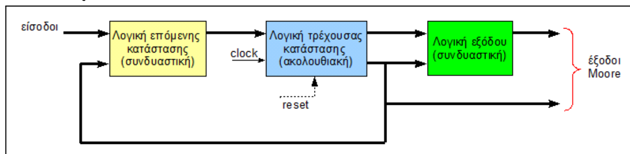
Τυπική οργάνωση μιας FSM



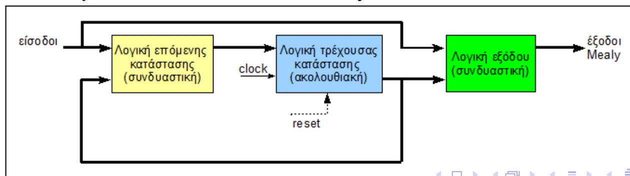
1. Λογική τρέχουσας κατάστασης: Υλοποιείται από καταχωρητή για την αποθήκευση της τρέχουσας κατάστασης της FSM. Η τιμή του αντιπροσωπεύει το συγκεκριμένο στάδιο στο οποίο βρίσκεται η λειτουργία της FSM
2. Λογική επόμενης κατάστασης: Συνδυαστική λογική η οποία παράγει την επόμενη κατάσταση της ακολουθίας. Η επόμενη κατάσταση αποτελεί συνάρτηση των εισόδων της FSM και της τρέχουσας κατάστασης
3. Λογική εξόδου: Συνδυαστική λογική που χρησιμοποιείται για την παραγωγή των σημάτων εξόδου του κυκλώματος. Οι έξοδοι αποτελούν συνάρτηση της εξόδου του καταχωρητή (τρέχουσας) κατάστασης και ΠΙΘΑΝΩΣ των εισόδων της FSM

Κατηγορίες FSM: τύπου Moore και τύπου Mealy (1)

- Στις FSM τύπου Moore οι έξοδοι είναι συνάρτηση μόνο της τρέχουσας κατάστασης
- Οργάνωση μιας FSM τύπου Moore



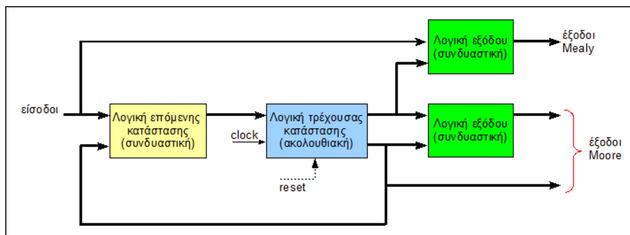
- Στις FSM τύπου Mealy οι έξοδοι είναι συνάρτηση των εισόδων και της τρέχουσας κατάστασης
- Οργάνωση μιας FSM τύπου Mealy



Κατηγορίες FSM: τύπου Moore και τύπου Mealy (2)

☞ Οι FSM τύπου Moore απαιτούν έναν κύκλο ρολογιού παραπάνω από τις τύπου Mealy για τον υπολογισμό των ίδιων εξόδων για τις ίδιες προδιαγραφές μιας FSM

- Σε ορισμένες περιπτώσεις FSM, οι έξοδοι παράγονται απευθείας από τον καταχωρητή τρέχουσας κατάστασης και έτσι δεν απαιτείται η ύπαρξη λογικής εξόδου
- Είναι εφικτός ο συνδυασμός των τύπων Mealy και Moore στην ίδια FSM
- Οργάνωση μιας FSM μικτού τύπου



