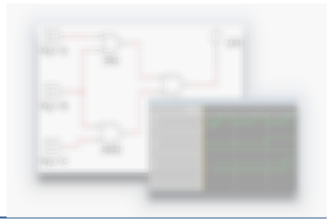


ECE119 – Ψηφιακή Σχεδίαση

Διδάσκοντες Εργαστηρίου: Δ. Καραμπερόπουλος
Α. Φεύγας

- Lab 8: Verilog (Μέρος 4)

Η Γλώσσα Verilog (Μέρος 4)



Μοντέλα περιγραφής κυκλωμάτων σε HDL



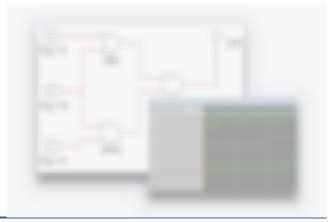
- **Μοντελοποίηση σε επίπεδο πυλών** (**gate-level modeling**), όπου χρησιμοποιούνται στιγμιότυπα είτε προκαθορισμένων, ή καθορισμένων από το χρήστη βασικών πυλών.
- **Μοντελοποίηση ροής δεδομένων** (**dataflow modeling**), όπου χρησιμοποιούνται εντολές διαρκούς ανάθεσης με την δεσμευμένη λέξη **assign**.
- **Μοντελοποίηση συμπεριφοράς** (**behavioral modeling**), όπου χρησιμοποιούνται διαδικαστικές εντολές ανάθεσης με την δεσμευμένη λέξη **always**.

Μοντελοποίηση συμπεριφοράς (behavioral modeling),



- Περιγράφει κυκλώματα σε **αλγοριθμικό** και **λειτουργικό** επίπεδο.
- Χρησιμοποιείται για την περιγραφή τόσο συνδυαστικών όσο και ακολουθιακών κυκλωμάτων, με ιδιαίτερη χρησιμότητα στην περιγραφή της συμπεριφοράς **στοιχείων μνήμης**.
- Περιγράφει τι κάνει ένα στοιχείο και όχι πως το κάνει.
- Συντίθεται σε ένα κύκλωμα που έχει αυτή τη συμπεριφορά.
- Χρησιμοποιεί το μπλοκ “**always**”

Always Block

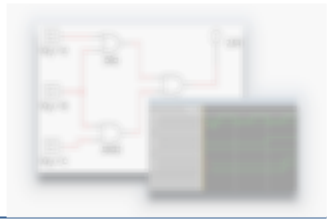


- Η γενική μορφή είναι:

```
always @ (έκφραση ελέγχου συμβάντων)
   begin
     // Διαδικαστικές εντολές που εκτελούνται όταν ικανοποιείται η ελεγχόμενη συνθήκη.
   end
```

- Η **έκφραση ελέγχου συμβάντων** (**λίστα ευαισθησίας**) είναι μία συνθήκη η οποία καθορίζει πότε θα αρχίσουν να εκτελούνται οι εντολές του μπλοκ.
- Οι **διαδικαστικές εντολές** εκτελούνται με τη σειρά που εμφανίζονται στο μπλοκ. Ωστόσο, στις μη ανασταλτικές αναθέσεις (<=), η ενημέρωση των μεταβλητών προγραμματίζεται για μεταγενέστερη φάση της ίδιας χρονικής στιγμής προσομοίωσης.

Always Block

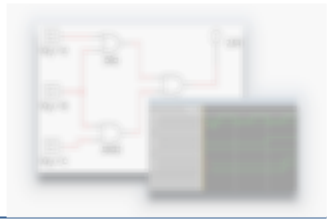


- Η γενική μορφή είναι:

```
always @ (έκφραση ελέγχου συμβάντων)
   begin
     // Διαδικαστικές εντολές που εκτελούνται όταν ικανοποιείται η ελεγχόμενη συνθήκη.
   end
```

- Οι εντολές του μπλοκ περικλείονται μεταξύ των **begin** και **end** και μπορούν να είναι:
  - Διαδικαστικές εντολές ανάθεσης.
  - Εντολές **if, case, for, while, repeat, forever**
- Εάν υπάρχει μόνο μία εντολή δεν χρειάζεται begin/end.

Always Block – Εντολές Ανάθεσης

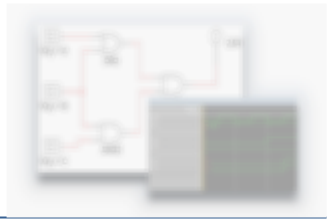


- Στη Verilog έχουμε 2 τύπους εντολών ανάθεσης:
  - **Διαρκείς εντολές ανάθεσης** (**έξω από μπλοκ** always ή initial)

    Έχουν υπονοούμενη λίστα ευαισθησίας που αφορά στις αλλαγές στάθμης σημάτων (Η λίστα ευαισθησίας περιλαμβάνει όλες τις μεταβλητές που βρίσκονται στο δεξιό τμήμα της)
    - **assign**
  - **Διαδικαστικές εντολές ανάθεσης** (**μέσα σε μπλοκ** always ή initial)

    Εκτελούνται όταν θα εκτελεστεί το μπλοκ που τις περιλαμβάνει
    - **=** **Blocking assignment**, ανασταλτική ανάθεση
      Ακολουθιακή εκτέλεση (Άμεση ενημέρωση της μεταβλητής πριν από την επόμενη εντολή)
    - **<=** **Non-blocking assignment**, μη ανασταλτική ανάθεση
      Παράλληλη-ταυτόχρονη εκτέλεση
      (Προγραμματισμένη ενημέρωση της μεταβλητής, χωρίς αναμονή ολοκλήρωσης της ανάθεσης)

Always Block



always @ (event list)

Καθορίζει πότε θα εκτελεστεί το always statement

↓

Απαιτεί αλλαγή τιμής ενός στοιχείου της λίστας

- Παρατηρήσεις:

