HY225
Οργάνωση Υπολογιστών

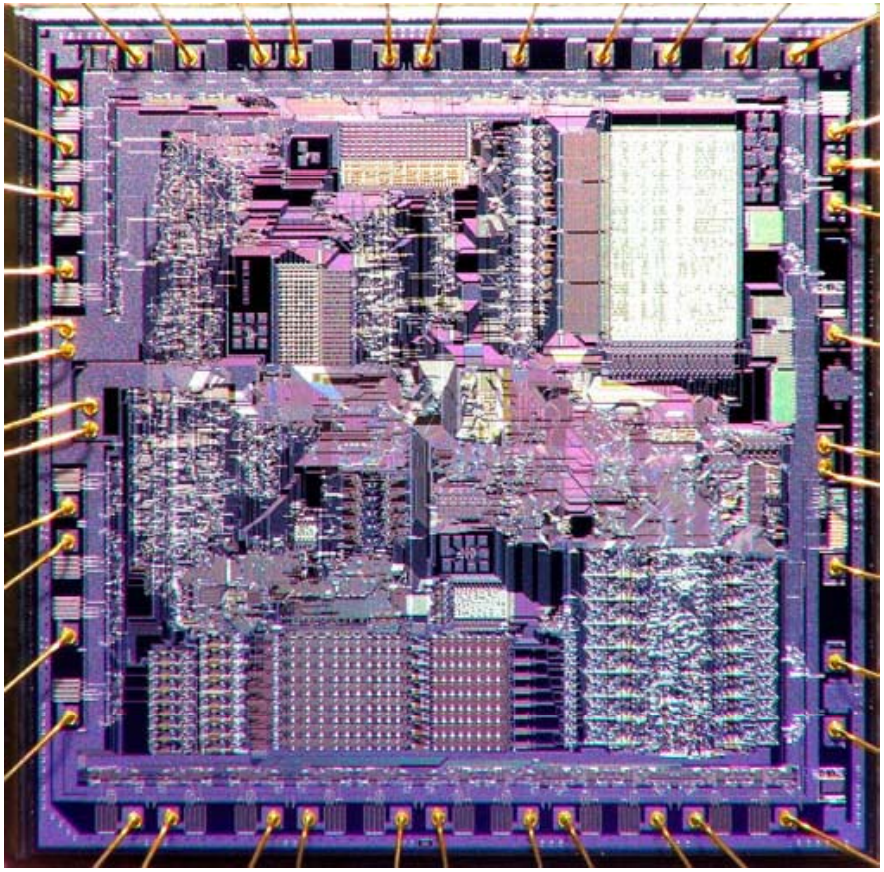
Εισαγωγή στη Verilog

Processors are everywhere

ARM-based products

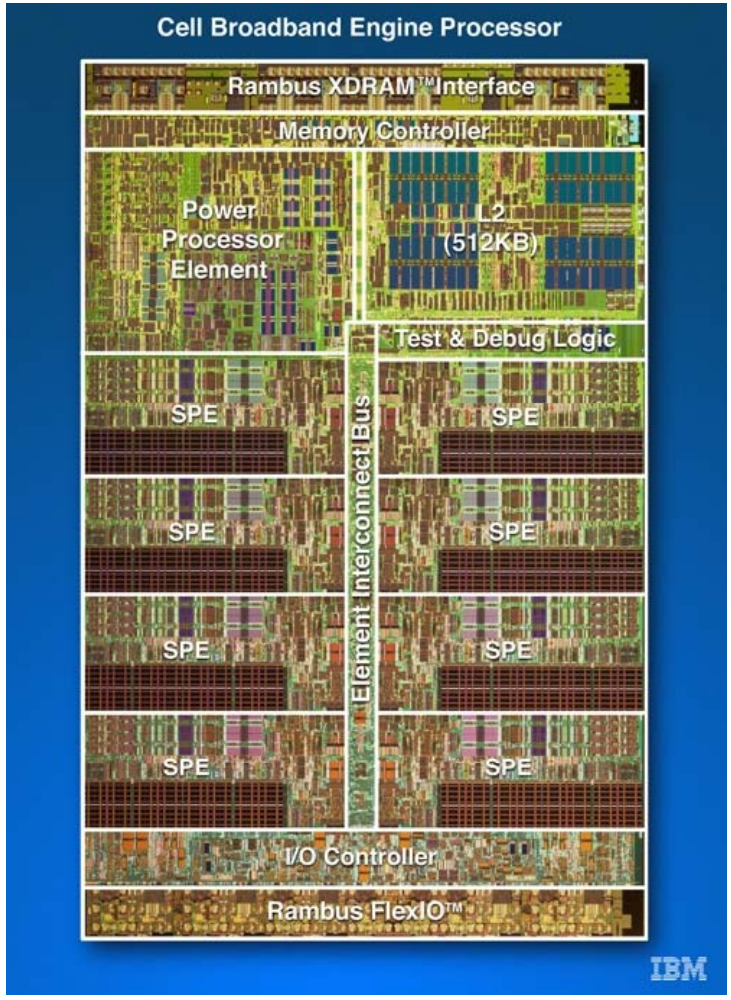
- CS225: How to build your own processor

