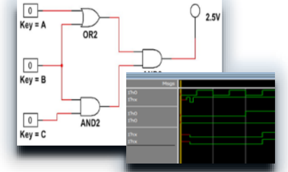
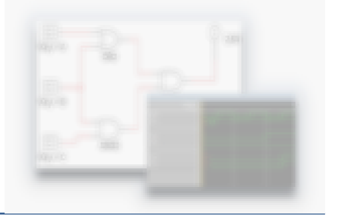


ECE119 – Ψηφιακή Σχεδίαση

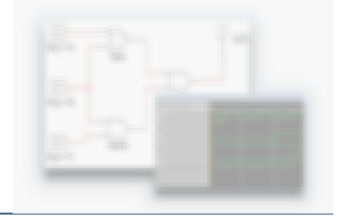
Διδάσκοντες Εργαστηρίου: Δ. Καραμπερόπουλος
Δ. Γαρυφάλλου

- Lab 8: Verilog (Μέρος 4)



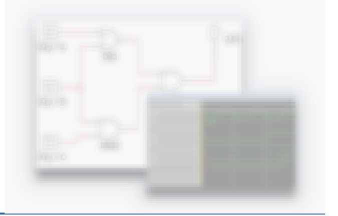
Η Γλώσσα Verilog (Μέρος 4)

Μοντέλα HDL Συνδυαστικών κυκλωμάτων



- Η βασική δομή για τη μοντελοποίηση κυκλωμάτων με χρήση της Verilog είναι η υπομονάδα (module)
- Η λογική λειτουργία μιας υπομονάδας μπορεί να περιγραφεί με ένα από τα ακόλουθα στιλ (τρόπους) μοντελοποίησης, ή με έναν συνδυασμό τους:
- **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**.

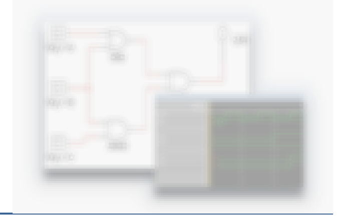
Μοντελοποίηση συμπεριφοράς (behavioral modeling),



- Περιγράφει κυκλώματα σε **αλγοριθμικό** και **λειτουργικό** επίπεδο.
- Χρησιμοποιείται κυρίως για την περιγραφή **ακολουθιακών κυκλωμάτων**.
- Περιγράφει τι κάνει ένα στοιχείο και όχι πως το κάνει.
- Συντίθεται σε ένα κύκλωμα που έχει αυτή τη συμπεριφορά.
- Χρησιμοποιεί το μπλοκ “**always**”

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ χρόνια δημιουργίας

Always Block



- Η γενική μορφή είναι:

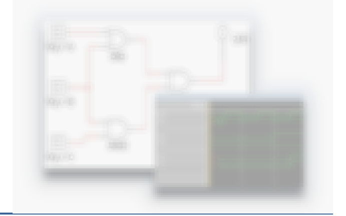
  Always @ (έκφραση ελέγχου συμβάντων) **begin**

  // Εντολές διαδικαστικής ανάθεσης 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 εντολές στο always block εκτελούνται ακολουθιακά (η μία μετά την άλλη).
- 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.

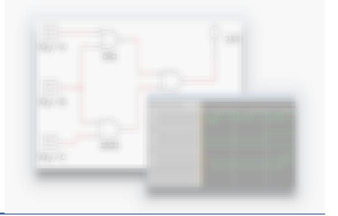
Always Block



- Στη Verilog έχουμε 2 τύπους εντολών ανάθεσης:
  - **Διαρκείς** (assign – **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)
  - **Διαδικαστικές** (**μέσα σε μπλοκ** always ή initial)
    Εκτελούνται όταν θα εκτελεστεί το μπλοκ που τις περιλαμβάνει

- **Διαδικαστικές εντολές ανάθεσης**:
  - **=** **Blocking assignment**, ανασταλτική ανάθεση
    ακολουθιακή εκτέλεση
  - **<=** **Non-blocking assignment**, μη ανασταλτική ανάθεση
    παράλληλη-ταυτόχρονη εκτέλεση

Always Block



always @ (event list)