Η ανάθεση γίνεται υποχρεωτικά σε μεταβλητές τύπου **reg**

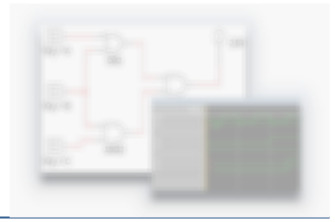
↓

Δεν ανανεώνονται αυτόματα αλλά κρατούν την τιμή τους μέχρι να επανεκτελεστεί το **always**

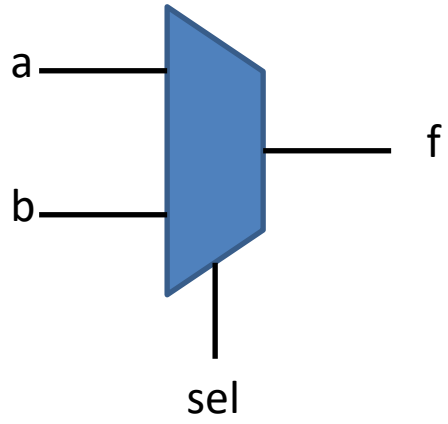
Η δήλωση **reg** δεν συνεπάγεται απαραίτητα υλοποίηση στοιχείου μνήμης. Το κύκλωμα που συντίθεται εξαρτάται από τη συμπεριφορά που περιγράφεται.

Always Block

Στη Verilog 2001 μπορεί να χρησιμοποιηθεί το **always @(*)**, ώστε η λίστα να συμπληρώνεται αυτόματα.



Πολυπλέκτης 2 σε1



a	b	sel	f
x	y	0	x
x	y	I	y

```
module mux2_1 (a, b, sel, f);
input  a, b, sel;
output f;
reg f;

always @(a or b or sel)
begin
  if (sel==0)
    f = a;
  else
    f = b;
end
endmodule
```

Λίστα ευαισθησίας

Έξοδος τύπου **reg**

Κατά την περιγραφή **συνδυαστικής λογικής**:
- Χρησιμοποιούμε blocking assignments
- Όλα τα inputs στην λίστα ευαισθησίας

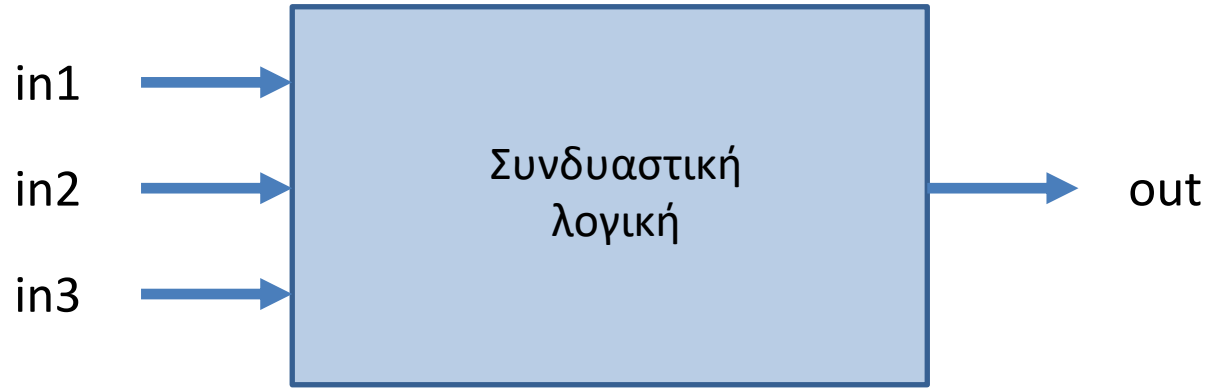
Στις απλές συνδυαστικές συναρτήσεις, η περιγραφή με **assign** είναι συχνά πιο σύντομη

```
module mux2_1 (a, b, sel, f);
input  a, b, sel;
output f;
assign f = (sel == 0) ? a : b;
endmodule
```

Always Block

Στην πρώτη περιγραφή η έξοδος δηλώνεται ως **reg**, επειδή ενημερώνεται μέσα σε **always**. Στη δεύτερη περιγραφή η έξοδος είναι τύπου **wire**, επειδή οδηγείται από συνεχή ανάθεση (**assign**).





Έξοδος τύπου **reg**

Λίστα ευαισθησίας

Ισοδύναμοι ορισμοί

```
reg out;

always @(in1 or in2 or in3)
    out = in1 | (in2 & in3);
```

```
wire out;

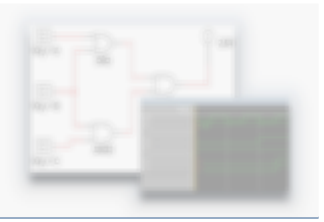
assign out = in1 | (in2 & in3);
```

Κατά την περιγραφή **συνδυαστικής λογικής**:

- Χρησιμοποιούμε blocking assignments
- Όλα τα inputs στην λίστα ευαισθησίας

Η μοντελοποίηση ροής δεδομένων είναι **προτιμότερη** στα συνδυαστικά κυκλώματα

Τμήματα always και initial



Τμήμα **initial**	Τμήμα **always**

`initial`
`begin`
`  // run once`
`  a=0; b=0;`
`  #5; a=1; b=1;`
`end`
	`always @(b or c)`
	`begin`
	`  // run always`
	`  a = b & c;`
	`end`

▸ **Εκτελείται** μια φορά, στην εκκίνηση της προσομοίωσης ▸ **Δεν επαναλαμβάνεται** ▸ Χρησιμοποιείται κυρίως στην προσομοίωση και στα **testbenches** (για να παρέχει διανύσματα εισόδου στο κύκλωμα)	▸ **Ενεργοποιείται** στην αρχή της προσομοίωσης και εκτελεί τις εντολές του κάθε φορά που συμβαίνει ένα από τα γεγονότα της λίστας ευαισθησίας ▸ **Επαναλαμβάνεται** σε κάθε ενεργοποίηση της λίστας ευαισθησίας ▸ Χρησιμοποιείται για να περιγράφει διαρκή και μόνιμη συμπεριφορά **(συνδυαστική ή ακολουθιακή)**