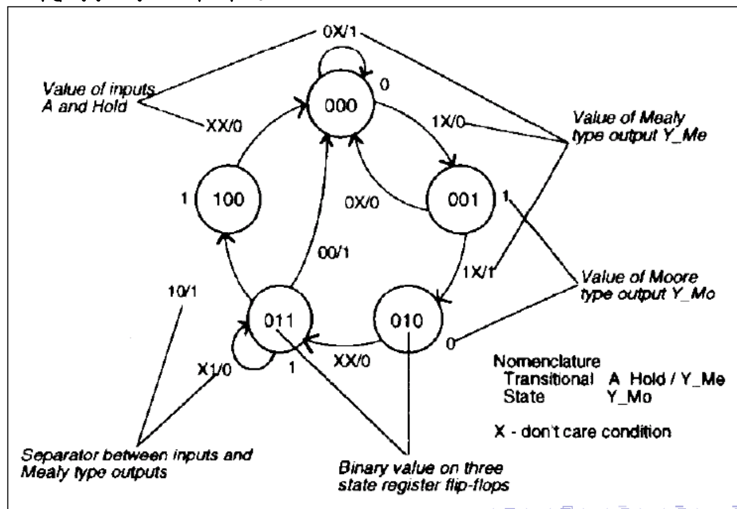
Τρόποι αναπαράστασης μιας FSM (1)

- Μία FSM μπορεί να περιγραφεί είτε με χρήση πίνακα καταστάσεων (state table) είτε με γραφικό τρόπο χρησιμοποιώντας διάγραμμα μεταγωγής καταστάσεων (state transition graph)
- Πίνακας καταστάσεων

Inputs		Current state	Next state	Outputs	
A	Hold			Y_Me	Y_Mo
0	X	000	000	1	0
1	X	000	001	0	0
0	X	001	000	0	1
1	X	001	010	1	1
X	X	010	011	0	0
X	1	011	011	1	1
0	0	011	000	1	1
1	0	011	100	0	1
X	X	100	000	0	1

Τρόποι αναπαράστασης μιας FSM (2)

- Διάγραμμα μεταγωγής καταστάσεων



Κωδικοποίηση της κατάστασης στις FSM (1)

- Για την κωδικοποίηση της κατάστασης μιας FSM χρησιμοποιείται κάποιου είδους δυαδική αριθμητική αναπαράσταση:
  - Ακολουθιακή (sequential): σε κάθε κατάσταση ανατίθενται δυαδικοί αριθμοί κατά αύξουσα σειρά. Ένας καταχωρητής κατάστασης των n bit μπορεί να χρησιμοποιηθεί για 2^n διακριτές καταστάσεις.
  - Κωδικοποίηση Gray: δύο διαδοχικοί αριθμοί διαφέρουν κατά ένα bit. Το i-οστό bit μιας λέξης Gray δίνεται από τη σχέση: $G_i = B_{i+1} \oplus B_i$, όπου B είναι ο αντίστοιχος αριθμός σε κωδικοποίηση binary

Κωδικοποίηση της κατάστασης στις FSM (2)

- (συνέχεια)
  - Κωδικοποίηση Johnson: κωδικοποίηση στην οποία επίσης διαδοχικοί αριθμοί διαφέρουν κατά ένα μόνο bit. Εκφράζεται ως αριστερή ή δεξιά ολίσθηση του αριθμού $2^n - 1$ κατά συγκεκριμένο αριθμό θέσεων
  - Κωδικοποίηση one-hot: σε κάθε κατάσταση αντιστοιχίζεται ξεχωριστό flip-flop. Σε κάθε κατάσταση ένα μόνο flip-flop έχει την τιμή `1'b1`. Για την περιγραφή n καταστάσεων απαιτούνται n flip-flop
  - Κωδικοποίηση one-cold: όπως η one-hot αλλά για κάθε κατάσταση ένα μόνο flip-flop έχει την τιμή `1'b0`

Κωδικοποίηση της κατάστασης στις FSM (3)

- Άλλες κωδικοποιήσεις
  - Κωδικοποίηση καθοριζόμενη από το χρήστη: οποιαδήποτε ανάθεση αριθμητικών αντιστοιχίσεων σε καταστάσεις
  - Παράδειγμα με χρήση `parameter`

```
parameter S1 = 4'b0110, S2 = 4'b0111, S3 = 4'b0000, S4 = 4'b1010;
```

