Ταχεία Πρωτοτυποποίηση Ψηφιακών Συστημάτων

Workshop #2 – Ακολουθιακά Κυκλώματα στη Verilog

Α.Ο. Φαρμάκης

In association with the ACM Student Chapter University of Patras

Περιεχόμενα του σημερινού workshop

- Επισκόπηση στα Finite State Machines (FSMs) (03 – 08)
- Επισκόπηση στα latch & στα flip-flops (09 – 12)
- Το always block της Verilog & τρόποι χρήσης (13 – 18)
- Καλές πρακτικές σχεδίασης με always block (19 – 20)
- Καταχωρητές (21 – 27)
- Μετρητές (28 – 37)
- FSMs σε Verilog (38 – 49)
- Βιβλιογραφία (50)

Επισκόπηση στα Finite State Machines (FSMs) (1/6)

Μια μηχανή πεπερασμένων καταστάσεων είναι μια πεντάδα $(Q, \Sigma, \delta, q_0, F)$, όπου:

- Q είναι ένα πεπερασμένο σύνολο, τα στοιχεία του οποίου ονομάζομαι ***καταστάσεις***.
- Σ είναι ένα πεπερασμένο σύνολο, που ονομάζεται ***αλφάβητο***.
- δ: $Q \times \Sigma \rightarrow Q$ είναι η ***συνάρτηση μεταβάσεων***.
- $q_0 \in Q$ είναι η ***εναρκτήρια κατάσταση***.
- $F \subseteq Q$ το ***σύνολο των καταστάσεων αποδοχής***.

Επισκόπηση στα Finite State Machines (FSMs) (2/6)

Τα FSMs είναι ένα από διάφορα ειδών υπολογιστικών μοντέλων, δηλαδή είναι “κατασκευές” που χρησιμοποιούμε για τον υπολογισμό διάφορων θεμάτων / καταστάσεων.

Χωρίς να μπούμε σε πολλές λεπτομέρειες όμως, τα FSMs αποτελούν το απλούστερο δυνατό υπολογιστικό μοντέλο με μνήμη, σε αντίθεση με τη συνδυαστική λογική που είδαμε προηγουμένως, η οποία δεν έχει την ικανότητα μνήμης.

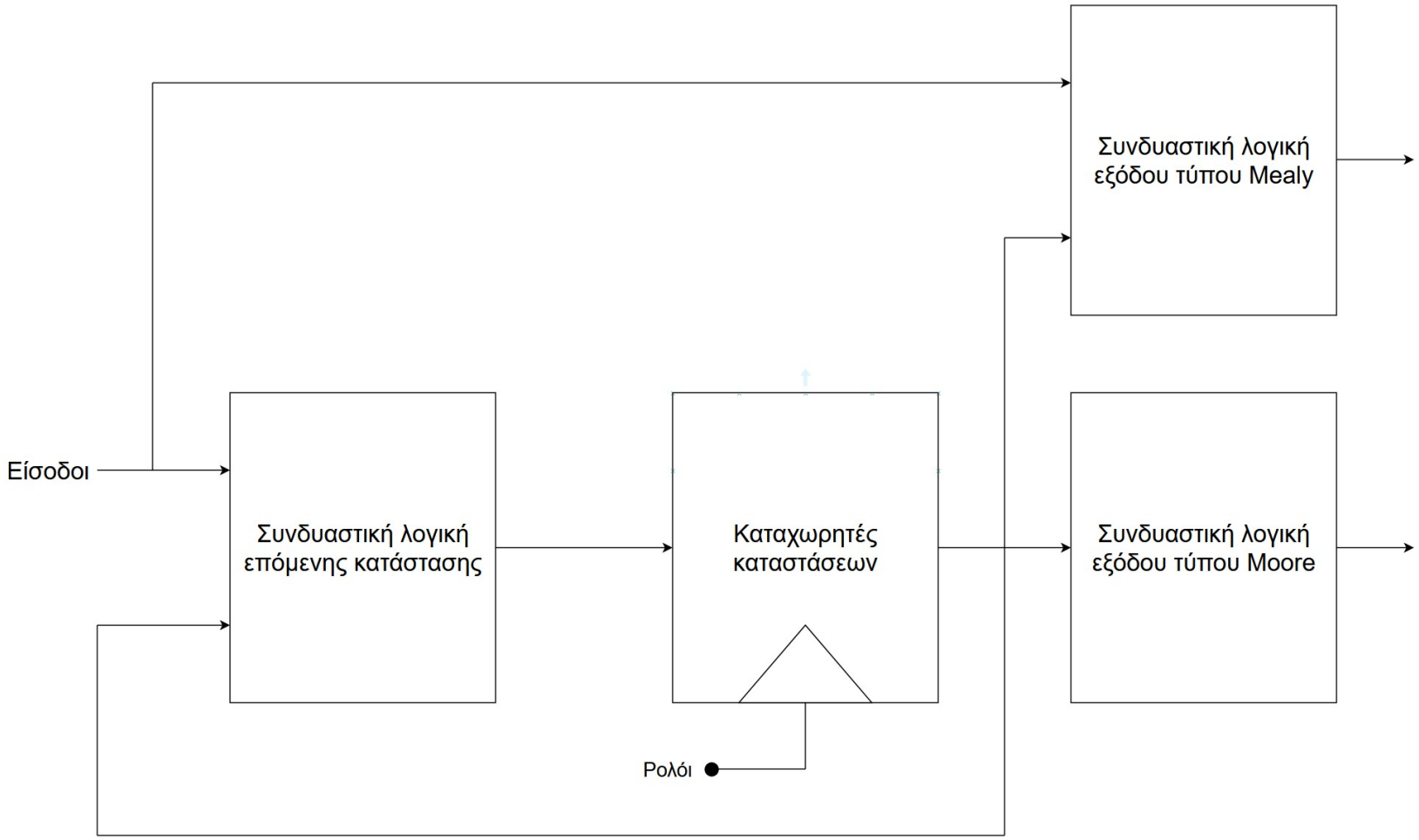
Επισκόπηση στα Finite State Machines (FSMs) (3/6)

Και που δηλαδή συναντάμε μηχανές πεπερασμένων καταστάσεων;

ΠΑΝΤΟΥ!

Αυτόματοι πωλητές, φανάρια αυτοκινητοδρόμων, ανελκυστήρες, ταμειακές μηχανές αποτελούν κάποια από τα πρώτα και πιο προφανή παραδείγματα τέτοιων μηχανών στη πραγματική ζωή.