Συνθήκη if/else

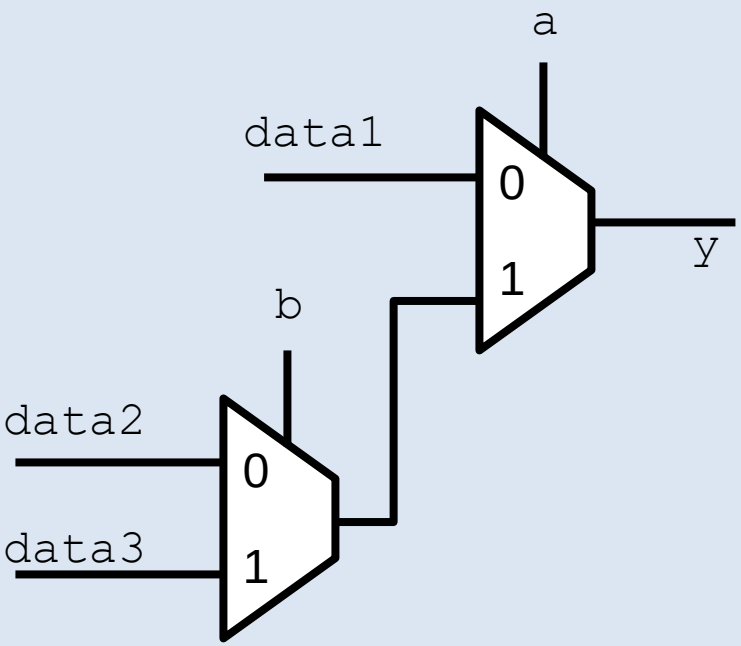
- Μόνο μέσα σε διαδικαστικά μπλοκ, όπως **always** και **initial**
- Αν η συνθήκη περικλείει πολλαπλές εντολές χρησιμοποιείται **begin/end**
- Επιτρέπονται
  - πολλαπλά **else if**
  - **if** μέσα σε **if**

Περιγραφή Verilog	Κυκλωματική Μορφή

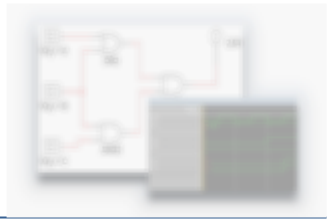
```
always @(a or b or data1 or data2 or data3)
begin

  if (a == 0)
    y = data1;
  else begin
    if (b == 0)
      y = data2;
    else
      y = data3;
  end

end
```



Συνθήκη if/else

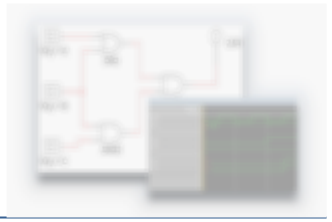


```
// Simple 4-1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel;                    // 2-bit control signal
input A, B, C, D;
output Y;
reg Y;                              // target of assignment

  always @(sel or A or B or C or D)
    if      (sel == 2'b00) Y = A;
    else if (sel == 2'b01) Y = B;
    else if (sel == 2'b10) Y = C;
    else if (sel == 2'b11) Y = D;       // ή else Y = D;

endmodule
```

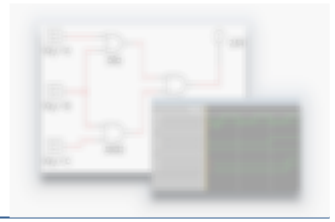
Συνθήκη if/else



```
// Simple 4-1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel;                    // 2-bit control signal
input A, B, C, D;
output Y;
reg Y;                              // target of assignment

  always @(sel or A or B or C or D)
    if (sel[1] == 0)
      if (sel[0] == 0) Y = A;
      else             Y = B;
    else
      if (sel[0] == 0) Y = C;
      else             Y = D;
endmodule
```

Συνθήκη case

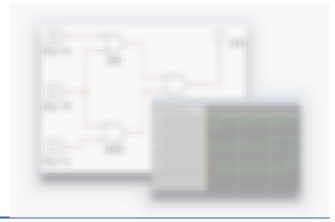


- Μόνο μέσα σε διαδικαστικά μπλοκ, όπως **always** και **initial**
- Μόνο για σταθερές εκφράσεις
- Εκτελούνται οι εντολές της πρώτης περίπτωσης που ταιριάζει. Δεν απαιτείται break
- Μπορεί να χρησιμοποιηθεί Default case.

```
// Simple 4-1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel;              // 2-bit control signal
input A, B, C, D;
output Y;
reg Y;                        // target of assignment

  always @(sel or A or B or C or D)
    case (sel)
      2'b00: Y = A;
      2'b01: Y = B;
      2'b10: Y = C;
      2'b11: Y = D;
    endcase
endmodule
```

Συνθήκη case

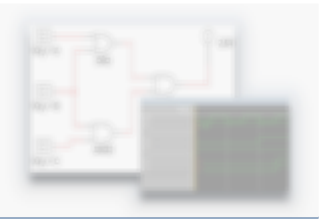


- Η ανάθεση της τιμής **X** δηλώνει άγνωστη τιμή στην προσομοίωση. Κατά τη σύνθεση, μπορεί να αξιοποιηθεί ως don't care, επιτρέποντας στο εργαλείο να επιλέξει τιμές που διευκολύνουν τη βελτιστοποίηση.

```
// Simple binary encoder (input is 1-hot)
module encode (A, Y);
input  [7:0] A;                  // 8-bit input vector
output [2:0] Y;                  // 3-bit encoded output
reg    [2:0] Y;                  // target of assignment

  always @(A)
    case (A)
      8'b00000001: Y = 0;        // 3'd0
      8'b00000010: Y = 1;        // 3'd1
      8'b00000100: Y = 2;        // 3'd2
      8'b00001000: Y = 3;        // 3'd3
      8'b00010000: Y = 4;        // 3'd4
      8'b00100000: Y = 5;        // 3'd5
      8'b01000000: Y = 6;        // 3'd6
      8'b10000000: Y = 7;        // 3'd7
      default:     Y = 3'bX;     // Don't care when input
                                 // is not 1-hot
    endcase
endmodule
```