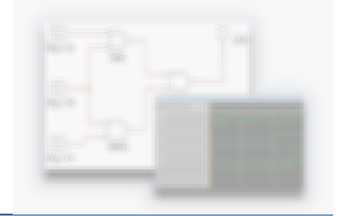
- Παρατηρήσεις:

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

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

Καθορίζει πότε θα εκτελεστεί το always statement

Απαιτεί αλλαγή τιμής ενός στοιχείου της λίστας



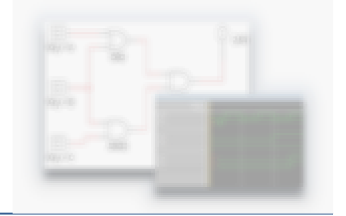
Τμήματα always και initial

Τμήμα **initial**	Τμήμα **always**

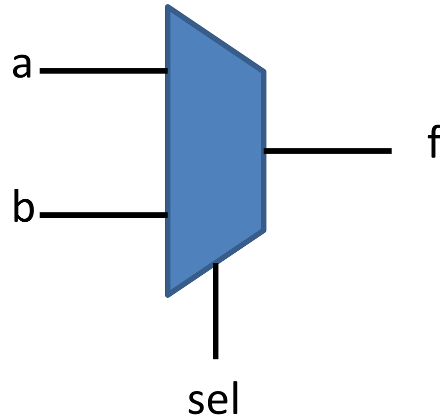
`initial`
`begin`
`// run once`
`a=0; b=0;`
`#5; a=1; b=1;`
`end`
	`always @(b or c)`
	`begin`
	`// run always`
	`a = b & c;`
	`end`

‣ Εκτελείται μια φορά, στην εκκίνηση της προσομοίωσης ‣ Δεν επαναλαμβάνεται ‣ Χρησιμοποιείται για να παρέχει διανύσματα εισόδου στο κύκλωμα (test bench) ‣ Δεν είναι συνθέσιμο	‣ Εκτελείται στην εκκίνηση της προσομοίωσης ‣ Είναι άπειρος βρόχος (επαναεκτελείται) ‣ Χρησιμοποιείται για να περιγράφει διαρκή και μόνιμη συμπεριφορά (συνδυαστική ή ακολουθιακή) ‣ Είναι συνθέσιμο