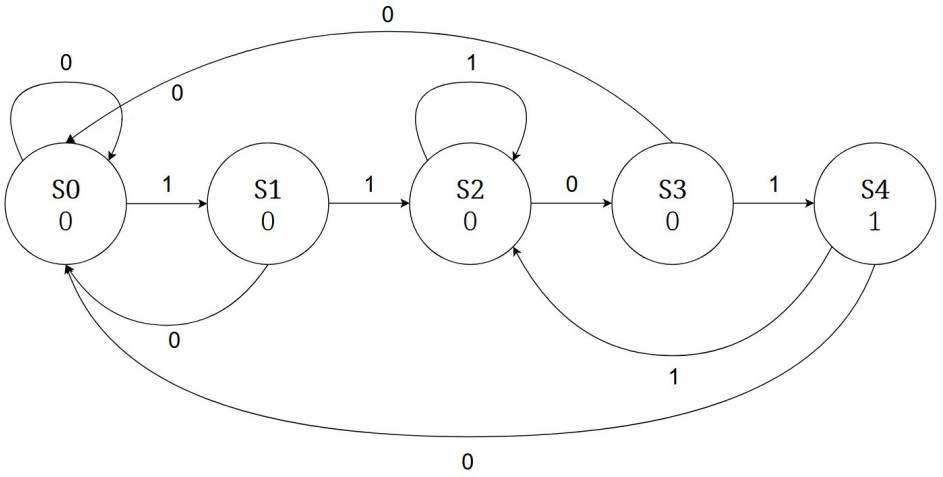
Επισκόπηση στα Finite State Machines (FSMs) (4/6)



Επισκόπηση στα Finite State Machines (FSMs) (5/6)

Παράδειγμα: Moore μηχανή για την αναγνώριση της "λέξης" 1101 μέσα σε μια συμβολοσειρά.

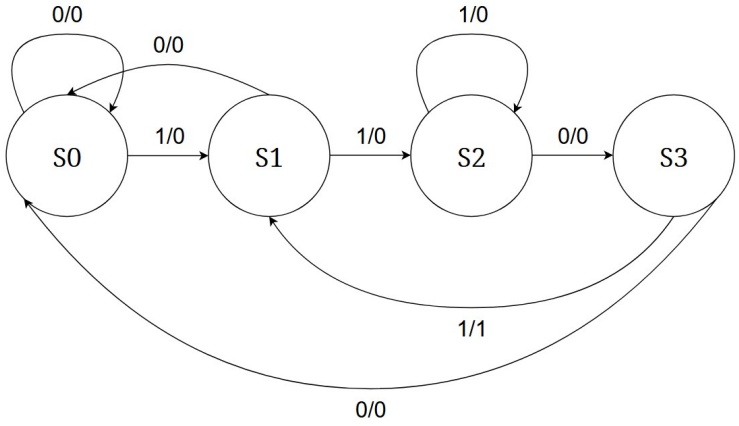
Η έξοδος της μηχανής είναι συνάρτηση της τωρινής κατάστασης.



Επισκόπηση στα Finite State Machines (FSMs) (6/6)

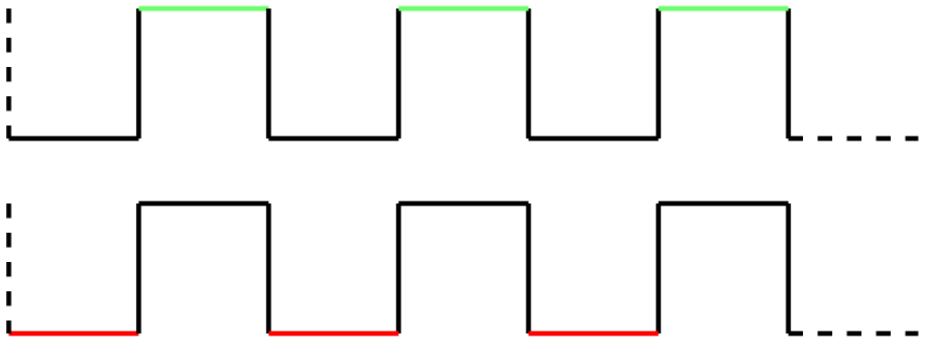
Παράδειγμα: Mealy μηχανή για την αναγνώριση της "λέξης" 1101 μέσα σε μια συμβολοσειρά.

Η έξοδος της μηχανής είναι συνάρτηση της τωρινής κατάστασης **ΚΑΙ** της εισόδου.



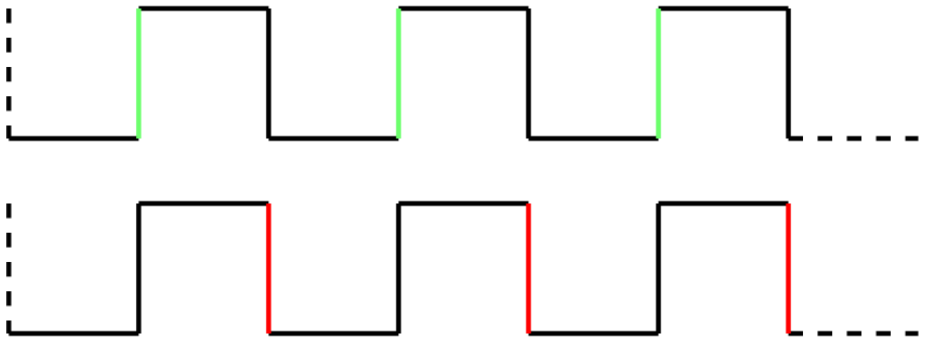
Επισκόπηση στα latch & στα flip-flops (1/4)

Μανδαλωτές (latches):
Συσκευές μνήμης οι οποίες δειγματοληπτούν τις εισόδους στη θετική (ή αρνητική αντίστοιχα) ημιπερίοδο του ρολογιού. Κατά την άλλη ημιπερίοδο, κρατάνε την τιμή που είχαν προηγουμένως.



Επισκόπηση στα latch & στα flip-flops (2/4)

Flip-flop: Συσκευές μνήμης οι οποίες δειγματοληπτούν τις εισόδους στη θετική (ή αρνητική αντίστοιχα) ακμή του ρολογιού. Μέχρι την επόμενη θετική (ή αρνητική) ακμή, κρατάνε την τιμή που είχαν προηγουμένως.