Βρόχοι for και while

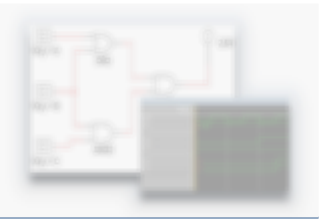


- Μόνο μέσα σε διαδικαστικά μπλοκ, όπως **always** και **initial**
- Λειτουργούν όπως σε παραδοσιακές γλώσσες προγραμματισμού
- Δεν υποστηρίζονται:
  - break, continue
  - i++, i--
- Χρήση:
  - Σε **testbenches** για επαναληπτική εφαρμογή τιμών εισόδου.
  - Σε **συνθέσιμες περιγραφές** για επαναλαμβανόμενες λογικές λειτουργίες (κυρίως for με σταθερά όρια).

```
integer i;
initial begin
  for (i = 0; i < 10; i = i + 1)
     begin
       $display ("i= %d",i);
     end
end
```

```
integer j;
initial begin
  j=0;
  while (j < 10)
     begin
       $display ("j= %b",j);
       j = j + 1;
     end
end
```

Βρόχοι for και while



Module

```
// Truth table
//   a   b   y
//   0   0   0
//   0   1   1
//   1   0   1
//   1   1   0

module gate (y, a, b);
output y;
input  a, b;
assign y = (~a & b) | (a & ~b);
endmodule
```

Test Bench

```
module test;
reg   a, b;
wire  y;
integer i;

gate  DUT (y, a, b);
initial begin
   $dumpfile("dump.vcd");
   $dumpvars;
   for (i=0; i<4; i=i+1)
      begin
         {a,b} = i;
         #5;
      end
   $finish;
end
endmodule
```

Η συνένωση {a,b} αντιμετωπίζεται ως διάνυσμα 2 bits και παίρνει διαδοχικά τις τιμές 00, 01, 10, 11.

Βρόχοι for και while

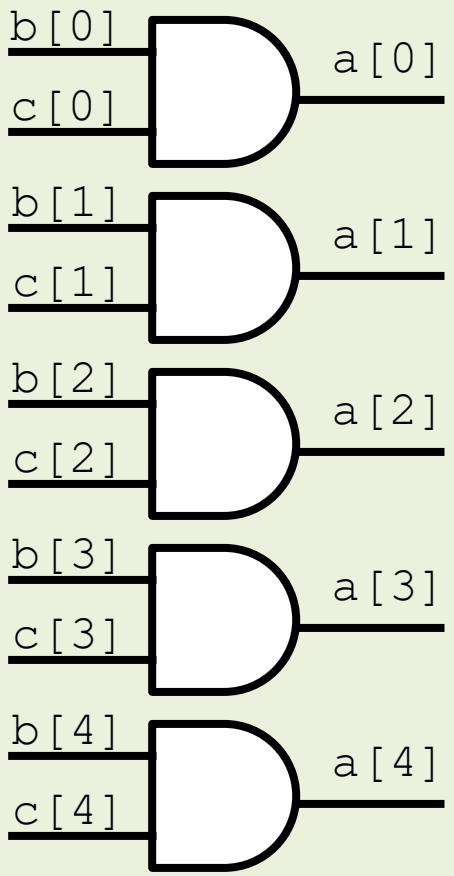
Περιγραφή Verilog	Κυκλωματική Μορφή

```
module example (a, b, c);
output reg [4:0]    a;
input [4:0]         b, c;

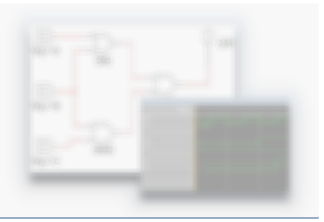
integer i;

always @(b or c) begin
  for (i = 0; i < 5; i = i + 1)
  begin
     a[i] = b[i] & c[i];
  end
end
endmodule
```

Συνθέσιμο



Βρόχοι repeat και forever



- **Repeat** (<number_of_loops>)

```
begin
        statement_1
        …
        statement_n
end
```

```
initial begin
    a = 4'b0000;
    repeat (15) begin
        #10 a = a + 1;
    end
end
```

Execute statements a fixed number of times

- **forever**

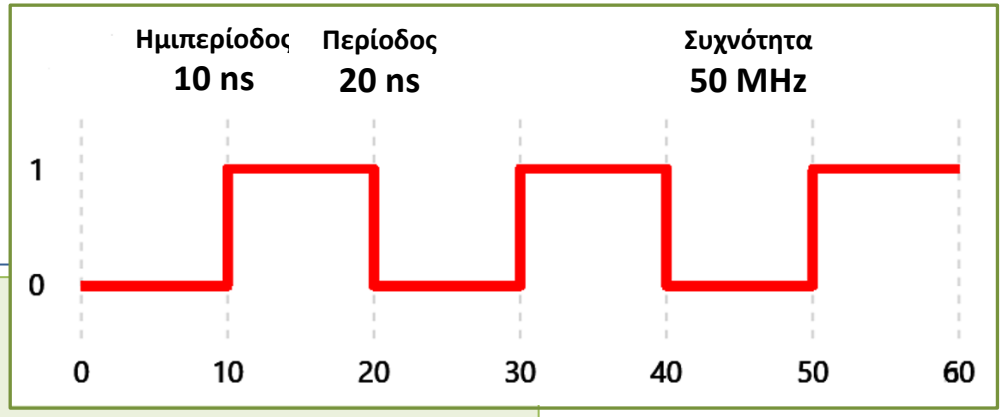
```
begin
        statement_1
        …
        statement_n
end
```

```
initial begin
    a = 4'b0000;
    forever begin
        #10 a = a + 1;
    end
end
```