  - Παράδειγμα σύμφωνα με το πρότυπο IEEE 1364.1-2002 για το υποσύνολο της Verilog HDL που υποστηρίζεται για λογική σύνθεση

```
// Attribute defined as a meta-comment
(* synthesis, fsm_state [=<encoding_scheme>] *)
...
// Example 1: Default encoding is used and next_state is the state vector.
(* synthesis, fsm_state *) reg [4:0] next_state;
// Example 2: "onehot" encoding is used and rst_state is the state vector.
(* synthesis, fsm_state = "onehot" *) reg [7:0] rst_state;
```

  - Κωδικοποίηση προσδιοριζόμενη κατά τη σύνθεση: το εργαλείο σύνθεσης επιλέγει την κωδικοποίηση με βάση κάποιο μετρικό, π.χ. μικρότερη επιφάνεια υλικού ή κατανάλωση ενέργειας

Κωδικοποίηση της κατάστασης στις FSM (4)

- Διαμορφώσεις για την κωδικοποίηση των καταστάσεων σε μία FSM

State	Sequential	Gray	Johnson	One-hot
0	0000	0000	00000000	0000000000000001
1	0001	0001	00000001	0000000000000010
2	0010	0011	00000011	0000000000000100
3	0011	0010	00000111	0000000000001000
4	0100	0110	00001111	0000000000010000
5	0101	0111	00011111	0000000000100000
6	0110	0101	00111111	0000000001000000
7	0111	0100	01111111	0000000010000000
8	1000	1100	11111111	0000000100000000
9	1001	1101	11111110	0000001000000000
10	1010	1111	11111100	0000010000000000
11	1011	1110	11111000	0000100000000000
12	1100	1010	11110000	0001000000000000
13	1101	1011	11100000	0010000000000000
14	1110	1001	11000000	0100000000000000
15	1111	1000	10000000	1000000000000000

Αρχικοποίηση και ασφαλής λειτουργία της FSM

- Για την αρχικοποίηση της FSM σε μία γνωστή αρχική κατάσταση επιβάλλεται η χρήση ασύγχρονης επανατοποθέτησης (asynchronous reset)
- Με τον τρόπο αυτό εξασφαλίζεται ότι με την πρώτη ακμή του ρολογιού, η FSM μπορεί να μεταβεί σε μία νέα κατάσταση
- Σε περίπτωση που είτε δεν χρησιμοποιείται reset είτε χρησιμοποιείται μόνο σύγχρονη επανατοποθέτηση (synchronous reset)
  - Δεν υπάρχει τρόπος να προβλεφθεί η αρχική κατάσταση της FSM
  - Υπάρχει περίπτωση εγκλωβισμού της FSM σε κάποια αχρησιμοποίητη κατάσταση (unused state)
- Προκειμένου να υπάρχει η δυνατότητα επιστροφής σε έγκυρη κατάσταση χρειάζεται να ληφθούν υπόψη στη σχεδίαση και οι τυχόν αχρησιμοποίητες καταστάσεις

Τεχνικές περιγραφής μιας FSM (1)

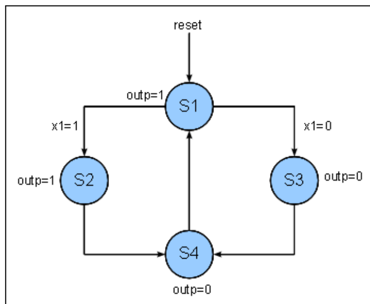
- Γενικά υφίστανται αρκετές τεχνικές για τη σύνταξη της περιγραφής μιας FSM
- Ορισμένες από τις περισσότερο διαδεδομένες είναι οι εξής:
  1. FSM με μία διεργασία (μπλοκ `always`): Η λογική επόμενης κατάστασης, τρέχουσας κατάστασης και εξόδου περιγράφονται σε μία `always`. Η τεχνική αυτή ΔΕΝ μπορεί να χρησιμοποιηθεί για ασύγχρονες εξόδους
  2. FSM με δύο διεργασίες: Η λογική επόμενης κατάστασης και τρέχουσας κατάστασης υλοποιούνται σε μία `always` και η λογική εξόδου σε μία δεύτερη
  3. FSM με τρεις διεργασίες: Η λογική επόμενης κατάστασης, τρέχουσας κατάστασης και εξόδου περιγράφονται σε ξεχωριστές `always`