Επισκόπηση στα latch & στα flip-flops (3/4)

Υπάρχουν διαφόρων ειδών latch και flip-flop, τα οποία μπορείτε να διερευνήσετε και τις λειτουργίες τους μόνοι σας.

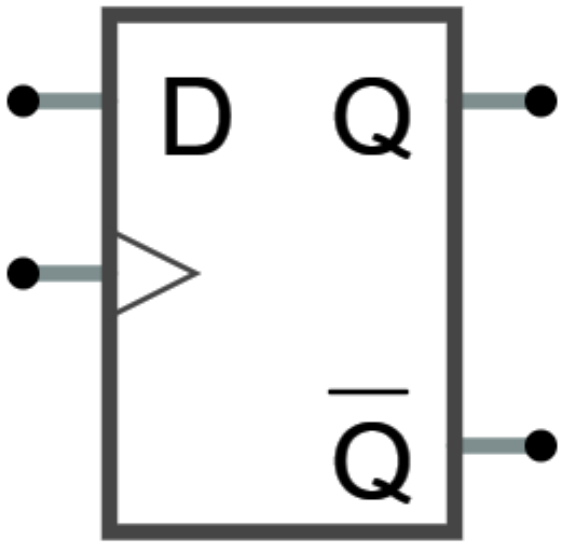
Στα πλαίσια του workshop, θα ασχοληθούμε (αποκλειστικά) με D(ata) Flip-Flop, όπου υπάρχει βεβαίως και αντίστοιχη λειτουργία για το latch.

Πέραν της απλότητας αυτού, το D FF είναι το πιο οικονομικό και για αυτό το λόγο χρησιμοποιείται κατά κόρο.

Επισκόπηση στα latch & στα flip-flops (4/4)

D Flip-flop

Λειτουργία: “Διοχέτευση” της εισόδου που δειγματολήφθηκε στην θετική ακμή του ρολογιού του προηγούμενου κύκλου στην έξοδο



Είσοδοι: D, Clk

Έξοδοι: Q, ~Q

Εξίσωση: Q(t) = D(t-1)

Πιθανώς και με (α)σύγχρονες εισόδους Reset και Preset

Το always block της Verilog & τρόποι χρήσης (1/6)

Στο προηγούμενο workshop είδαμε το assign, το οποίο αποτελεί ένα είδους συνεχούς ανάθεσης (continuous assignment), δηλαδή εκτελείται καθ'όλη τη διάρκεια εκτέλεσης.

Σήμερα, θα δούμε το always, το οποίο αποτελεί ένα είδους διαδικαστικής ανάθεσης (procedural assignment), δηλαδή η τιμή των αναθέσεων όταν προκληθεί αλλαγή / γεγονός που περιγράφεται στη *λίστα ευαισθησίας*.

Το always block της Verilog & τρόποι χρήσης (2/6)

Το συντακτικό του **always** είναι το εξής:

```
always @(event) [statement]

always @(event) begin    // @(*) includes all related signals
    [multiple statements]
end
```

Όπου οτιδήποτε μέσα στις παρενθέσεις (τα γεγονότα δηλαδή) αποτελούν τη λίστα ευαισθησίας. Σε κάθε block οι αναθέσεις εκτελούνται σειριακά, αλλά ΟΛΑ τα block σε ένα κύκλωμα εκτελούνται παράλληλα. Επίσης, οι αναθέσεις ΕΝΤΟΣ του **always** πρέπει να είναι τύπου **reg**.

Το always block της Verilog & τρόποι χρήσης (3/6)

```
module simple_4bit_alu (a, b, opcode, carry, zero, alu_out);
   input [3:0] a, b;
   input [1:0] opcode;
   output reg [3:0] alu_out;
   output reg carry;
   output zero;

   // Notice that “carry” and “alu_out” are reg types!
   always @(a or b or opcode) begin     // Could be done as “@(*)” as well
      case (opcode) // Functionally equivalent to ternary (sort of, more on that soon)
         2’b00: {carry, alu_out} = $signed(a) + $signed(b);
         2’b01: {carry, alu_out} = $signed(a) - $signed(b);
         2’b10: alu_out = a & b;
         2’b11: alu_out = a | b;
      endcase
   end
   assign zero = (alu_out == 0);      // Notice that “zero” is a wire type!

endmodule
```

Το always block της Verilog & τρόποι χρήσης (4/6)

Όπως είδαμε και στο προηγούμενο παράδειγμα, μπορούμε να περιγράψουμε και συνδυαστική λογική με το always.

Όμως, το always είναι επίσης ο μόνος τρόπος για να περιγράψουμε ακολουθιακή λογική, δηλαδή latches και flip-flops.

Το always block της Verilog & τρόποι χρήσης (5/6)

```
module d_ff (reset, clk, d, q);
   input reset, clk, d;
   output reg q;

   always @(posedge clk or negedge reset) begin
      if (!reset) q <= 0;  // Reset D FF on negative edge (this is asynchronous!!!)
      else q <= d;         // It's asynchronous because it's in the sensitivity list
   end

endmodule

module d_latch (reset, clk, d, q);
   input reset, clk, d;
   output reg q;

   always @(clk) begin
      if (reset) q <= 0;   // Reset D latch when reset=1, synchronized with the clk!
      else q <= d;         // Notice that the reset is NOT in the sensitivity list!
   end

endmodule
```

Το always block της Verilog & τρόποι χρήσης (6/6)

Είδαμε ότι σε always block έχουμε δύο (2) ειδών αναθέσεις:

“<=” → Non-blocking assignment, η οποία δε χρειάζεται να περιμένει τη προηγούμενη ανάθεση να εκτελέσει ώστε να ξεκινήσει, οπότε θα τρέξουν αυτές οι αναθέσεις παράλληλα.

”=” → Blocking assignment, η οποία αναγκαστικά περιμένει τη προηγούμενη ανάθεση να εκτελέσει ώστε να ξεκινήσει, οπότε είναι σειριακή ανάθεση!

Καλές πρακτικές σχεδίασης με always block (1/2)

Γενικώς, ακολουθήστε τις παρακάτω οδηγίες κωδικοποίησης για να εγγυηθείτε όσο το δυνατόν περισσότερο την σωστή λειτουργία των συστημάτων σας.