Intel 8086 Processor



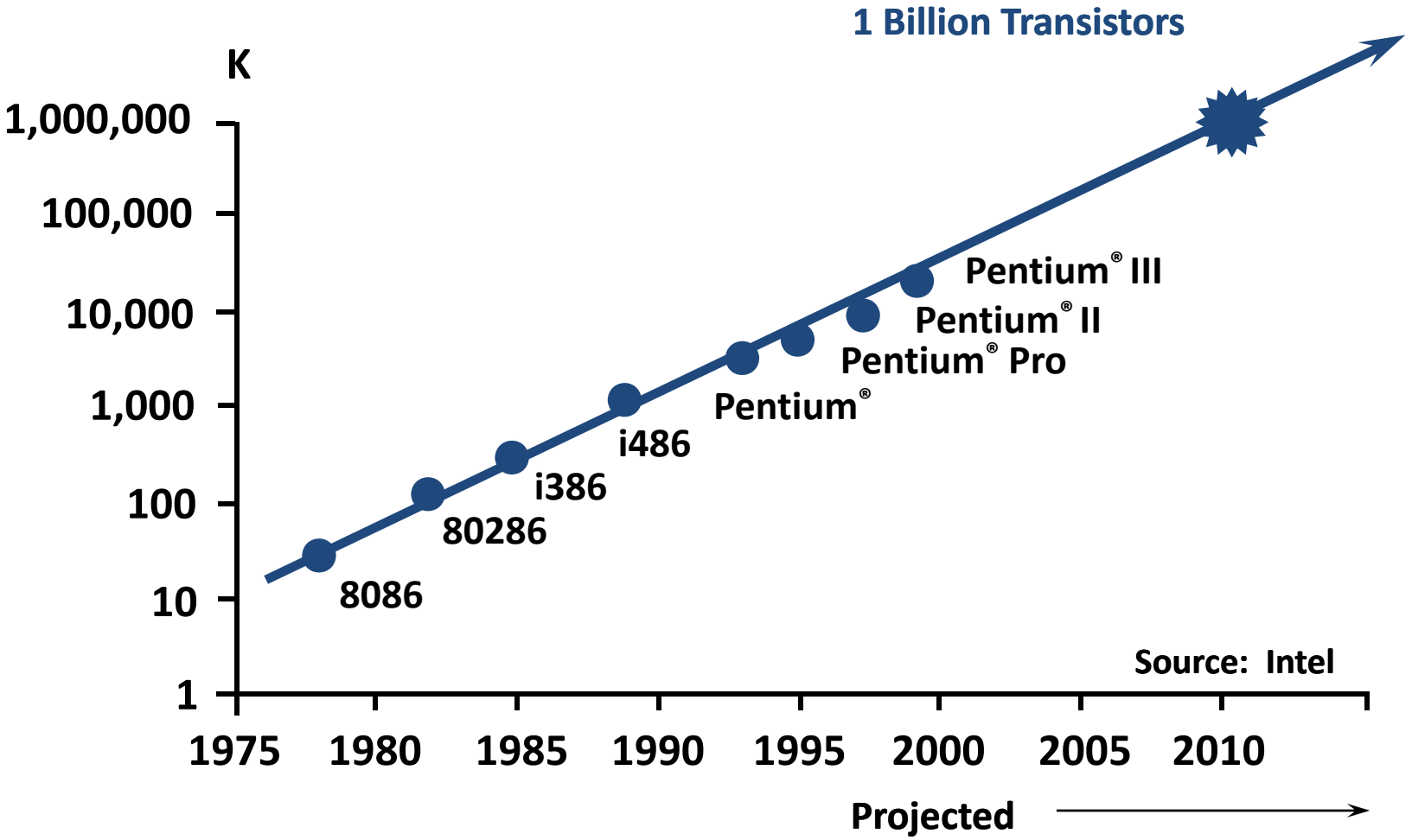
1978
29.000 transistors
5 MHz operation

Cell Processor, PS3

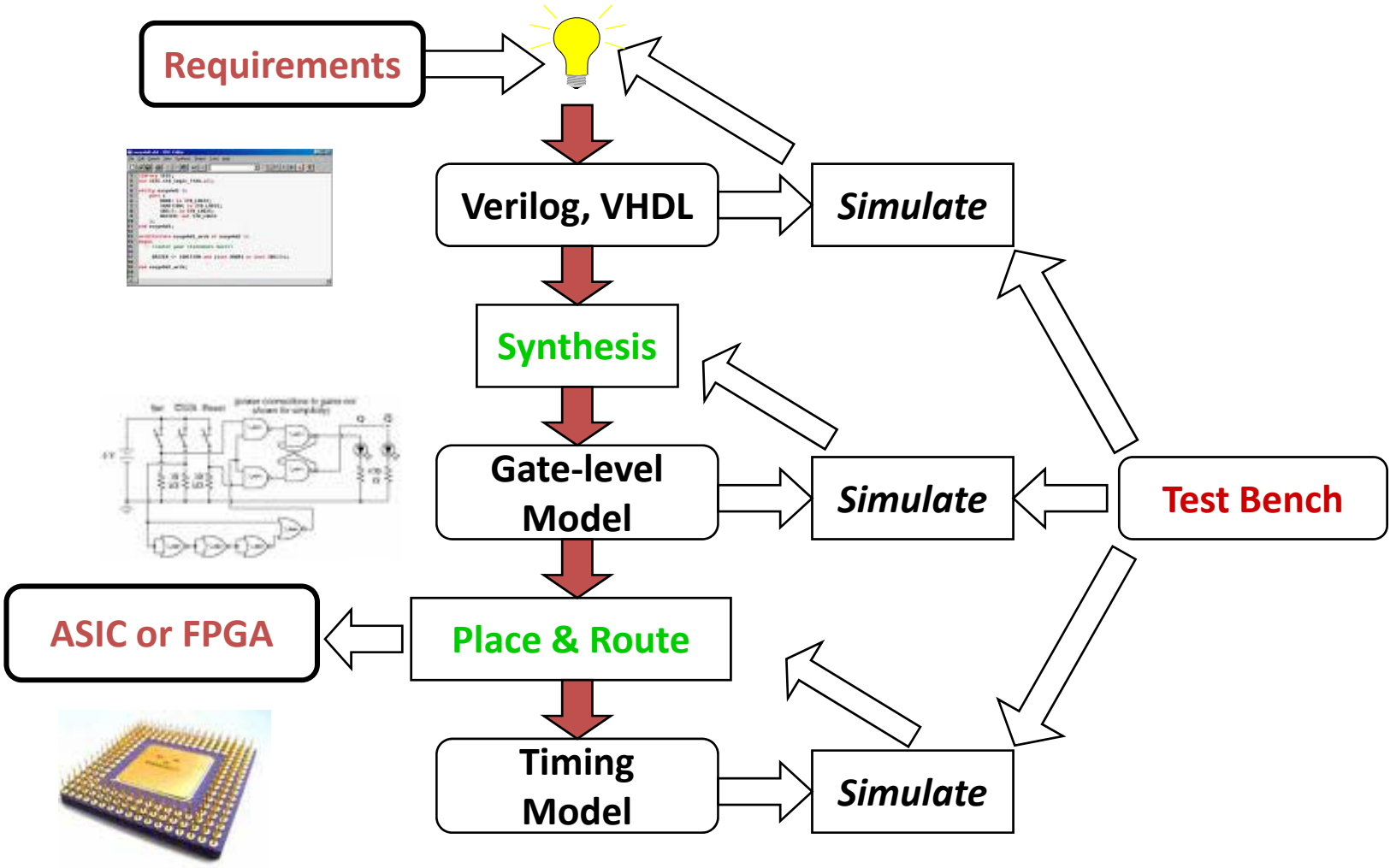


- 2008