Execute statements forever

Επειδή η a είναι τεσσάρων bits, μετά την τιμή 1111 επιστρέφει στην 0000 λόγω υπερχείλισης

Περιγραφή σήματος ρολογιού



1ος τρόπος

```
initial
begin
    clock = 1'b0;
    repeat (30) #10 clock = ~clock;
end
```

2ος τρόπος

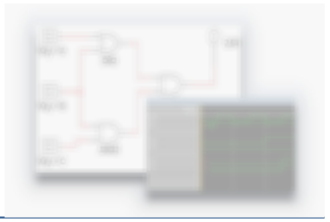
```
initial
begin
    clock = 1'b0;
    forever #10 clock = ~clock;
end
initial #300 $finish;
```

3ος τρόπος

```
initial
begin
    clock = 1'b0;
end
always #10 clock = ~clock;
initial #300 $finish;
```

Το #10 καθορίζει την ημιπερίοδο του ρολογιού. Με timescale 1ns/1ps, η περίοδος είναι 20 ns. Οι παραπάνω περιγραφές χρησιμοποιούνται σε testbenches και όχι για τη σύνθεση φυσικού κυκλώματος παραγωγής clock.

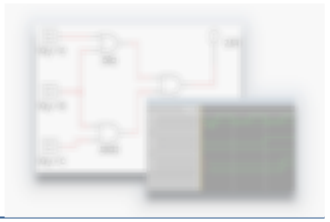
Τελεστές # και @



- **Τελεστής ελέγχου καθυστέρησης #**
  - Επιτάσσει αναμονή παρέλευσης **συγκεκριμένου χρόνου**
    Π.χ. `#10 a = 1'b1`
  - Χρησιμοποιείται συχνά σε περιγραφές εφάπαξ συμπεριφοράς (initial)

- **Τελεστής ελέγχου συμβάντων @**
  - Επιτάσσει αναστολή στην εκτέλεση της διαδικασίας μέχρι να πραγματοποιηθεί ένα από τα **καθορισμένα συμβάντα**, όπως μεταβολή τιμής σήματος ή θετική/αρνητική ακμή.
    Π.χ. `@(A or B),  @(A)`
  - Χρησιμοποιείται στην always

Λίστα ευαισθησίας



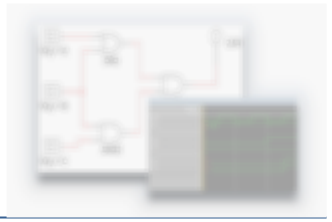
- **@ (λίστα ευαισθησίας)**
- Καθορίζει την συνθήκη η οποία πρέπει να ικανοποιηθεί προκειμένου να ξεκινήσει η εκτέλεση των εντολών
- Καθορίζει τα συμβάντα που πρέπει να λάβουν χώρα προκειμένου να ξεκινήσει η εκτέλεση των εντολών
  - Ως συμβάν μπορεί να θεωρηθεί η μεταβολή της τιμής ενός σήματος @(A) ή η εμφάνιση συγκεκριμένης ακμής του @(posedge A), @(negedge A)
- Συνήθεις μορφές ελέγχου συμβάντων:
  - **or**
  - **posedge** (+ακμή)
  - **negedge** (-ακμή)

→

Οι (+, -) ακμές χρησιμοποιούνται κυρίως για:
- Εντοπισμό ακμών ρολογιού
- Ασύγχρονων σημάτων ελέγχου, όπως reset και set

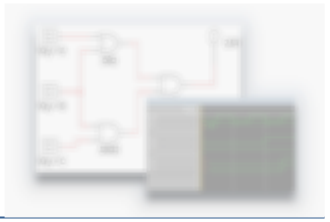
- Στη συνδυαστική λογική, η λίστα ευαισθησίας πρέπει να περιλαμβάνει όλα τα σήματα που διαβάζονται από το μπλοκ. Εναλλακτικά χρησιμοποιείται always @(*)

Τελεστής ελέγχου συμβάντων @ , Λίστα ευαισθησίας



- `always` // Επαναλαμβάνει διαρκώς τις εντολές του. Αν δεν υπάρχει κάποια εντολή αναμονής (π.χ. #10) μέσα στο μπλοκ, μπορεί να δημιουργηθεί άπειρος βρόχος χωρίς πρόοδο του χρόνου προσομοίωσης
- `always @(*)` // Ενεργοποιείται όταν μεταβληθεί κάποιο από τα σήματα που διαβάζονται από το μπλοκ και περιλαμβάνονται στην αυτόματα υπολογιζόμενη λίστα ευαισθησίας.
- `always @(A)`
- `always @(A or B or C)` // Verilog 1995
- `always @(A, B, C)` // Verilog 2001, 2005
- `always @(posedge clock)`
- `always @(posedge clock, negedge reset)`

Διαδικαστικές εντολές αναθέσεις



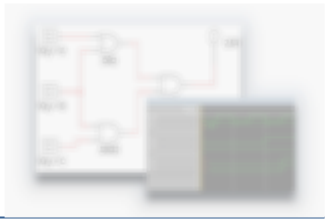
Υπάρχουν δύο είδη διαδικαστικών αναθέσεων.