- **Οδηγία #1**: Ακολουθιακά κυκλώματα ή/και μανδαλωτές → χρησιμοποιήστε *non-blocking assignments*.
- **Οδηγία #2**: Συνδυαστικά κυκλώματα εντός ενός always block → χρησιμοποιήστε *blocking assignments*.
- **Οδηγία #3**: Ακολουθιακά ΚΑΙ συνδυαστικά εντός του ίδιου always block → χρησιμοποιήστε *non-blocking assignments*.

Καλές πρακτικές σχεδίασης με always block (2/2)

Γενικώς, ακολουθήστε τις παρακάτω οδηγίες κωδικοποίησης για να εγγυηθείτε όσο το δυνατόν περισσότερο την σωστή λειτουργία των συστημάτων σας.

- **Οδηγία #4**: Αν και γίνεται, γενικώς ***ΜΗΝ*** αναμειγνύετε blocking και non-blocking assignments στο ίδιο always!
- **Οδηγία #5**: Μη κάνετε αναθέσεις στην ίδια μεταβλητή σε παραπάνω του ενός always block.
- **Οδηγία #6**: Στη περίπτωση ακολουθιακών κυκλωμάτων, συνιστάται η χρήση μόνο flip-flop και ενός μοναδικού ρολογιού.

Καταχωρητές (1/7)

Τα flip-flop (και οι μανδαλωτές βεβαίως) αποθηκεύουν 1-bit δεδομένα, οπότε ενώνοντας πολλές τέτοιες συσκευές μπορούμε να δημιουργήσουμε μεγαλύτερα στοιχεία μνήμης τα οποία ονομάζουμε **καταχωρητές**.

Συνεπώς, έχει νόημα να εξετάσουμε το πώς διαβάζουμε αλλά κυρίως το πώς γράφουμε δεδομένα στους καταχωρητές μας.

Καταχωρητές (2/7)

Παράδειγμα: Καταχωρητής Ολίσθησης (SIPO / SISO)

```
module shift_register (reset, clk, load, serial_in, reg_out, serial_out);
   parameter N = 8;

   input reset, clk, load, serial_in;
   output reg [N-1:0] reg_out;
   output serial_out;

   // Asynchronous reset shift register with
   // a load enable signal attached to it
   always @(posedge clk or posedge reset) begin
      if (reset) reg_out <= 0;
      else if (load) reg_out <= {serial_in, reg_out[N-1:1]};
   end

   assign serial_out = reg_out[0];

endmodule
```

Καταχωρητές (3/7)

Παράδειγμα: Καταχωρητής Παράλληλης Φόρτωσης

```
module pipo_register (reset, clk, load, parallel_in, reg_out);
   parameter N = 8;

   input reset, clk, load;
   input [N-1:0] parallel_in;
   output reg [N-1:0] reg_out;

   // Asynchronous reset PIPO (Parallel-in, Parallel-Out)
   // register with a load enable signal attached to it
   always @(posedge clk or posedge reset) begin
      if (reset) reg_out <= 0;
      else if (load) reg_out <= parallel_in;
   end

endmodule
```

Καταχωρητές (4/7)

Παράδειγμα: Καταχωρητής Ολίσθησης Αριστερά/Δεξιά

```
module left_right_shift_register (reset, clk, load, left_right, serial_in, reg_out);
   parameter N = 8;

   input reset, clk, load, left_right, serial_in;
   output reg [N-1:0] reg_out;

   // Asynchronous reset bidirectional (why?) SIPO shift
   // register with a load enable signal attached to it
   always @(posedge clk or posedge reset) begin
      if (reset) reg_out <= 0;
      else if (load) begin
        if (left_right) reg_out <= {reg_out[N-2:0], serial_in};
        else reg_out <= {serial_in, reg_out[N-1:1]};
      end
   end

endmodule
```

Καταχωρητές (5/7)

Παράδειγμα: Καταχωρητής Πολλαπλών Λειτουργιών

```
module multimode_shift_register (reset, clk, load, mode, parallel_in, reg_out);
   parameter N = 8;

   input [N-1:0] parallel_in;
   input [1:0] mode;
   input reset, clk, load, mode;
   output reg [N-1:0] reg_out;

   // Register with circular shift and parallel loading capabilities
   always @(posedge clk or posedge reset) begin
      if (reset) reg_out <= 0;
      else if (load) begin
        if (mode == 2'b11) reg_out <= parallel_in;
        else if (mode == 2'b10) reg_out <= {reg_out[N-2:0], reg_out[N-1]};
        else if (mode == 2'b01) reg_out <= {reg_out[0], reg_out[N-1:1]};
      end
   end

endmodule
```

Καταχωρητές (6/7)

Ωραία όλα αυτά τα κυκλώματα, προφανώς, και αρκετά χρήσιμα. Πάμε όμως λίγο να εξετάσουμε τι μπορούμε να κάνουμε με την ανατροφοδότηση.

Καταχωρητές (7/7)

Παράδειγμα: Καταχωρητής / Μετρητής Johnson

```
module johnson_ring_counter (reset, clk, reg_out);
   parameter N = 8;

   input reset, clk;
   output reg [N-1:0] reg_out;

   // Feeding back the complement of the least-significant bit
   always @(posedge clk or posedge reset) begin
      if (reset) reg_out <= 0;
      else reg_out <= {~reg_out[0], reg_out[N-1:1]};
   end

endmodule
```

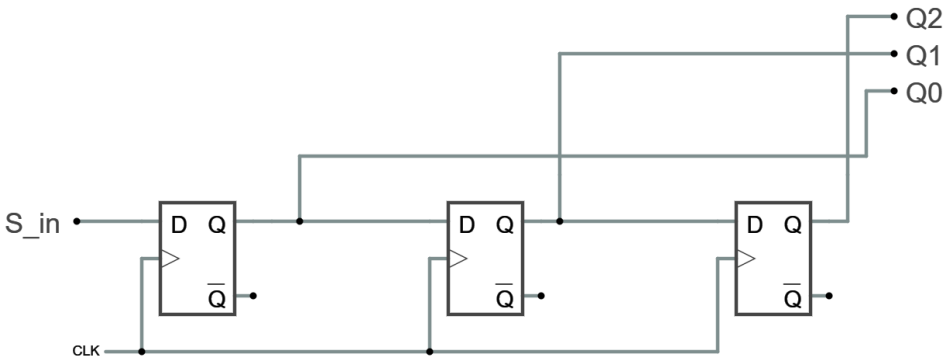
Μετρητές (1/10)