Τεχνικές περιγραφής μιας FSM (2)

- Εναλλακτικές τεχνικές για τη σύνταξη της περιγραφής μιας FSM
  1. FSM με δύο διεργασίες με τη λογική τρέχουσας κατάστασης να υλοποιείται σε μία `always` και τη λογική επόμενης κατάστασης και εξόδου να υλοποιούνται σε μία δεύτερη `always`
  2. Εναλλακτικά η λογική εξόδου μπορεί να υλοποιηθεί από συντρέχουσες δηλώσεις `assign` στην περίπτωση των ασύγχρονων εξόδων
  3. Σε πολλές εφαρμογές τα σήματα εξόδου πρέπει να είναι σύγχρονα έτσι ώστε η έξοδος να ενημερώνεται μόνο με την ακμή του ρολογιού. Μία τέτοια FSM μπορεί να υλοποιηθεί με χρήση ακολουθιακού μπλοκ `always` για τη λογική εξόδου

Διάγραμμα μεταγωγής καταστάσεων για μία απλή FSM 4 καταστάσεων

- Η FSM του παραδείγματος καθορίζεται από:
  - Τέσσερις καταστάσεις: S1, S2, S3, S4
  - Μία είσοδο: x1
  - Μία έξοδο: outp
  - Πέντε περιπτώσεις μετάβασης από κατάσταση σε κατάσταση σύμφωνα με το παρακάτω διάγραμμα μεταγωγής καταστάσεων



Παράδειγμα FSM με μία διεργασία

```
module fsm_1 (clk, reset, x1, outp);
  input clk, reset, x1;
  output outp;
  reg outp;
  reg [1:0] state;
  parameter s1 = 2'b00, s2 = 2'b01,
            s3 = 2'b10, s4 = 2'b11;

  initial begin
    state = 2'b00;
  end

  always @(posedge clk or posedge reset)
  begin
    if (reset) begin
      state <= s1;
      outp  <= 1'b1;
    end
    else begin
      case (state)
```

```
        s1: begin
          if (x1 == 1'b1) begin
            state <= s2;
          end
          else begin
            state <= s3;
          end
          outp  <= 1'b1;
        end
        s2: begin
          state <= s4;
          outp  <= 1'b1;
        end
        s3: begin
          state <= s4;
          outp  <= 1'b0;
        end
        s4: begin
          state <= s1;
          outp  <= 1'b0;
        end
      endcase
    end
  end
endmodule
```

Παράδειγμα FSM με δύο διεργασίες (ξεχωριστή Output Logic)

```
module fsm_2 (clk, reset, x1, outp);
  input clk, reset, x1;
  output outp;
  reg outp;
  reg [1:0] state;
  parameter s1 = 2'b00, s2 = 2'b01,
            s3 = 2'b10, s4 = 2'b11;

  initial begin
    state = 2'b00;
  end

  always @(posedge clk or posedge reset)
  begin
    if (reset)
      state <= s1;
    else
    begin
      case (state)
```

```
        s1: if (x1 == 1'b1)
              state <= s2;
            else
              state <= s3;
        s2: state <= s4;
        s3: state <= s4;
        s4: state <= s1;
      endcase
    end
  end

  always @(state)
  begin
    case (state)
      s1: outp = 1'b1;
      s2: outp = 1'b1;
      s3: outp = 1'b0;
      s4: outp = 1'b0;
    endcase
  end
endmodule
```