Always Block



Πολυπλέκτης 2 σε1



a	b	sel	f
x	y	0	x
x	y	1	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
```

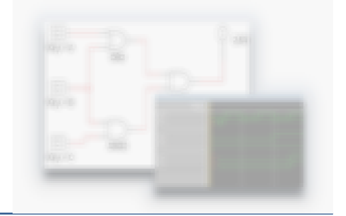
Λίστα ευαισθησίας

Έξοδος τύπου **reg**

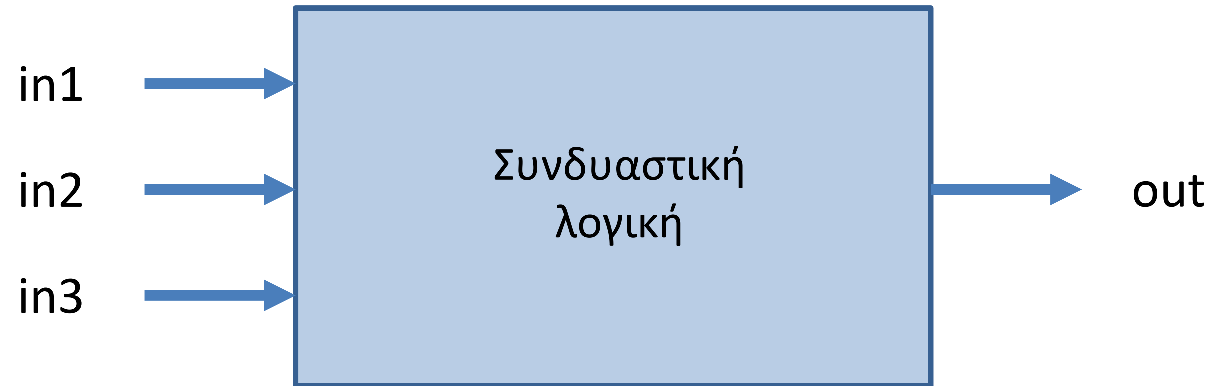
Κατά την περιγραφή συνδυαστικής λογικής χρησιμοποιούμε blocking assignments και τοποθετούμε **όλα τα inputs στην λίστα ευαισθησίας** του procedural block.

Η μοντελοποίηση ροής δεδομένων είναι προτιμότερη στα συνδυαστικά κυκλώματα

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



Always Block

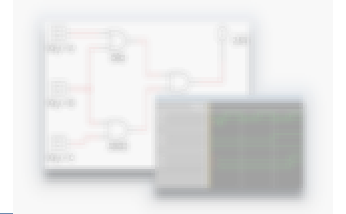


Ισοδύναμοι ορισμοί

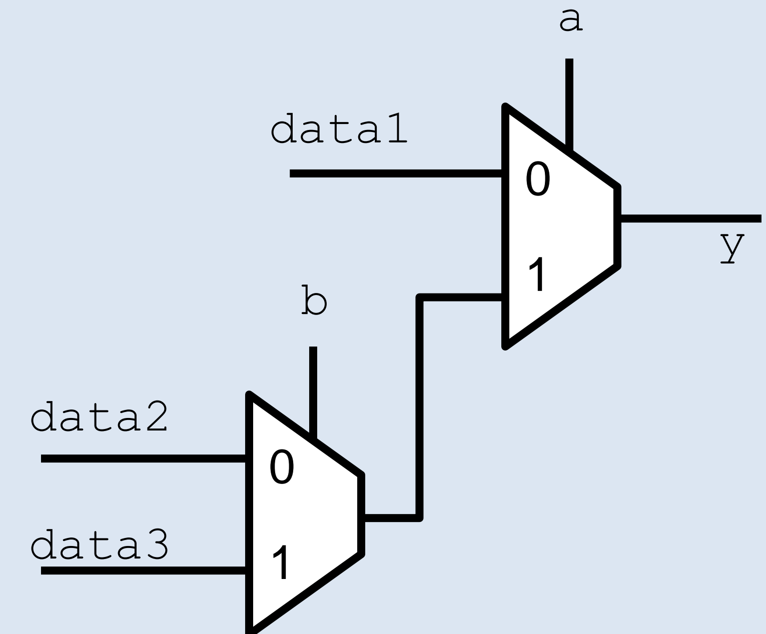
```
reg out;

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

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

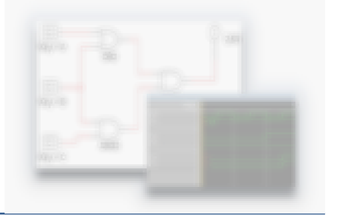
Συνθήκη if/else



- Μόνο μέσα σε blocks !
- 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`	

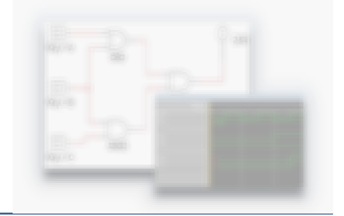
```
always @(a or b or data1 or
data2 or data3)
begin

  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;

endmodule
```

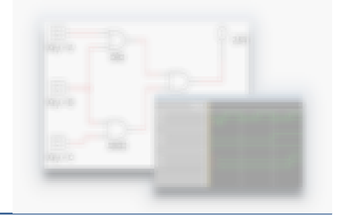
Συνθήκη if/else



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

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
χρόνια δημιουργίας

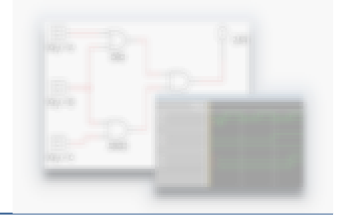
Συνθήκη case



- Μόνο μέσα σε blocks !
- Μόνο για σταθερές εκφράσεις
- Εκτελείται μόνο η πρώτη περίπτωση που ταιριάζει. (Δεν υπάρχει `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
```

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
χρόνια δημιουργίας

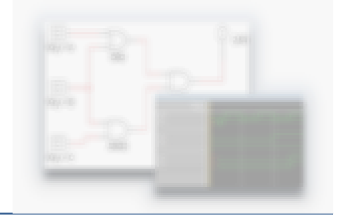


Συνθήκη case

- Αναθέτοντας X σε μία μεταβλητή, δίνουμε το δικαίωμα στο εργαλείο σύνθεσης να δώσει ό,τι τιμή θέλει.