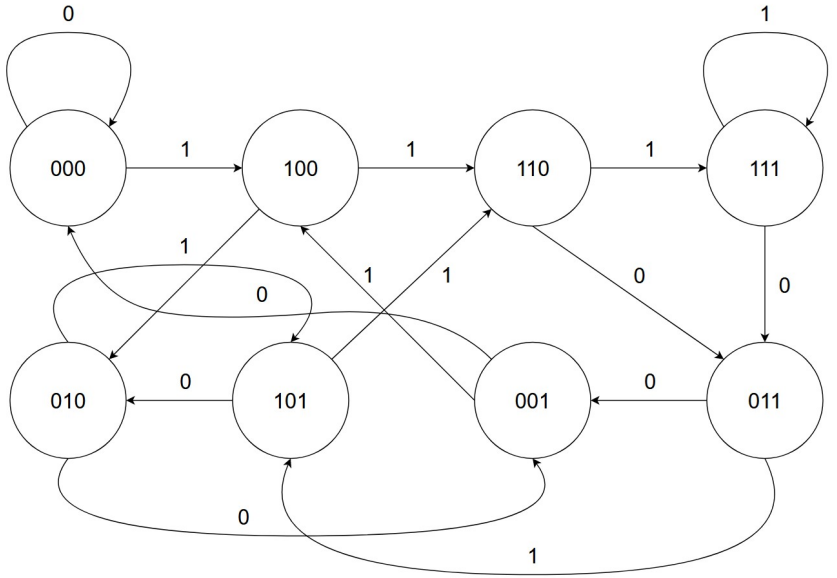
Μόλις δημιουργήσαμε ένα είδους μετρητή από έναν απλό καταχωρητή ολίσθησης! Αλλά υπάρχει ένα μικρό πρόβλημα εδώ... Στο μετρητή Johnson χρησιμοποιούμε 2·N καταστάσεις από τις διαθέσιμες 2^N!

Βέβαια όμως, η υλοποίησή του στο υλικό είναι σαφώς πολύ πιο απλή διαδικασία. Υπάρχει μήπως κάπου συσχέτιση μεταξύ της υλοποίησης ως FSM και της υλοποίησης στο επίπεδο του υλικού;

Μετρητές (2/10)

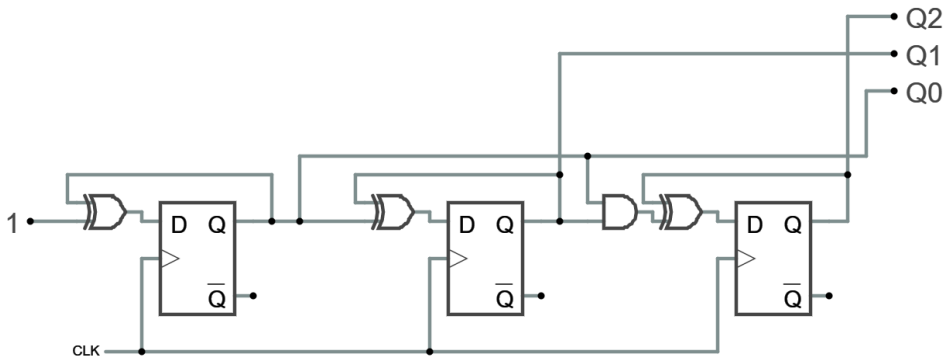
Παράδειγμα: Καταχωρητής ολίσθησης των 3-bit

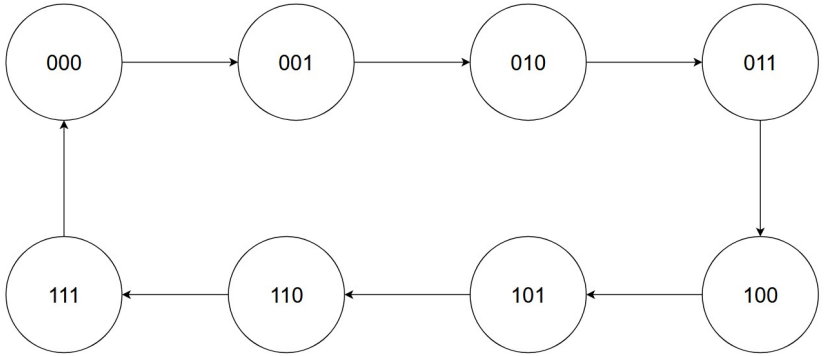




Μετρητές (3/10)

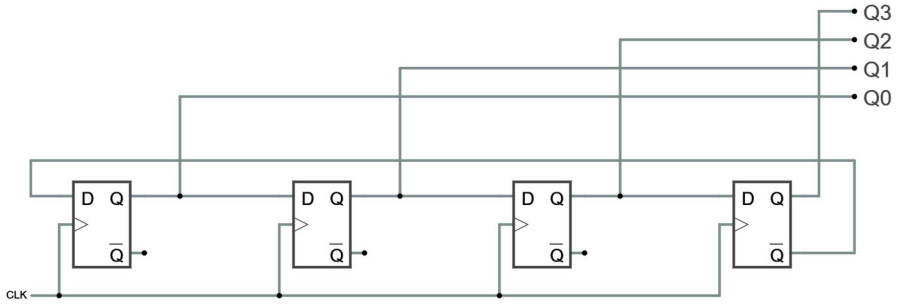
Παράδειγμα: Μετρητής “προς τα πάνω” των 3-bit

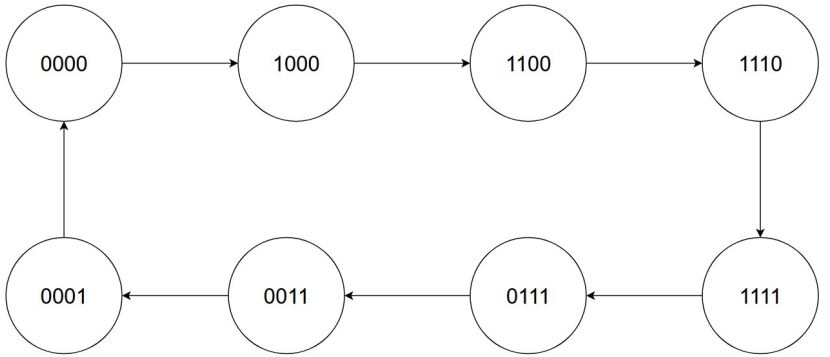




Μετρητές (4/10)