- 9 Processors
- 234M transistors (~10.000 i8086)
- Clock speed: 4GHz
- 230 GFlops

Transistor Counts

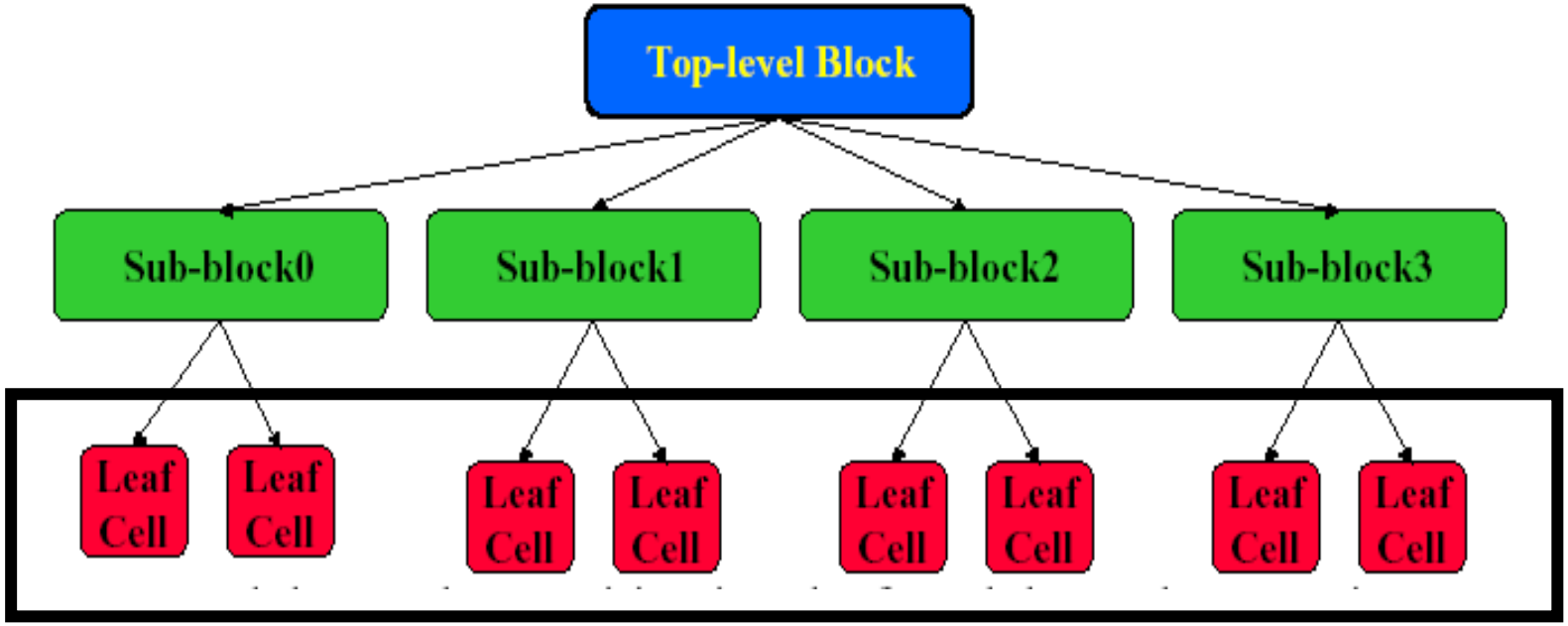


Τυπική Ροή Σχεδίασης (Design Flow)