```
// 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;
      8'b00000010: Y = 1;
      8'b00000100: Y = 2;
      8'b00001000: Y = 3;
      8'b00010000: Y = 4;
      8'b00100000: Y = 5;
      8'b01000000: Y = 6;
      8'b10000000: Y = 7;
      default:     Y = 3'bX;      // Don't care when input
                                  // is not 1-hot
    endcase
endmodule
```

Βρόχοι for και while

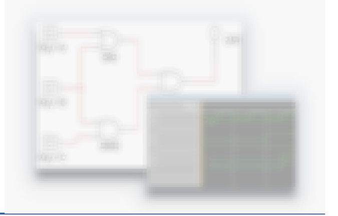


- Μόνο μέσα σε blocks !
- Λειτουργούν όπως σε παραδοσιακές γλώσσες προγραμματισμού
- Δεν υποστηρίζονται:
  - break, continue
  - i++, i--
- Χρήση:
  - testbench
  - επαναληπτική εμφάνιση

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

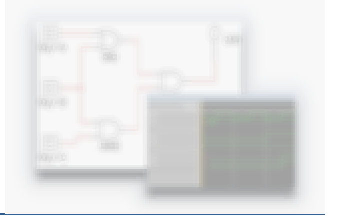
```
integer j;
initial begin
  j=0;
  while (j < 10)
     begin
       $display (“j= %b”,j);
       j = j + 1;
     end
end
```

Βρόχοι for και while



Test Bench	Module

`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`
`end`
`endmodule`
	`// 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`

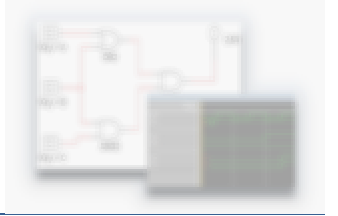


Βρόχοι for και while

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
	b[0], c[0] → a[0]
	b[1], c[1] → a[1]
	b[2], c[2] → a[2]
	b[3], c[3] → a[3]
	b[4], c[4] → a[4]

Βρόχοι repeat και forever



- **Repeat** (<number_of_loops>)

```
Repeat (<number_of_loops>)
    begin
        statement_1
        …
        statement_n
    end
```

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

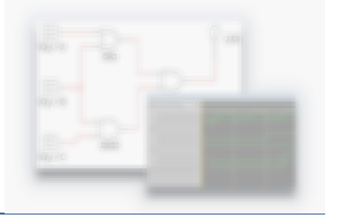
Execute statements a fixed number of times

- **forever**

```
forever
    begin
        statement_1
        …
        statement_n
    end
```

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

Execute statements forever



Περιγραφή σήματος ρολογιού

1ος τρόπος

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

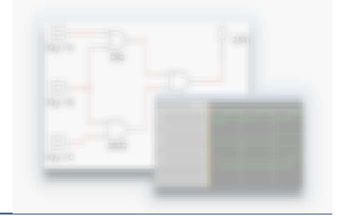
2ος τρόπος