Παράδειγμα: Μετρητής Johnson των 4-bit





Μετρητές (5/10)

Εύκολα μπορεί να καταλάβει κανείς ότι μπορεί να γίνει μέτρηση με αρκετούς διαφορετικούς τρόπους στο επίπεδο του υλικού. Οι κυριότερες ιδέες όμως είναι:

- Με πυροδότηση (toggling) ανά παλμό
- Ολίσθηση και ανατροφοδότηση ψηφιολέξης
- Συνδυασμός αυτών των δύο μεθόδων

Τι γίνεται όμως αν θέλουμε παραπάνω ευχρηστία;

Μετρητές (6/10)

Ας δούμε πρώτα το βασικότερο μετρητή και μετά να του προσθέσουμε λειτουργίες!
Παράδειγμα: Δυαδικός Μετρητής

```
module binary_counter (reset, clk, count);
   parameter N = 8;

   input reset, clk;
   output reg [N-1:0] count;

   always @(posedge clk or posedge reset) begin
      if (reset) count <= 0;
      else count <= count + 1;
   end

endmodule
```

Μετρητές (7/10)

Παράδειγμα: Modulo Μετρητής

```
module modulo_counter (reset, clk, count, modulo);
   parameter N = 8;
   parameter MODULO = 159;

   input reset, clk;
   output reg [N-1:0] count;
   output modulo;

   assign modulo = (count == MODULO);

   // We also want to reset (synchronously here!)
   // when we reach the desired modulus
   always @(posedge clk or posedge reset) begin
      if (reset) count <= 0;
      else if (modulo) count <= 0;
      else count <= count + 1;
   end

endmodule
```

Μετρητές (8/10)

Παράδειγμα: Δυαδικός Μετρητής Παράλληλης Φόρτωσης

```
module binary_counter_parallel_load (reset, clk, load, parallel_in, count);
   parameter N = 8;

   input reset, clk, load;
   input [N-1:0] parallel_in;
   output reg [N-1:0] count;

   always @(posedge clk or posedge reset) begin
      if (reset) count <= 0;
      else if (load) count <= parallel_in;
      else count <= count + 1;
   end

endmodule
```

Μετρητές (9/10)

Παράδειγμα: Δυαδικός Μετρητής “Προς τα Πάνω/Κάτω”

```
module up_down_counter (reset, clk, up_down, count);
   parameter N = 8;

   input reset, clk, up_down;
   output reg [N-1:0] count;

   // When up_down = 1 → count upwards, otherwise
   // we count downwards
   always @(posedge clk or posedge reset) begin
      if (reset) count <= 0;
      else if (up_down) count <= count + 1;
      else count <= count - 1;
   end

endmodule
```

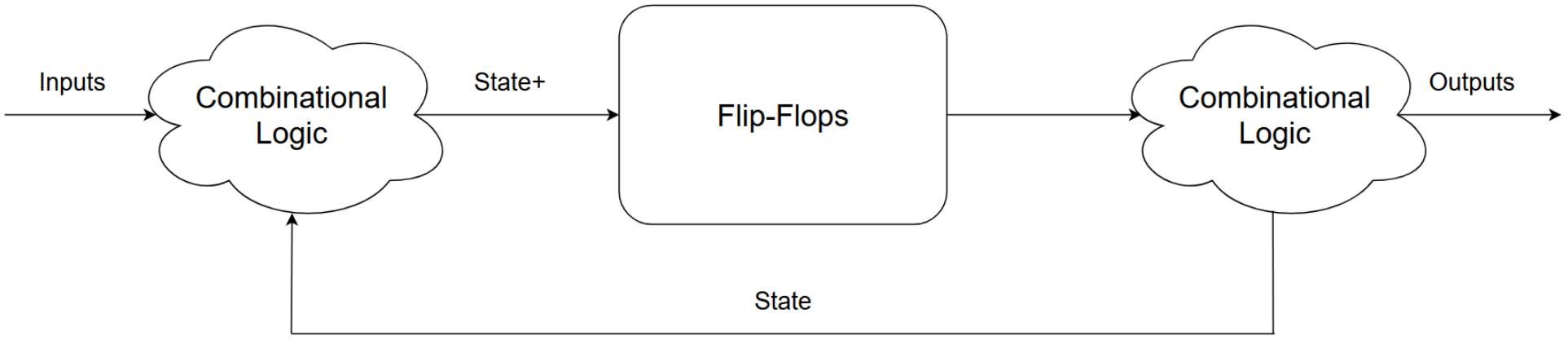
Μετρητές (10/10)

Μια πολύ χρήσιμη εφαρμογή των μετρητών είναι η διαίρεση συχνότητας, μέσω της σχέσης:
$f_{\text{εξόδου}} = (f_{\text{εισόδου}}) / (\text{Καταστάσεις Μετρητή})$

Αντί για μεγάλους μετρητές, συχνά χρησιμοποιούνται μικρότεροι που η εξάντληση της ακολουθίας μέτρησης (μηδενισμός) του ενός θα είναι η επίτρεψη μέτρησης του επόμενου, ώστε το κρίσιμο μονοπάτι να μείνει μικρό → η ακρίβεια διαίρεσης συχνότητας διατηρείται!

FSMs στη Verilog (1/12)

Ας δούμε πάλι το παράδειγμα με το οποίο “ανοίξαμε” αρχικά, το οποίο ήταν η αναγνώριση της “λέξης” 1101 μέσα σε μια συμβολοσειρά. Η συνδυαστική και ακολουθιακή λογική μας από αφαιρετικό επίπεδο αφαιρετικά θα μοιάζει ως εξής:



FSMs στη Verilog (2/12)

Γενικώς, είναι πάντα καλή ιδέα να χωρίζουμε όλα μας τα προβλήματα σε μικρότερα και πιο απλά. Επομένως, θα περιγράψουμε το ζητούμενο κύκλωμα με δύο (2) always blocks: ένα (1) για το καταχωρητή επόμενης κατάστασης (**ακολουθιακό τμήμα**) και ένα (1) για τη λογική υπολογισμού της επόμενης κατάστασης και των εξόδων (**συνδυαστικό τμήμα**).

Διαίρει και βασίλευε!