Ιεραρχικές Μεθοδολογίες Σχεδίασης

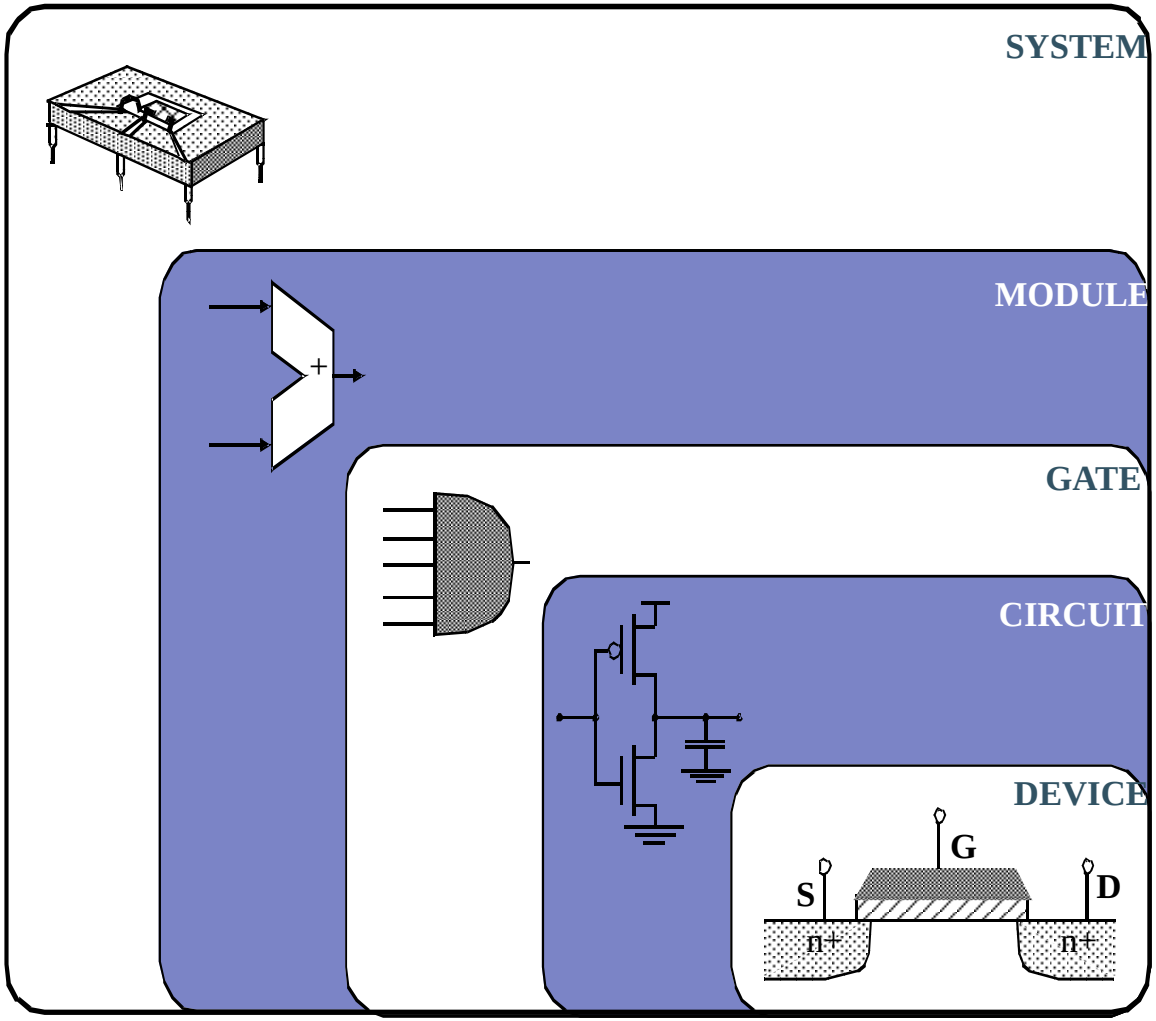
- Top-Down ή Bottom-Up
  - Συνήθως μια μίξη
- Τελικό σύστημα αποτελείται από τα Leaf blocks που τρέχουν όλα παράλληλα.



Τι είναι η Verilog;

- Verilog Hardware Description Language (HDL)
  - Μία υψηλού επιπέδου γλώσσα που μπορεί να αναπαριστά και να προσομοιώνει ψηφιακά κυκλώματα.
    - Hardware concurrency
    - Parallel Activity Flow
    - Semantics for Signal Value and Time
  - Παραδείγματα σχεδίασης με Verilog HDL
    - Intel Pentium, AMD K5, K6, Athlon, ARM7, etc
    - Thousands of ASIC designs using Verilog HDL
- Other HDL : VHDL, SystemC, SystemVerilog

Design Abstraction Levels



Matlab, SystemC, SystemVerilog

Verilog, VHDL

VLSI

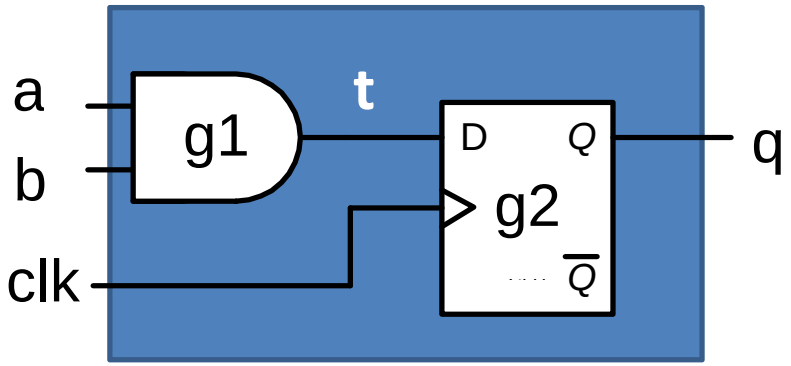
Αναπαράσταση Ψηφιακών Συστημάτων

- Η Verilog HDL χρησιμοποιείται για να φτιάξουμε το μοντέλο ενός συστήματος.
- Λόγοι:
  - Ορισμός Απαιτήσεων (requirements specification)
  - Documentation
  - Έλεγχος μέσω προσομοίωσης (simulation)
  - Λειτουργική Επαλήθευση (formal verification) !
  - Μπορούμε να το συνθέσουμε!
- Στόχος
  - Αξιόπιστη διεργασία σχεδίασης με χαμηλές απαιτήσεις κόστους και χρόνου
  - Αποφυγή και πρόληψη λαθών σχεδίασης

Βασικό Block: Module

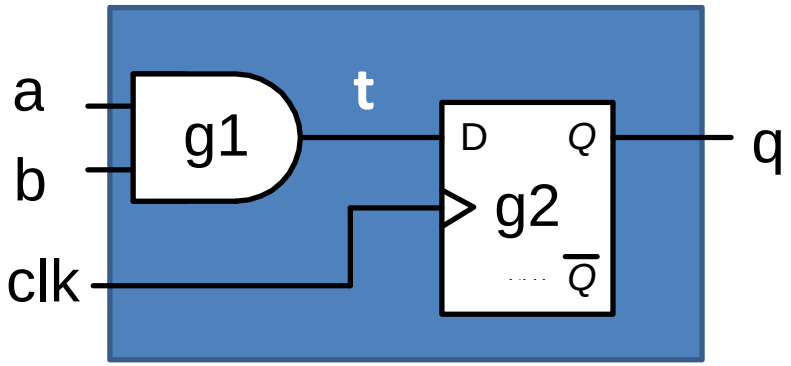

```
module and_ff(a, b, q, clk);
input a, b;
output q;
…
…
endmodule
```