- **Διαδικαστικές εντολές ανάθεσης**:
  - **=** **Blocking assignment**, ανασταλτική ανάθεση

    Ακολουθιακή εκτέλεση
    - Εκτέλεση με τη σειρά εμφάνισης
    - Αξιολόγηση άμεσα Δεξιά και άμεση Ανάθεση Αριστερά
    - Η επόμενη εντολή βλέπει τη νέα τιμή της μεταβλητής
  - **<=** **Non-blocking assignment**, μη ανασταλτική ανάθεση

    Παράλληλη-ταυτόχρονη εκτέλεση
    - Γίνεται κατ' αρχάς υπολογισμός του συνόλου των αλγεβρικών εκφράσεων που βρίσκονται στο δεξιό τμήμα των σχετικών εντολών
    - Έπειτα γίνονται ταυτόχρονα οι αναθέσεις τιμών στις μεταβλητές που αναφέρονται στο αριστερό τμήμα των εντολών

Διαδικαστικές εντολές αναθέσεις



Παράδειγμα:

- Έστω ότι κάποια χρονική στιγμή: **a=1, b=2, c=3**

```
a = 10;
b = a;
c = b;
```

```
a <= 10;
b <= a;
c <= b;
```

Blocking assignment:

a = 10

b = 10

c = 10

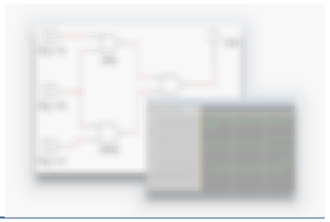
Non-blocking assignment:

a = 10

b = 1 (old value of a)

c = 2 (old value of b)

Διαδικαστικές εντολές αναθέσεις



- Δεν θα λειτουργήσει όπως αναμένουμε!

```
reg d1, d2, d3, d4;

always @(posedge clk)    d2 = d1;
always @(posedge clk)    d3 = d2;
always @(posedge clk)    d4 = d3;
```

Η χρήση **blocking assignments** σε διαφορετικά clocked always blocks μπορεί να δημιουργήσει **race conditions**, επειδή η σειρά εκτέλεσης των blocks δεν είναι καθορισμένη

- Εκτελείται με κάποια σειρά, αλλά δεν γνωρίζουμε με ποια!

Διαδικαστικές εντολές αναθέσεις



- Αυτή η εκδοχή όμως θα λειτουργήσει!

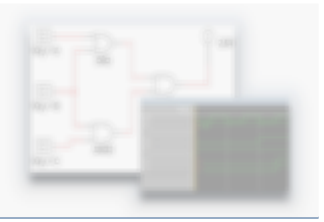
```
reg d1, d2, d3, d4;

always @(posedge clk)    d2 <= d1;
always @(posedge clk)    d3 <= d2;
always @(posedge clk)    d4 <= d3;
```

Nonblocking rule:
RHS evaluated when assignment runs

LHS updated only after all events
for the current instant have run

Διαδικαστικές εντολές αναθέσεις



- **Blocking assignments (=)**

  Χρησιμοποιούνται κυρίως για την περιγραφή **συνδυαστικής λογικής** μέσα σε διαδικαστικά μπλοκ always.
  - Η ενημέρωση της μεταβλητής πραγματοποιείται άμεσα.
  - Στη λίστα ευαισθησίας περιλαμβάνονται όλα τα σήματα που διαβάζονται από το μπλοκ.
  - Προτιμάται η χρήση always @(*).

- **Non-blocking assignments (<=)**

  Χρησιμοποιούνται κατά κανόνα για την περιγραφή **ακολουθιακής λογικής**, όπως μανδαλωτών (latches) και flip-flops.
  - Η ενημέρωση των μεταβλητών προγραμματίζεται για μεταγενέστερη φάση της ίδιας χρονικής στιγμής προσομοίωσης (Μοντελοποίηση ταυτόχρονης εκτέλεσης συμβάντων (π.χ. από ακμές ρολογιού)).
  - Στα **flip-flops** χρησιμοποιούμε λίστα ευαισθησίας με ακμή ρολογιού και, όπου απαιτείται, ασύγχρονο reset ή set.
  - Στα **level-sensitive latches** χρησιμοποιούμε λίστα ευαισθησίας που περιλαμβάνει τις εισόδους και το enable, ή always @(*).

Διαδικαστικές εντολές αναθέσεις



Προτεινόμενη σύμβαση κωδικοποίησης για συνθέσιμη Verilog:

- **Blocking assignment** **(=)** **Συνδυαστική λογική**

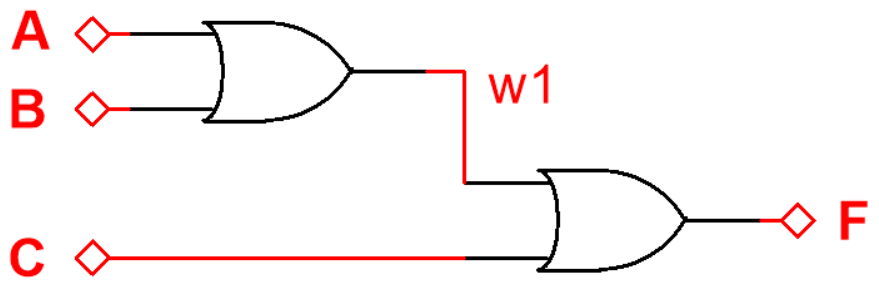
  Ανασταλτικές αναθέσεις

- **Non-blocking assignment** **(<=)** **Ακολουθιακή λογική**

  Μη-ανασταλτικές αναθέσεις

Το είδος του κυκλώματος καθορίζεται από τη συνολική περιγραφή και όχι αποκλειστικά από τον τελεστή ανάθεσης.