FSMs στη Verilog (3/12)

```
module pattern_recognizer_moore (reset,
clk, string_in, seen);
   localparam [3:0] S0 = 4'b0000,
                    S1 = 4'b0001,
                    S2 = 4'b0010,
                    S3 = 4'b0100,
                    S4 = 4'b1000;

   input reset, clk, string_in;
   output seen;
   reg [3:0] state, next;

   // Sequential always block
   // for the state register
   always @(posedge clk or posedge reset)
   begin
      if (reset) begin
         state <= 4'b0000;
      end
      else begin
         state <= next;
      end
   end
```

```
   // Combinational always block, so
   // we'll use blocking statements here
   always @(*) begin
   next = 4'bx;
   seen = 1'b0;
   case (state)
      S0:
      begin
         if (string_in) next = S1;
         else next = S0;
      end
      S1:
      begin
         if (string_in) next = S2;
         else next = S0;
      end
      S2:
      begin
         if (!string_in) next = S3;
         else next = S2;
      end
```

```
      S3:
      begin
         if (string_in) next = S4;
         else next = S0;
      end
      S4:
      begin
         seen = 1'b1;
         if (string_in) next = S1;
         else next = S0;
      end
      default: next = state;
   endcase
end

endmodule
```

One-hot προσέγγιση για κωδικοποίηση καταστάσεων: Χαμηλή κατανάλωση ενέργειας λόγω (συνήθως) το πολύ εναλλαγή δύο bit. Πολύ καλό για ταχύτητα, ιδιαίτερα για FPGAs, αλλά παραπάνω flip-flops → παραπάνω απαιτούμενος χώρος & παραπάνω "παράνομες" καταστάσεις.

FSMs στη Verilog (4/12)

```
module pattern_recognizer_mealy (reset,
clk, string_in, seen);
   localparam [1:0] S0 = 2'b00,
                    S1 = 2'b01,
                    S2 = 2'b11,
                    S3 = 2'b10;

   input reset, clk, string_in;
   output seen;
   reg [1:0] state, next;

   // Sequential always block
   // for the state register
   always @(posedge clk or posedge reset)
   begin
      if (reset) begin
         state <= 2'b00;
      end
      else begin
         state <= next;
      end
   end
```

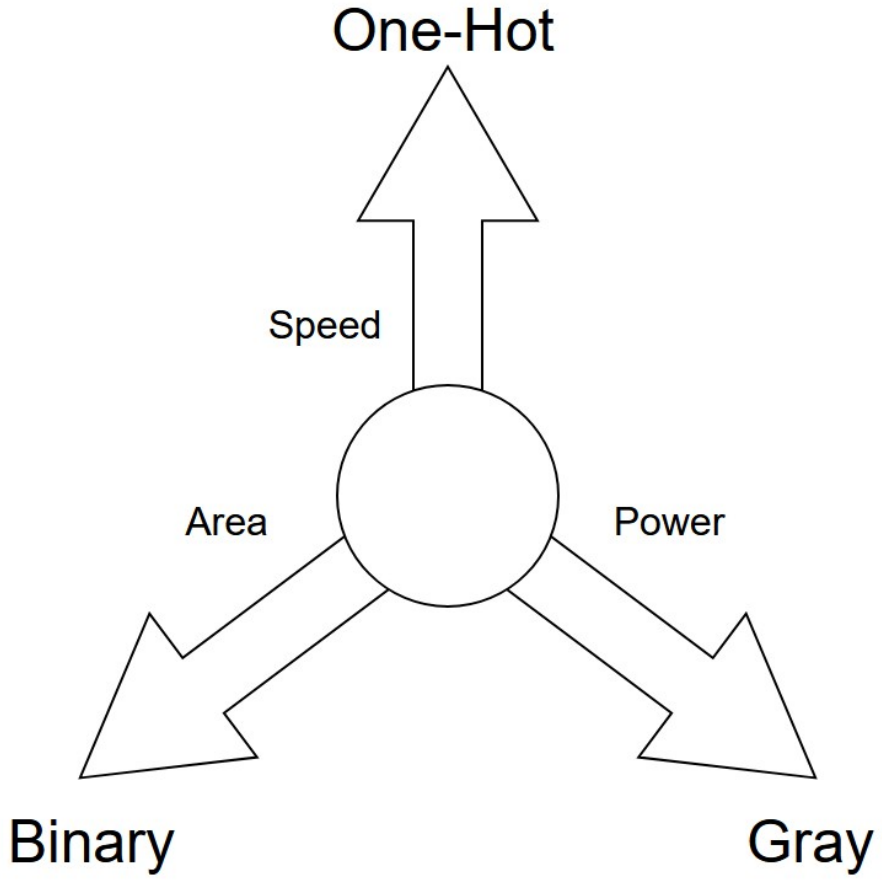
```
   // Combinational always block, so
   // we'll use blocking statements here
   always @(*) begin
   next = 2'bx;
   seen = 1'b0;
   case (state)
      S0:
      begin
         if (string_in) next = S1;
         else next = S0;
      end
      S1:
      begin
         if (string_in) next = S2;
         else next = S0;
      end
      S2:
      begin
         if (string_in) next = S3;
         else next = S2;
      end
      S3:
      begin
         if (string_in) begin
            seen = 1'b1;
            next = S1;
         else next = S0;
      end
   endcase
end

endmodule
```

Gray-code προσέγγιση για κωδικοποίηση καταστάσεων. Όσο λιγότερα bit αλλάζουν, θα έχουμε χαμηλότερη κατανάλωση ενέργειας και λιγότερο θόρυβο λόγω της εναλλαγής μόνο ενός bit, ενώ επίσης χρησιμοποιούμε το ελάχιστο πλήθος bit που απαιτείται για τις καταστάσεις μας.

FSMs στη Verilog (5/12)

Τρόποι κωδικοποίησης καταστάσεων και trade-offs:



FSMs στη Verilog (6/12)

Αξίζει όμως να κάνουμε όλη αυτή τη διαδικασία *πάντα* όταν θέλουμε να σχεδιάσουμε κάποιο ακολουθιακό κύκλωμα που περιγράφεται με μηχανή πεπερασμένων καταστάσεων;

Σίγουρα όχι! Πάντα υπάρχουν και άλλες ισοδύναμες ιδέες!