```
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;
```

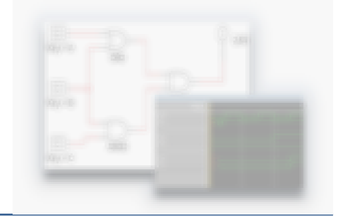
Τελεστές # και @



- **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/αναμονή μιας δραστηριότητας μέχρι την εκπλήρωση συγκεκριμένων συνθηκών ή την πραγματοποίηση προκαθορισμένων συμβάντων (δηλαδή αλλαγών τιμών σημάτων)
    Π.χ. `@(A or B),` `@(A)`
  - Χρησιμοποιείται στην always

Λίστα ευαισθησίας

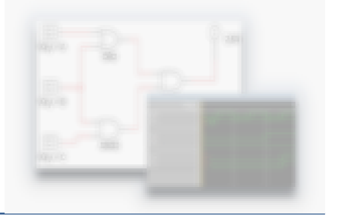


- **@ (λίστα ευαισθησίας)**
- Καθορίζει την συνθήκη η οποία πρέπει να ικανοποιηθεί προκειμένου να ξεκινήσει η εκτέλεση των εντολών
- Καθορίζει τα συμβάντα που πρέπει να λάβουν χώρα προκειμένου να ξεκινήσει η εκτέλεση των εντολών
  - Ως συμβάν θεωρείται μία άνευ συνθηκών αλλαγή της τιμής ενός σήματος. Π.χ. `@(A)`
- Επιτρέπονται μόνο οι εκφράσεις:
  - **or**
  - **posedge** (+ακμή)
  - **negedge** (-ακμή)

Οι (+, -) ακμές χρησιμοποιούνται μόνο για
- Ρολόγια
- Σήματα αρχικοποίησης (reset)

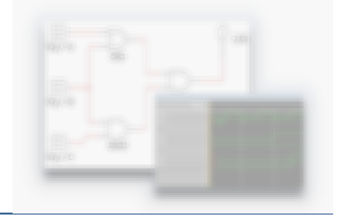
- Στην περιγραφή συνδυαστικής λογικής, όλα τα σήματα εισόδου πρέπει να περιλαμβάνονται

Τελεστής ελέγχου συμβάντων @ , Λίστα ευαισθησίας



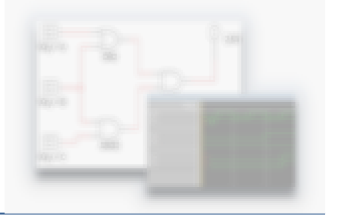
- **always** // εκτελείται συνεχώς
- **always** @* // εκτελείται οποτεδήποτε αλλάξει η τιμή οποιουδήποτε σήματος αναγράφεται στο δεξί μέρος κάποιας ανάθεσης στο εσωτερικό του
- **always** @(A)
- **always** @(A **or** B **or** C) // Verilog 1995
- **always** @(A, B, C) // Verilog 2001, 2005
- **always** @(**posedge** clock)
- **always** @(**posedge** clock, **negedge** reset)

Διαδικαστικές εντολές αναθέσεις



Υπάρχουν δύο είδη διαδικαστικών αναθέσεων.

- **Διαδικαστικές εντολές ανάθεσης**:
  - **=** **Blocking assignment**, ανασταλτική ανάθεση
    Ακολουθιακή εκτέλεση
    - Εκτελούνται με την σειρά εμφάνισης τους στο μπλοκ εντολών
    - Αξιολόγηση άμεσα Δεξιά και άμεση Ανάθεση Αριστερά
    - Εκτέλεση ακολουθιακά της επόμενης πρότασης
  - **<=** **Non-blocking assignment**, μη ανασταλτική ανάθεση
    Παράλληλη-ταυτόχρονη εκτέλεση
    - Γίνεται κατ’ αρχάς ταυτόχρονος υπολογισμός του συνόλου των αλγεβρικών εκφράσεων που βρίσκονται στο δεξιό τμήμα των σχετικών εντολών
    - Έπειτα γίνονται 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

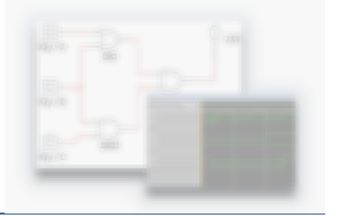
c = 10

Non-blocking assignment:

a = 10

b = 1 (old value of a)

c = 2 (old value of b)

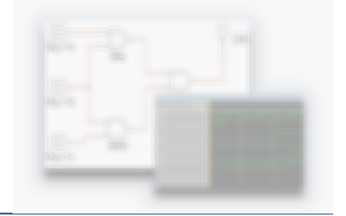


Διαδικαστικές εντολές αναθέσεις

- 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;
```

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



Διαδικαστικές εντολές αναθέσεις

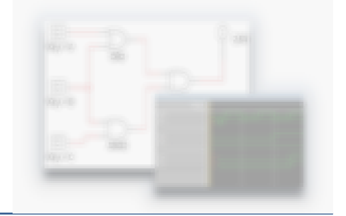
- Αυτή η εκδοχή όμως θα λειτουργήσει!