Παράδειγμα FSM με τρεις διεργασίες

```
module fsm_3 (clk, reset, x1, outp);
  input clk, reset, x1;
  output outp;
  reg outp;
  reg [1:0] state;
  reg [1:0] next_state;
  parameter s1 = 2'b00, s2 = 2'b01,
            s3 = 2'b10, s4 = 2'b11;

  initial begin
    state = 2'b00;
  end

  always @(posedge clk or posedge reset)
  begin
    if (reset)
      state <= s1;
    else
      state <= next_state;
  end
```

```
  always @(state or x1)
  begin
    case (state)
      s1: if (x1 == 1'b1)
            next_state = s2;
          else
            next_state = s3;
      s2: next_state = s4;
      s3: next_state = s4;
      s4: next_state = s1;
    endcase
  end

  always @(state)
  begin
    case (state)
      s1: outp = 1'b1;
      s2: outp = 1'b1;
      s3: outp = 1'b0;
      s4: outp = 1'b0;
    endcase
  end
endmodule
```

Το ίδιο παράδειγμα με δύο διεργασίες και απομόνωση της λογικής τρέχουσας κατάστασης

```
module fsm_2b (clk, reset, x1, outp);
  input clk, reset, x1;
  output outp;
  reg outp;
  reg [1:0] state;
  reg [1:0] next_state;
  parameter s1 = 2'b00, s2 = 2'b01,
            s3 = 2'b10, s4 = 2'b11;

  initial begin
    state = s1;
  end

  always @(posedge clk or posedge reset)
  begin
    if (reset)
      state <= s1;
    else
      state <= next_state;
  end

  always @(state or x1)
  begin
    case (state)
```

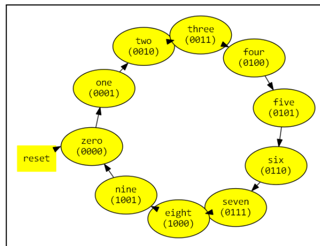
```
      s1: begin
        outp = 1'b1;
        if (x1 == 1'b1)
          next_state = s2;
        else
          next_state = s3;
      end
      s2: begin
        outp = 1'b1;
        next_state = s4;
      end
      s3: begin
        outp = 1'b0;
        next_state = s4;
      end
      s4: begin
        outp = 1'b0;
        next_state = s1;
      end
      default: begin
  // avoid outputs and next state logic
  // being latched
        outp = 1'b0;
        next_state = s1;
      end
    endcase
  end
endmodule
```

Το ίδιο παράδ. με δύο διεργασίες, απομόνωση λογικής τρέχουσας κατάστασης και αποθηκευμένη έξοδο

```
module fsm_2c (clk, reset, x1, outp);
  input clk, reset, x1;
  output outp;
  reg outp, temp;
  reg [1:0] state;
  reg [1:0] next_state;
  parameter s1 = 2'b00, s2 = 2'b01,
            s3 = 2'b10, s4 = 2'b11;

  initial begin
    state = s1;
  end

  always @(posedge clk or posedge reset)
  begin
    if (reset)
      state <= s1;
    else
      outp  <= temp;
      state <= next_state;
  end

  always @(state or x1)
  begin
    case (state)
      s1: begin
        temp = 1'b1;
        if (x1 == 1'b1)
          next_state = s2;
        else
          next_state = s3;
      end
      s2: begin
        temp = 1'b1;
        next_state = s4;
      end
      s3: begin
        temp = 1'b0;
        next_state = s4;
      end
      s4: begin
        temp = 1'b0;
        next_state = s1;
      end
      default: begin
        temp = 1'b0;
        next_state = s1;
      end
    endcase
  end
endmodule
```