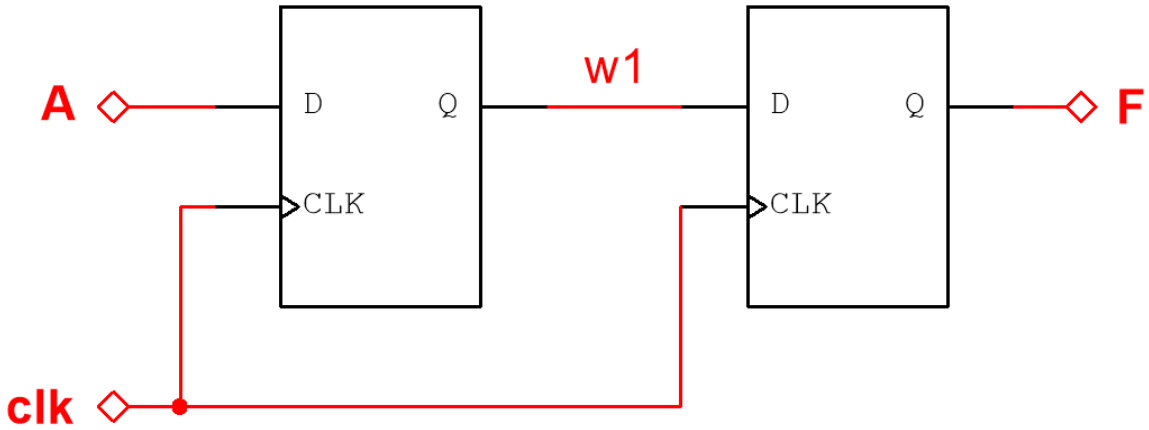
Διαδικαστικές εντολές αναθέσεις




```
module circuit1 (output reg F, input A, B, C);

reg w1;

always @(A, B, C)
begin
   w1 = A | B;
   F = w1 | C;
end
endmodule
```

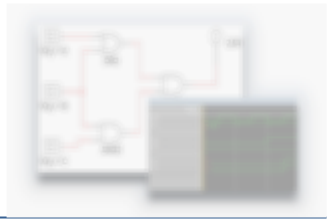



```
module circuit2 (output reg F, input A, clk);

reg w1;

always @(posedge clk)
begin
   w1 <= A;
   F <= w1;
end
endmodule
```

Παράλληλες Αναθέσεις – `fork/join`



- Το τμήμα **`fork`** εκτελεί παράλληλα όλες τις προτάσεις μέχρι το **`join`**
  - Χρησιμοποιείται κυρίως σε testbenches και γενικά δεν υποστηρίζεται από τις συνήθεις ροές σύνθεσης

- Μορφή
  - **`fork`**

```
    statement;
    statement;
    statement;
    …
  join
```

Παράλληλες Αναθέσεις – fork/join



Χωρίς fork/join

```
reg x,y;
reg [1:0] z;

initial
begin
      x = 1'b0;     // time 0
  #5  y = 1'b1;     // time 5
  #10 z = {x,y};    // time 15
end
```

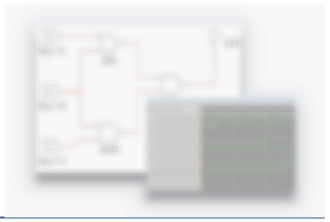
Με fork/join

```
reg x,y;
reg [1:0] z;

initial
  fork
      x = 1'b0;     // time 0
    #5  y = 1'b1;   // time 5
    #10 z = {x,y};  // time 10
  join
```

Στο **begin/end** οι καθυστερήσεις αθροίζονται διαδοχικά, ενώ στο **fork/join** κάθε παράλληλη διαδικασία μετρά τη δική της καθυστέρηση από την έναρξή της.

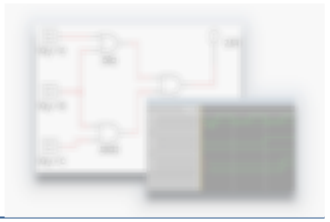
Εναλλακτική σύνταξη Verilog 2001, 2005



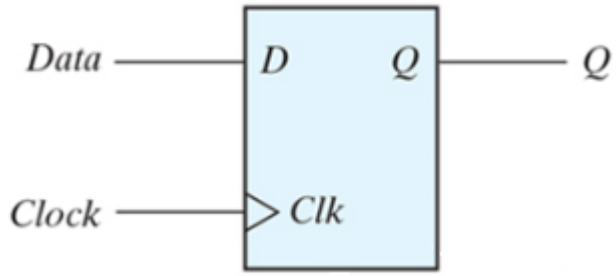
Verilog 1995	Verilog 2001, 2005
module D_latch (Q, enable, D); **output** Q; **input** enable, D; **reg** Q; …	**module** D_latch (**output reg** Q, **input** enable, D); …

Οι δύο συντάξεις είναι ισοδύναμες. Η σύνταξη Verilog 2001/2005 επιτρέπει τη δήλωση των θυρών απευθείας στην επικεφαλίδα του module

Υλοποίηση flip-flop χωρίς είσοδο μηδενισμού (reset)



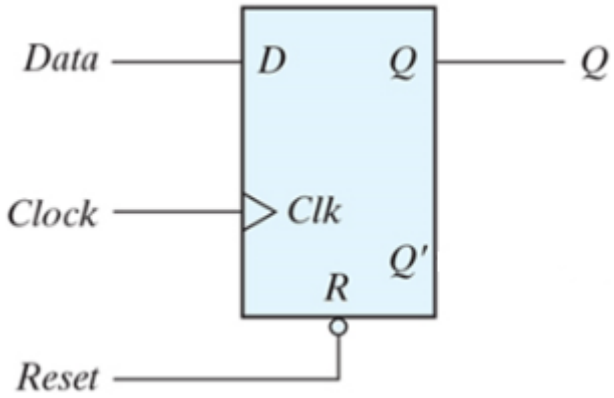
- Η **έξοδος** πρέπει να δηλωθεί ως **reg**
  - Αναγκαίο διότι λαμβάνει τιμή μέσω μια εντολής διαδικαστικής ανάθεσης.