Nonblocking rule:
RHS evaluated when assignment runs

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

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

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

Διαδικαστικές εντολές αναθέσεις



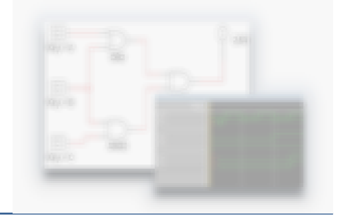
- Χρησιμοποιούμε **αναστaλτικές αναθέσεις (=)** σε περιπτώσεις όπου απαιτείται ακολουθιακή εκτέλεση των εντολών (μια προς μία), ή όταν θέλουμε να μοντελοποιήσουμε κυκλική συμπεριφορά (μπλοκ always) που σχετίζεται με στάθμες σημάτων (δηλαδή **συνδυαστική λογική**)

  (Στην περίπτωση αυτή τοποθετούμε **όλα τα inputs** στην λίστα ευαισθησίας του της always)

- Χρησιμοποιούμε **μη-ανασταλτικές αναθέσεις (<=)** όταν μοντελοποιούμε ταυτόχρονη εκτέλεση συμβάντων (π.χ. από ακμές ρολογιού) και όταν μοντελοποιούμε συμπεριφορά μανδαλωτών, flip-flop και άλλων στοιχείων σύγχρονης λειτουργίας (δηλαδή **ακολουθιακή λογική**).

  (Στην περίπτωση αυτή τοποθετούμε **μόνο το σήμα ρολογιού και το reset** στην λίστα ευαισθησίας της always)

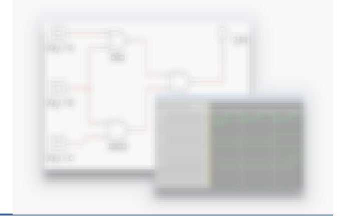
Διαδικαστικές εντολές αναθέσεις



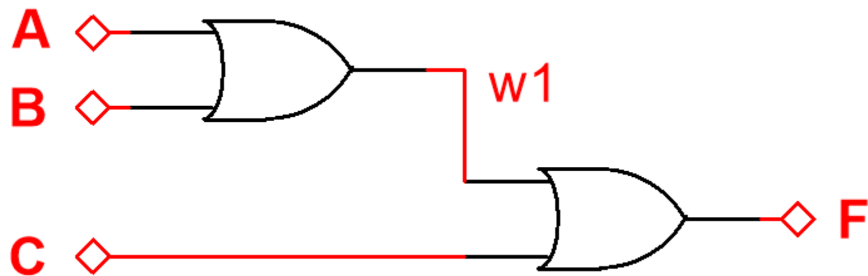
Συνεπώς…

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

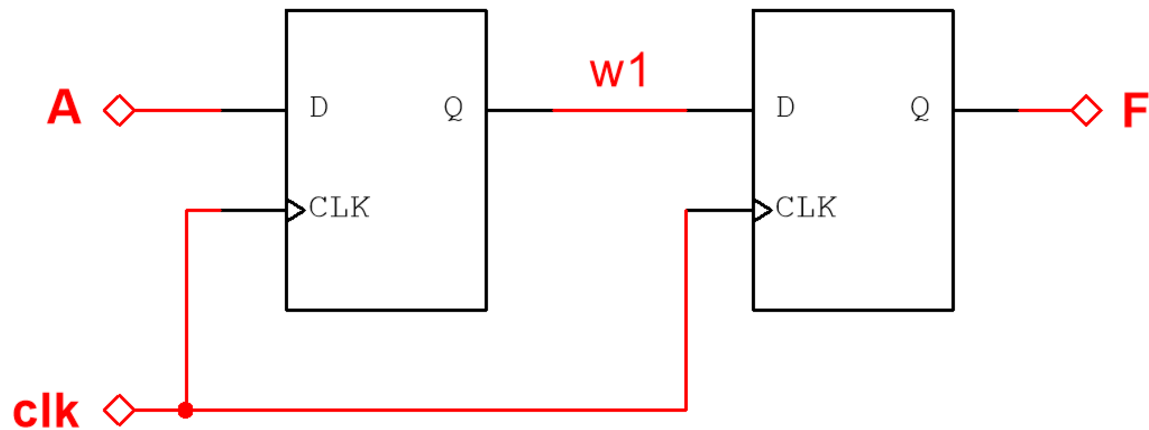
- **Non-blocking assignment** **(<=)** **Ακολουθιακή λογική**
  Μη-ανασταλτικές αναθέσεις