Σχεδιασμός FSM για τον απαριθμητή BCD

- Ο απαριθμητής BCD αποτελεί παράδειγμα υλοποίησης μιας μηχανής Moore επειδή η έξοδός του εξαρτάται μόνο από την αποθηκευμένη (τρέχουσα) κατάσταση
- Για την κωδικοποίηση της κατάστασης του FSM ως ακολουθιακή χρειάζεται ένας καταχωρητής εύρους $\lceil log_2(10) \rceil = 4$ bit.
- Διάγραμμα μεταγωγής καταστάσεων για τον απαριθμητή BCD



Περιγραφή του απαριθμητή BCD σε Verilog HDL (1)

```
module bcd (clk, reset, count);
  input clk, reset;
  output [3:0] count;
  reg [3:0] count;
  reg [3:0] current_state;
  reg [3:0] next_state;
  parameter ZERO = 4'b0000, ONE = 4'b0001, TWO = 4'b0010, THREE = 4'b0011,
            FOUR = 4'b0100, FIVE = 4'b0101, SIX = 4'b0110, SEVEN = 4'b0111,
            EIGHT = 4'b1000, NINE = 4'b1001;

  always @(posedge clk or posedge reset)
  begin
    if (reset)
      current_state <= ZERO;
    else
      current_state <= next_state;
  end

  always @(current_state)
  begin
```

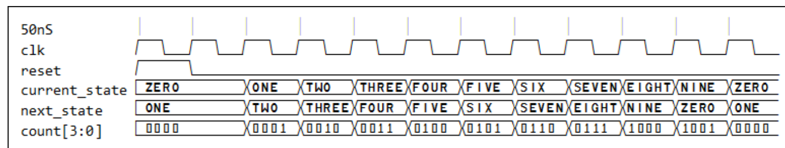
Περιγραφή του απαριθμητή BCD σε Verilog HDL (2)

```
    case (current_state)
      ZERO: begin
        count = ZERO;
        next_state = ONE;
      end
      ONE: begin
        count = ONE;
        next_state = TWO;
      end
      TWO: begin
        count = TWO;
        next_state = THREE;
      end
      THREE: begin
        count = THREE;
        next_state = FOUR;
      end
      FOUR: begin
        count = FOUR;
        next_state = FIVE;
      end
```

```
      FIVE: begin
        count = FIVE;
        next_state = SIX;
      end
      SIX: begin
        count = SIX;
        next_state = SEVEN;
      end
      SEVEN: begin
        count = SEVEN;
        next_state = EIGHT;
      end
      EIGHT: begin
        count = EIGHT;
        next_state = NINE;
      end
      NINE: begin
        count = NINE;
        next_state = ZERO;
      end
      default: begin
        count = 4'b1111;
        next_state = ZERO;
      end
    endcase
  end
endmodule
```

Προσομοίωση του απαριθμητή BCD

Χρονικό διάγραμμα του κυκλώματος

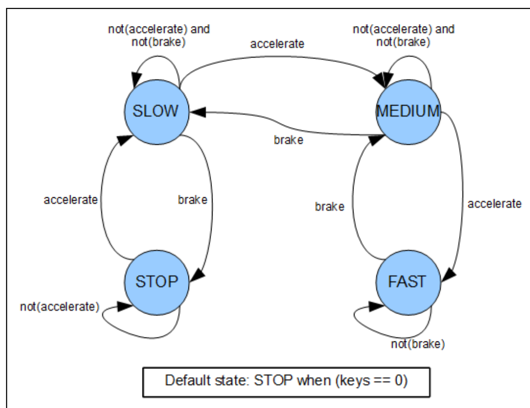


Σχεδιασμός ελεγκτή ταχύτητας οχήματος (car speed controller)