FSMs στη Verilog (7/12)

```
module pattern_recognizer_s2p_comparator (reset, clk, string_in, seen);
   input reset, clk, string_in;
   output seen;

   reg [3:0] serial_to_parallel;

   // A simple shift register...
   always @(posedge clk or posedge reset) begin
      if (reset) serial_to_parallel <= 4'b0000;
      else serial_to_parallel <= {serial_to_parallel[2:0], string_in};
   end

   // ...and a comparator!
   assign seen = (serial_to_parallel == 4'b1101);

endmodule
```

FSMs στη Verilog (8/12)

Υλοποίηση με FSM:
+ λιγότερα FFs (gray/binary)
- πιο σύνθετη λογική

Η περίοδος του ρολογιού:

$T_{clk} \geq \max \{Q + control\_logic + setup,$
$Q + output\_logic\}$

Συνήθως το 1ο εκ των δύο υπερισχύει όμως

Υλοποίηση με σύγκριση:
+ απλούστερη λογική
- παραπάνω υλικό γενικώς

Η περίοδος του ρολογιού:

$T_{clk} \geq Q + output\_logic$

Όπου εδώ θα ισχύει ότι output_logic → AND4

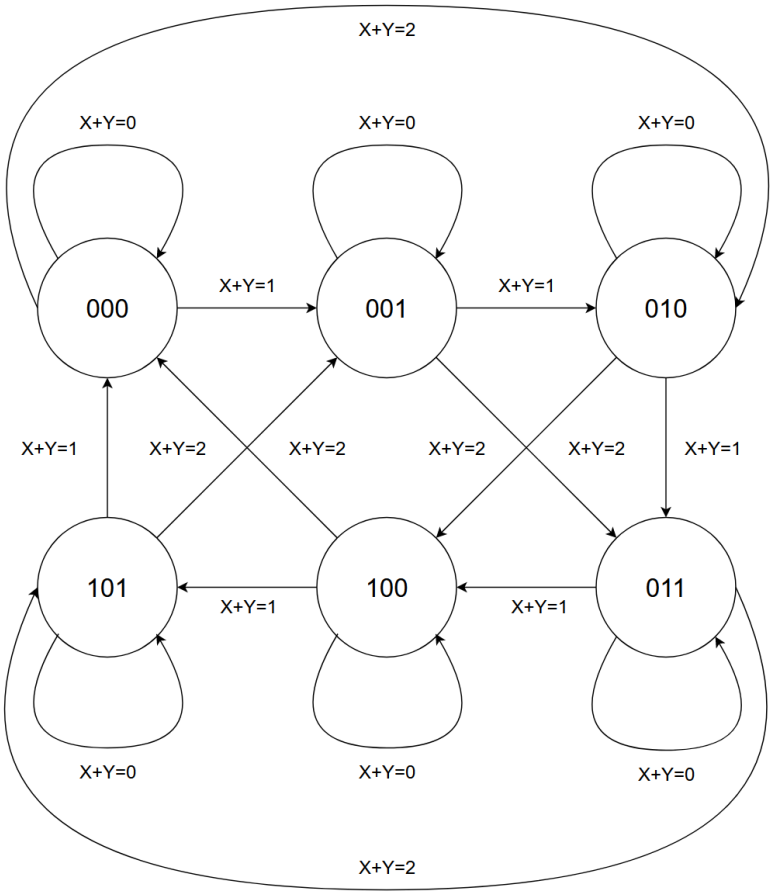
FSMs στη Verilog (9/12)

Ένα λίγο πιο σύνθετο παράδειγμα που θα συνδυαστούν όλα όσα έχουμε δει, όπως και καινούργιες λειτουργίες.

Έστω ακολουθιακό κύκλωμα το οποίο υπολογίζει αν το άθροισμα δύο αριθμών X και Y, οι οποίοι εισάγονται σειριακά αρχίζοντας από το λιγότερο σημαντικό bit, διαιρείται με το 6. Η κωδικοποίηση των καταστάσεων να γίνει με το απλό δυαδικό σύστημα αρίθμησης.

FSMs στη Verilog (10/12)

Διάγραμμα καταστάσεων που λύνει το πρόβλημα:



FSMs στη Verilog (11/12)

```
module divisable_by_six (reset, clk, X_in, Y_in,
divisable);
   input reset, clk, X_in, Y_in;
   output divisable;
   reg [2:0] state, next;

   always @(posedge clk or posedge reset) begin
      if (reset) state <= 3'b000;
      else state <= next;
   end

   always @(*) begin
   next = 3'bx;
   case (state)
      3'b100:
      begin
         if (X_in & Y_in) next = 3'b000;
         else next = state + X_in + Y_in;
      end
      3'b101:
      begin
         if (X_in & Y_in) next = 3'b001;
         else if (X_in | Y_in) next = 3'b000;
         else next = state;
      end
      // Heavy use of default case, because
      // with it we can take full advantage
      // of the symmetry of our FSM's design
      default: next = state + X_in + Y_in;
   endcase
end

// NOR reduction operator
assign divisable = ~|state;

endmodule
```

FSMs στη Verilog (12/12)

Μια τελευταία λεπτομέρεια σχετικά με τη λέξη-κλειδί case. Πέραν της χρησιμότητας του default, ***ΠΡΕΠΕΙ*** να το χρησιμοποιείτε αν δεν έχετε γράψει ρητά όλες σας τις περιπτώσεις! Γιατί όμως;

ΓΙΑΤΙ ΑΛΛΙΩΣ ΔΗΜΙΟΥΡΓΕΙ ΑΝΕΠΙΘΥΜΗΤΑ LATCH!!!

Αν η λογική επόμενης κατάστασης μείνει απροσδιόριστη σε ορισμένα case/if-else (και εδώ ισχύει αυτό!) statements → Διατήρηση τιμής → **ΥΠΟΝΟΕΙ ΚΑΤΑΣΤΑΣΗ ΑΠΟΘΗΚΕΥΣΗΣ**

Βιβλιογραφία

- Introduction to the Theory of Computation, Michael Sipser, 3rd Edition
- Digital Design: Principles and Practices, John F. Wakerly, 5th Edition
- CMOS VLSI Design: A Circuits and Systems Perspective, Neil H. E. Weste, David M. Harris, 4th Edition
- FPGA Prototyping by Verilog Examples: Xilinx Spartan-3 Version, Pong P. Chu
- Σχεδίαση Συστημάτων με Χρήση Υπολογιστών (E-CAD), Πανεπιστήμιο Πατρών, Χαρίδημος Βέργος
- Introduction to Verilog HDL, Synopsys Courseware, Jorge Ramírez