- Στην **sensitivity list** της always θα βάλουμε **(posedge clock)** (ή negedge clock).
- Το μπλοκ ενεργοποιείται μόνο στην καθορισμένη ακμή του ρολογιού. Οι μεταβολές των εισόδων σε άλλες χρονικές στιγμές δεν επηρεάζουν την αποθηκευμένη τιμή της εξόδου.

Υλοποίηση flip-flop με ασύγχρονο μηδενισμό (ενεργό-χαμηλά)

- **Ο μηδενισμός** (η δράση του reset) **είναι ασύγχρονος, επειδή μια αλλαγή στην τιμή του reset μπορεί να πυροδοτήσει την εκτέλεση της διεργασίας ανεξάρτητα από την κατάσταση του ρολογιού (clock)**



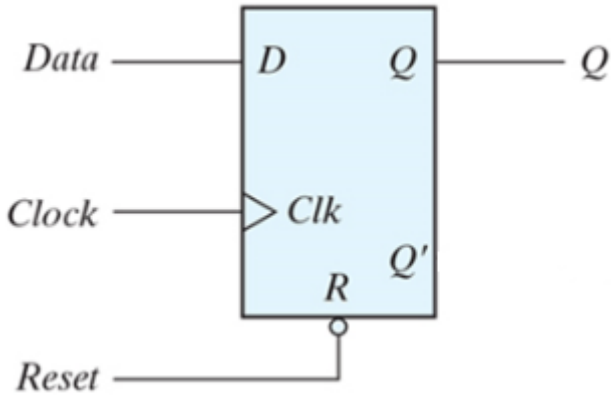
- Εάν το flip-flop περιλαμβάνει μία ασύγχρονη είσοδο μηδενισμού (**reset**), τότε στην περιγραφή του θα χρησιμοποιήσουμε μία **if**.

- Στην **sensitivity list** της always θα βάλουμε **(posedge clock, negedge reset)**, ώστε να ενεργοποιηθεί στις αντίστοιχες ακμές των σημάτων.

- Η ενεργοποίηση του reset (1 → 0) μηδενίζει την έξοδο ανεξάρτητα από το clock. Μετά την απενεργοποίηση του reset, η έξοδος ενημερώνεται ξανά στην επόμενη ενεργή ακμή του clock.

Υλοποίηση flip-flop με ασύγχρονο μηδενισμό (ενεργό-χαμηλά)

- Για την περιγραφή ενός D flip-flop με ασύγχρονο reset ισχύουν τα εξής:



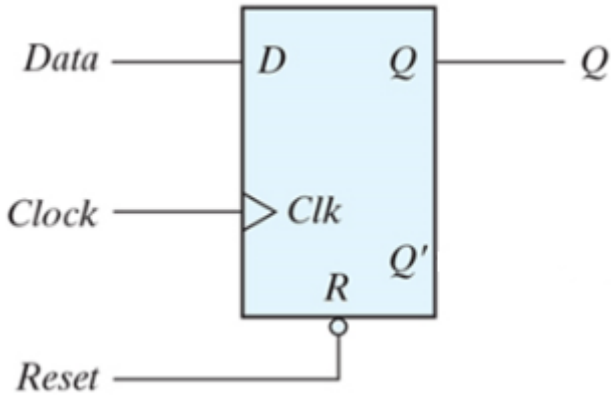
1. Στη **λίστα ευαισθησίας** περιλαμβάνουμε τη θετική ακμή του clock **(posedge clock)** και την αρνητική ακμή του reset **(negedge reset)**.

2. Η συνθήκη **if (!reset)** ελέγχει πρώτη την κατάσταση του ασύγχρονου reset, ώστε ο **μηδενισμός να έχει προτεραιότητα**.

3. Στον κλάδο **else** περιγράφουμε την κανονική λειτουργία του flip-flop, δηλαδή τη φόρτωση της εισόδου D στην έξοδο Q κατά τη θετική ακμή του clock.

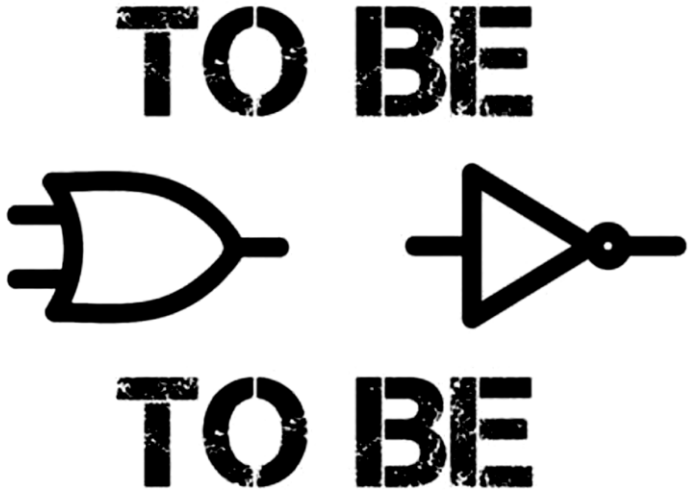
Υλοποίηση flip-flop με ασύγχρονο μηδενισμό (ενεργό-χαμηλά)

- Το reset είναι ασύγχρονο και ενεργό-χαμηλά.
  - Το **reset είναι ασύγχρονο και έχει προτεραιότητα**. Η ενεργοποίηση του παρακάμπτει το clock.
  - Όταν reset=0, η έξοδος Q μηδενίζεται ανεξάρτητα από το clock.
  - Όσο το reset παραμένει ενεργό, η έξοδος Q διατηρείται στο 0.
  - Όταν reset=1, το flip-flop λειτουργεί κανονικά και η έξοδος Q ενημερώνεται στη θετική ακμή του clock.



Ευχαριστώ για την προσοχή σας!

- **Ερωτήσεις / Απορίες ;**



Επικοινωνία: ece119.uth@gmail.com