Διαδικαστικές εντολές αναθέσεις



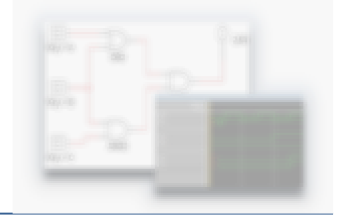

```
module circuit1 (output reg F, input A, B, C);

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
```

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
χρόνια δημιουργίας

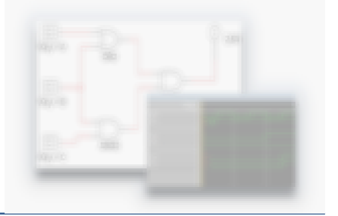
Διαδικασίες Συστήματος



- Καθιερωμένες διαδικασίες που ορίζει ένα εργαλείο EDA
  - Ξεκινούν με $, λ.χ. $monitor

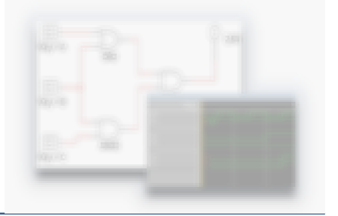
Όνομα Διαδικασίας	Λειτουργία
`$time`	Επιστρέφει τον χρόνο της προσομοίωσης (με την μορφή ενός 64-bit vector)
`$display`	Τυπώνει τιμές σημάτων – ανάλογη της printf `$display("`*`format-string`*`", expr1, …, exprn);` `%d` (decimal), `%h` (hex), `%b` (binary), `%t` (time)
`$monitor`	Παρακολουθεί σήματα ως γεγονότα, και τα τυπώνει όταν αποκτήσουν νέα τιμή – έχει ανάλογα ορίσματα όπως η `$display`
`$stop`	Διακόπτει την προσομοίωση (παύση)
`$finish`	Ολοκληρώνει την προσομοίωση & έξοδος από το περιβάλλον προσομοίωσης
`$random`	Επιστρέφει ένα 32-bit ψευδοτυχαίο αριθμό
`$readmemh`, `$readmemb`	Ανάγνωση περιεχομένων μνήμης

Διαδικασίες Συστήματος



Παραδείγματα	
`$display("Error at time %t: value is %h, expected %h", $time, actual_value, expected_value);`	
`$monitor("x=%b y=%b out=%b",x, y, out)`	x=0 y=0 out=0 x=0 y=1 out=1 x=1 y=0 out=1 x=1 y=1 out=0
`$monitor ("Time=%2d  x=%b y=%b out=%b", $time, x, y, out)`	Time= 0 x=0 y=0 out=0 Time=10 x=0 y=1 out=1 Time=20 x=1 y=0 out=1 Time=30 x=1 y=1 out=0
`$random %64`	

Παράλληλες Αναθέσεις – fork/join

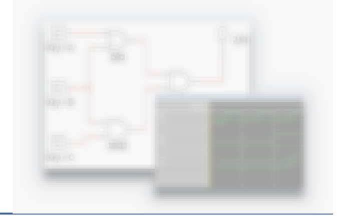


- Το τμήμα **fork** εκτελεί παράλληλα όλες τις προτάσεις μέχρι το **join**
  - Δύσκολα υποστηρίζονται στην σύνθεση

- Μορφή
  - **fork**

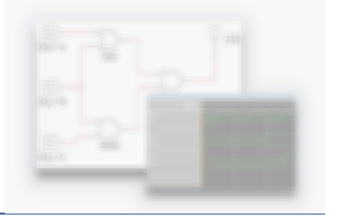
```
    statement;
    statement;
    statement;
    …
join
```

Παράλληλες Αναθέσεις – fork/join



Χωρίς fork/join	Με 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`
	`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`