- Στο πρόβλημα ζητείται ο σχεδιασμός μιας FSM για τον έλεγχο της ταχύτητας ενός οχήματος το οποίο διαθέτει αυτόματο κιβώτιο ταχυτήτων
- Το όχημα διαθέτει τρεις ταχύτητες (SLOW, MEDIUM, FAST), οι οποίες θα αντιπροσωπευτούν από αντίστοιχες καταστάσεις
- Όταν το όχημα βρίσκεται σε ηρεμία (δεν έχει εισαχθεί το κλειδί του αυτοκινήτου) αντιστοιχεί η κατάσταση STOP
- Το κλειδί, αντιπροσωπεύει ένα εξωτερικό σήμα το οποίο ονομάζουμε keys και στην ουσία αντιστοιχεί σε είσοδο επανατοποθέτησης (reset) αρνητικής λογικής
- Ο ελεγκτής δέχεται τα σήματα εισόδου accelerate και brake από τον ποδομοχλό επιταχύνσεως (γκάζι) και το ποδόφρενο (φρένο), αντίστοιχα

Διάγραμμα μεταγωγής καταστάσεων για τον car speed controller

- Οι είσοδοι της FSM είναι το κλειδί (keys), το σήμα από το γκάζι (accelerate) και το σήμα από το φρένο (brake). Έξοδος της FSM είναι η στάθμη ταχύτητας (speed)



Περιγραφή του car speed controller σε Verilog HDL (1)

```
module car_speed_cntl (clock, keys, brake, accelerate, speed);
  input clock, keys, brake, accelerate;
  output [1:0] speed;
  reg[1:0] speed, newspeed;
  parameter STOP = 2'b00, SLOW = 2'b01,
            MEDIUM = 2'b10, FAST = 2'b11;

  always @(posedge clock or negedge keys)
  begin
    if (!keys)
      speed <= STOP;
    else
      speed <= newspeed;
  end

  always @(speed or keys or brake or accelerate)
  begin
    case (speed)
      STOP: begin
        if (accelerate)
          newspeed = SLOW;
        else
          newspeed = STOP;
      end
```

Περιγραφή του car speed controller σε Verilog HDL (2)

```
      SLOW: begin
        if (brake)
          newspeed = STOP;
        else if (accelerate)
          newspeed = MEDIUM;
        else
          newspeed = SLOW;
      end
      MEDIUM: begin
        if (brake)
          newspeed = SLOW;
        else if (accelerate)
          newspeed = FAST;
        else
          newspeed = MEDIUM;
      end
      FAST: begin
        if (brake)
          newspeed = MEDIUM;
        else
          newspeed = FAST;
      end
      default:
        newspeed = STOP;
    endcase
  end

endmodule
```

Αρχείο testbench για τον car speed controller (1)

```
`timescale 1 ns / 10 ps

module main;
  reg clock, keys, brake, accelerate;
  wire [1:0] speed;

  car_speed_cntl test (
    .clock(clock),
    .keys(keys),
    .brake(brake),
    .accelerate(accelerate),
    .speed(speed)
  );

  // Test clock generation.
  //
  // Clock pulse starts from 0.
  initial
    clock = 1'b0;
  // Free-running clock
  always
    #25 clock = ~clock;

  // Data stimulus
  initial
  begin
```

Αρχείο testbench για τον car speed controller (2)

```
    #10 keys = 1'b0; brake = 1'b0; accelerate = 1'b0;
    #50 keys = 1'b1;
    #50 accelerate = 1'b1;
    #50 accelerate = 1'b1;
    #50 brake = 1'b1;
    #50 brake = 1'b1; accelerate = 1'b1;
    #50 brake = 1'b0; accelerate = 1'b1;
    #50 accelerate = 1'b0;
    #50 accelerate = 1'b1;
    #150 brake = 1'b1;
    #200
//     $stop;    // for Modelsim
    $finish; // for Icarus Verilog
  end

  // Write simulation data to a Value Change Dump (VCD) file.
  // More on this in lecture 07, scheduled next week!
  initial
  begin
    // Open a VCD file for writing
    $dumpfile("car_speed_cntl.vcd");
    // Dump the values of all nets and wires in module
    // "main", since simulation time 0
    $dumpvars(0, main);
  end

endmodule
```

Προσομοίωση του car speed controller

Χρονικό διάγραμμα του κυκλώματος