Βασικό Block: Module



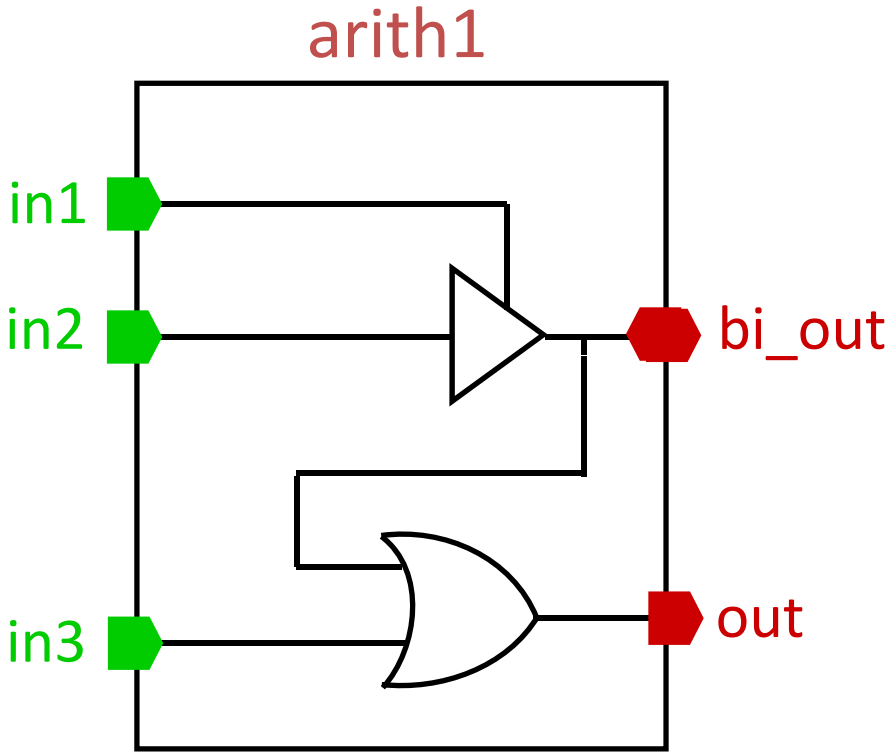

```
module and_ff(a, b, q, clk);
input a, b, clk;
output q;




endmodule
```

Βασικό Block: Module




```
module and_ff(a, b, q, clk);
input a, b, clk;
output q;

wire t;
and g1(a, b, t),
d_ff g2(t, clk, q);
endmodule
```

Πόρτες ενός Module

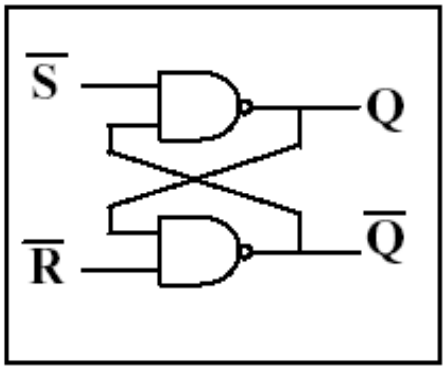



```
module arith1 (bi_out,
out, in1, in2, in3);

inout bi_out;

output out;

input in1, in2;

input in3;

…

…

endmodule
```

Modules vs Instances

- Instantiation είναι η διαδικασία δημιουργίας αντικειμένου από το module.

Storage Cell



$\overline{S}$	$\overline{R}$	Q	$\overline{Q}$
0	0	undef	
0	1	1	0
1	0	0	1
1	1	Q	$\overline{Q}$

```
module nand(out, a, b,);
input a, b;
output out;

wire out = ~ (a & b);

endmodule
```

```
module SRLATCH(Q, Qbar, Sbar, Rbar);
input  Sbar, Rbar;
output Q, Qbar;

// Instantiate lower-level modules
nand n1(Q, Sbar, Qbar);
nand n2(Qbar, Rbar, Q);

endmodule
```

Primitives

- Επίπεδο Πυλών
  - and, nand, or, nor, xor, xnor, not, buf
  - Παράδειγμα:

```
and N25 (out, A, B) // instance name
and #10  (out, A, B) // delay
or  #15 N33(out, A, B) // name + delay
```

Verilog Modeling

- Structural
  - Για να συνδέουμε τα modules
    e.g. counter counter_1(clk, enable, count_out);
- Dataflow
  - όταν χρησιμοποιούμε βασικές πύλες
    e.g. wire = (a & b) | (c & d);
- Behavioral or Procedural
  - procedural calls
    e.g. always @(posedge clk) begin … end

Verilog Modeling

- Synthesizable vs Not Synthesizable
  - e.g. a <= #10 b + 1;
- RTL (Register Transfer Level)
  - όταν η σχεδίαση γίνεται σε επίπεδο registers

Συμβάσεις στην γλώσσα Verilog

- Η Verilog είναι case sensitive.
  - Λέξεις κλειδιά είναι σε μικρά.
- Σχόλια
  - Για μία γραμμή είναι //
  - Για πολλές /* */
- Βασικές τιμές 1-bit σημάτων
  - 0: λογική τιμή 0.
  - 1: λογική τιμή 1
  - x: άγνωστη τιμή
  - z: ασύνδετο σήμα, high impedance

Αριθμοί

- Αναπαράσταση αριθμών
  - <size>’ <base_format> <number>

    <size> δείχνει τον αριθμό απο bits

    <base_format> μπορεί να είναι : d, h, b, o (default: d)
  - Όταν το <size> λείπει το μέγεθος καθορίζεται από τον compiler
  - Όταν το <number> έχει πολλα ψηφία μπορούμε να το χωρίζουμε με _ (underscore) όπου θέλουμε