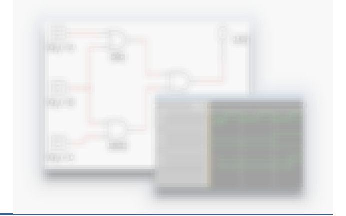


Εναλλακτική σύνταξη Verilog 2001, 2005

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);`
	…

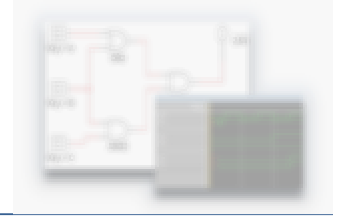
Υλοποίηση flip-flop χωρίς είσοδο μηδενισμού (reset)



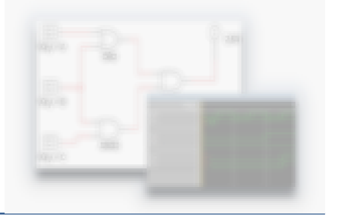
- Η **έξοδος** πρέπει να δηλωθεί ως **reg**
  - Αναγκαίο διότι λαμβάνει τιμή μέσω μια εντολής διαδικαστικής ανάθεσης.

- Στην **sensitivity list** της always θα βάλουμε **(posedge clock)** (ή negedge clock) ώστε να διασφαλίσουμε ότι το μπλοκ θα εκτελεστεί κατά την **θετική** (ή αρνητική) **ακμή** του ρολογιού και η έξοδος θα ενημερωθεί εκείνη και μόνο την χρονική στιγμή. Τυχούσα αλλαγή των εισόδων σε οποιαδήποτε άλλη χρονική στιγμή δεν θα επηρεάσει την έξοδο.

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

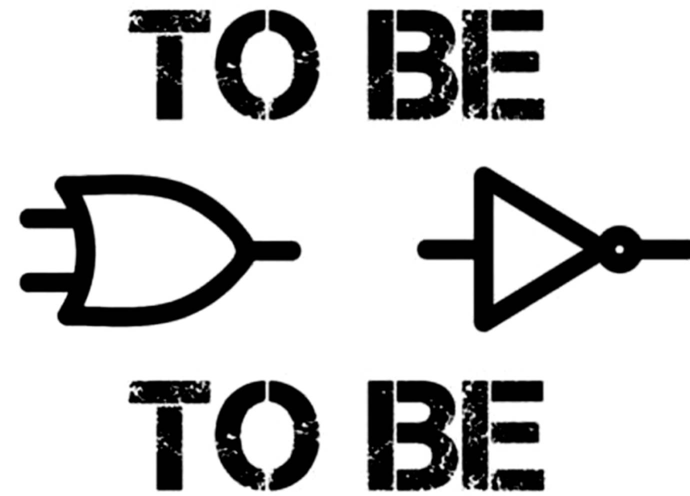


- **Ο μηδενισμός (η δράση του reset) είναι ασύγχρονος, επειδή μια αλλαγή στην τιμή του reset μπορεί να πυροδοτήσει την εκτέλεση της διεργασίας ανεξάρτητα από την κατάσταση του ρολογιού (clock)**
- Η **έξοδος** πρέπει να δηλωθεί ως **reg**
- Εάν το flip-flop περιλαμβάνει μία ασύγχρονη είσοδο μηδενισμού (**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.
- Οι κανόνες είναι οι εξής:
  1. **Τα ασύγχρονα συμβάντα ελέγχονται πρώτα (πριν το συμβάν του ρολογιού)**
  2. **Άρα κάθε εντολή if ή else if στις διαδικαστικές αναθέσεις πρέπει να αντιστοιχεί σε ένα ασύγχρονο συμβάν.**
  3. **Η δήλωση else της τελευταίας εντολής πρέπει να αντιστοιχεί στο συμβάν του ρολογιού (συγχρονισμού).**
- Το σήμα **reset είναι ασύγχρονο και έχει προτεραιότητα**. Η ενεργοποίηση του παρακάμπτει το clock.
  - Για όσο χρόνο το reset είναι 0, η έξοδος Q μηδενίζεται (ακόμα και αν το clock έχει θετική μετάβαση)
  - Μόνο εάν reset = 1 μπορεί το συμβάν ρολογιού posedge να μεταφέρει σύγχρονα τις τιμές των εισόδων στην Q



Ευχαριστώ για την προσοχή σας!

- **Ερωτήσεις / Απορίες ;**



Επικοινωνία: ece119.uth@gmail.com