- 100 // 100
- 4’b1111 // 15, 4 bits
- 6’h3a // 58, 6 bits
- 6’b111010 // 58, 6 bits
- 12’h13x // 304+x, 12 bits
- 8’b10_10_1110 // 174, 8 bits

Τελεστές (Operators)

- Arithmetic + - * / %
- Logical ! && ||
- Relational < > <= >=
- Equality == !=
- Bit-wise ~ & | ^ ^~ (ή ~^)
- Reduction & ~& | ~| ^ ^~(ή ~^)
- Shift << >>
- Concatenation/Replication {A,B,…} {4{A}} (πολλούς τελεστέους)
- Conditional x ? y : z (3 τελεστέους)

* Εφαρμόζεται μόνο σε έναν τελεστέο

Τελεστές (Operators)

- Arithmetic + - * / %
- Logical ! && ||
- Relational < > <= >=
- Equality == !=
- Bit-wise ~ & | ^ ^~ (ή ~^)
- Reduction & ~& | ~| ^ ^~(ή ~^)
- Shift << >>
- Concatenation/Replication {A,B,…} {4{A}} (πολλούς τελεστέους)
- Conditional x ? y : z (3 τελεστέους)

ΠΡΟΣΟΧΗ
στην υλοποίηση

* Εφαρμόζεται μόνο σε έναν τελεστέο

Χρόνος Προσομοίωσης

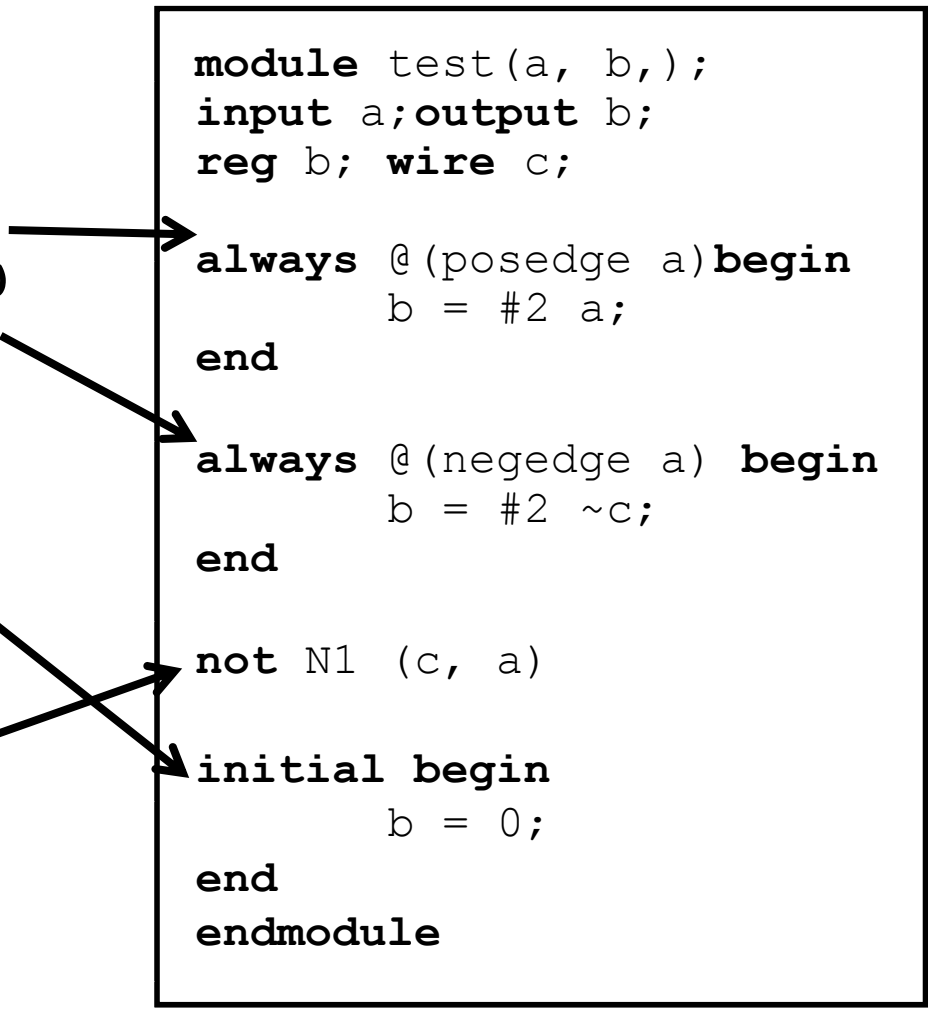
- `timescale <time_unit>/<time_precision>
  - time_unit: μονάδα μέτρησης χρόνου
  - time_precision: ελάχιστο χρόνο βήματα κατά την προσομοίωση.
  - Μονάδες χρόνου : s, ms, us, ns, ps, fs
- #<time> : αναμονή για χρόνο <time>
  - #5 a=8'h1a
- @ (<σήμα>): αναμονή μέχρι το σήμα να αλλάξει τιμή (event)
  - @ (posedge clk)// θετική ακμή
  - @ (negedge clk)// αρνητική ακμή
  - @ (a)
  - @ (a or b or c)

Module Body

- declarations
- always blocks:

  Μπορεί να περιέχει πάνω από ένα
- initial block:

  Μπορεί να περιέχει ένα ή κανένα.
- modules/primitives instantiations



```
module test(a, b,);
input a;output b;
reg b; wire c;

always @(posedge a)begin
        b = #2 a;
end

always @(negedge a) begin
        b = #2 ~c;
end

not N1 (c, a)

initial begin
        b = 0;
end
endmodule
```

Τύποι μεταβλητών στην Verilog

- integer // αριθμός
- wire // καλώδιο – σύρμα
- reg // register
- tri // tristate

Wires

- Συνδυαστική λογική (δεν έχει μνήμη)
- Γράφος εξαρτήσεων
- Μπορεί να περιγράψει και ιδιαίτερα πολύπλοκη λογική…

```
wire sum  = a ^ b;
wire c = sum | b;
wire a = ~d;
```

```
wire sum;
...
assign sum = a ^ b;
```

```
wire muxout = (sel == 1) ? a : b;
wire op = ~(a & ((b) ? ~c : d) ^ (~e));
```

Σύρματα και συνδυαστική λογική

- module … endmodule
- Δήλωση εισόδων - εξόδων
- Concurrent statements

```
module adder(a, b, sum, cout);
input  a, b;
output sum, cout;

wire sum  = a ^ b;
wire cout = a & b;

endmodule
```

Regs και ακολουθιακή λογική

- Στοιχεία μνήμης

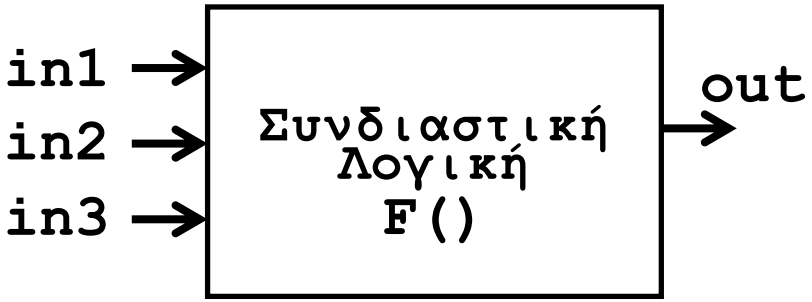
  … κάτι ανάλογο με μεταβλητές στη C
- Μόνο regs (οχι wires) παίρνουν τιμή σε initial και always blocks.
  - Χρήση των begin και end για grouping πολλών προτάσεων
- Όπου χρησιμοποιούμε reg δεν σημαίνει οτι θα συμπεριφέρεται σαν καταχωρητής !!!

```
reg a;

initial begin
 a = 0;
 #5;
 a = 1;
end
```

```
reg q;

always @(posedge clk)
begin
  q = #2 (load) ? d : q;
end
```

Regs και συνδυαστική λογική

Αν η συνάρτηση F() είναι πολύπλοκη τότε



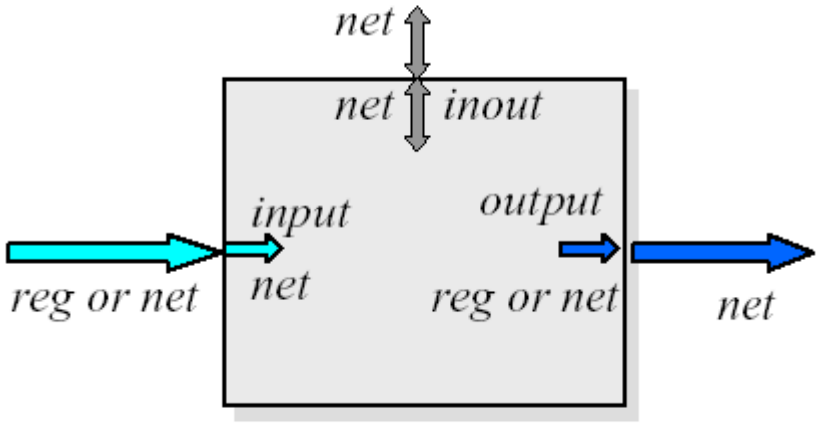

```
reg out;
always @(in1 or in2 or in3)
        out = f(in1,in2,in3);
```

Ισοδύναμα

```
reg out;

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

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

Κανόνες Πορτών Module



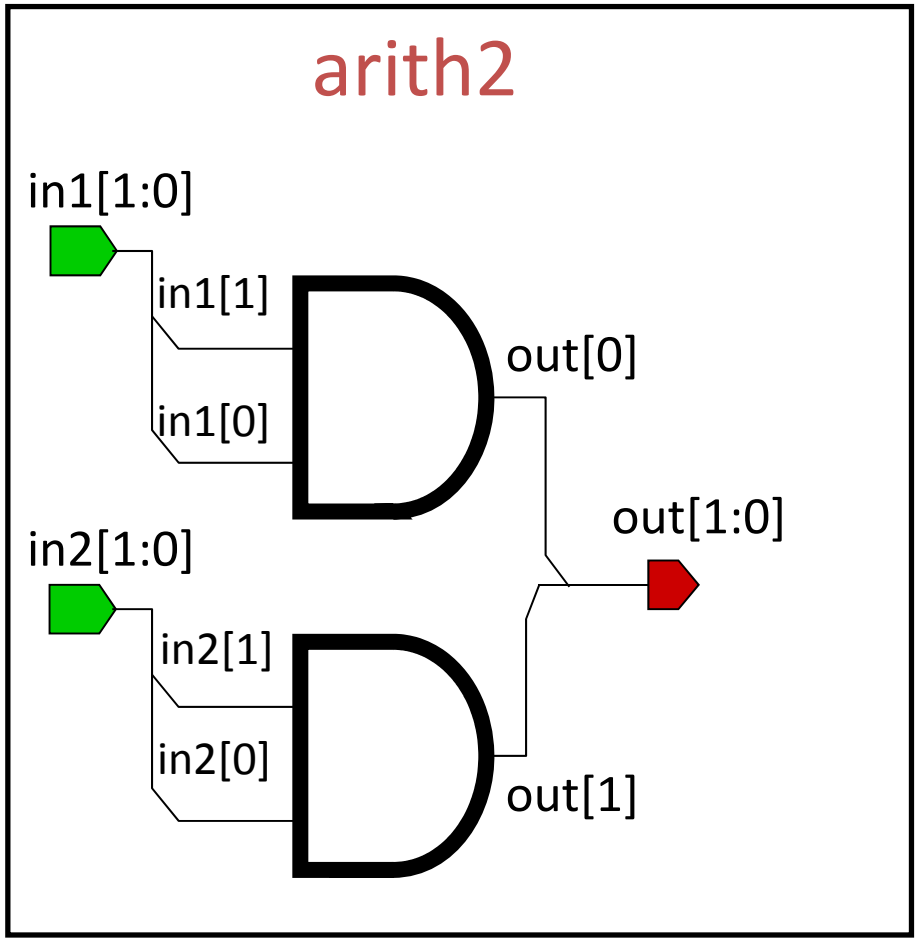
- Τα input και inout έχουν τύπο wire μέσα στο module
- Τα outputs μπορεί να έχουν τύπο wire ή reg

Συνδέσεις μεταξύ Instances

- Με βάση την θέση
  - module adder(Sum, In1, In2)
  - adder add1(A, B, C) // Sum = A, In1 = B, In2 = C

- Συσχετίζοντας ονόματα (το καλύτερο)
  - module adder(Sum, In1, In2)
  - adder add1(.In2(B), .In1(A), .Sum(C))
    // Sum = C, In1 = A, In2 = B

Buses (1/2)




```
module arith2 (out, in1,
in2);

output [1:0] out;

input [1:0] in1, in2;

…

…

endmodule
```

Buses (2/2)

- Καμία διαφορά στη συμπεριφορά
- Συμβάσεις:
  - [high : low]
  - [msb : lsb]
- Προσοχή στις αναθέσεις (μήκη) και τις συνδέσεις εκτός του module…

```
module adder(a, b, sum, cout);

input  [7:0] a, b;
output [7:0] sum;
output       cout;

wire [8:0] tmp = a + b;

wire [7:0] sum  = tmp[7:0];
wire       cout = tmp[8];

endmodule
```

Συνθέσιμος Κώδικας

- Ο Synthesizable κώδικας μπορεί να γίνει synthesize και να πάρουμε gate-level μοντέλο για ASIC/FPGA.

  π.χ.

```
wire [7:0] sum  = tmp[7:0] & {8{a}};
wire cout = tmp[8];
```

- Non-synthesizable κώδικας χρησιμοποιείται μόνο για προσομοίωση και αφαιρείται κατά την διαδικασία της σύνθεσης (logic synthesis).

  π.χ.

```
initial begin
    a = 0; b = 0;
    #5 a = 1;
    b = 1;
end
```

Χρήση Καθυστέρησης στην Verilog

- Λειτουργική Επαλήθευση - Functional Verification (RTL Model)
  - Η καθυστέρηση είναι προσεγγιστική. Π.χ.

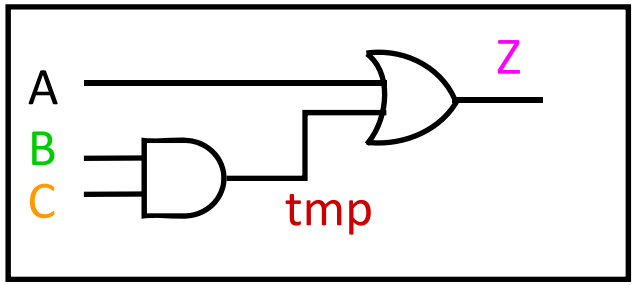
```
always @(posedge clk)
   q <= #2 d; // FF με 2 μονάδες καθυστέρηση
```

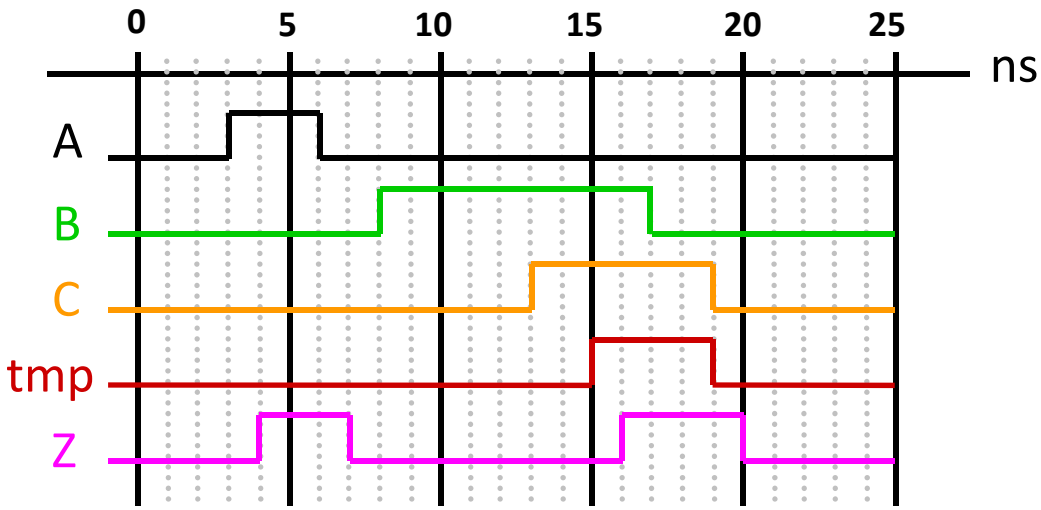
  - Συνήθως θεωρούμε ότι η συνδυαστική λογική δεν έχει καθυστέρηση. π.χ.

```
wire a = (b & c) | d;
// μόνο την λειτουργία όχι καθυστέρηση πυλών
```

  - Η καθυστέρηση χρησιμοποιείται κυρίως στο testbench κώδικα για να φτιάξουμε τα inputs.
- Χρονική Επαλήθευση - Timing Verification
  - Αναλυτικά κάθε πύλη έχει καθυστέρηση.
  - Συνήθως κάνουμε timing verification σε gate-level model το οποίο φτιάχνεται από ένα synthesis tool.

Οι πύλες έχουν καθυστέρηση !!!





Έστω καθυστερήσεις: T_{and} = 2ns και T_{or} = 1ns

- Έστω ότι τα καλώδια δεν έχουν καθυστέρηση

3 Μονοπάτια (paths) προς την έξοδο:

- A➔Z, (1 ns)
- B➔tmp➔Z (3 ns)
- C➔tmp➔Z (3 ns)

Η συμπεριφορά του κυκλώματος φαίνεται στις κυματομορφές (waveforms)

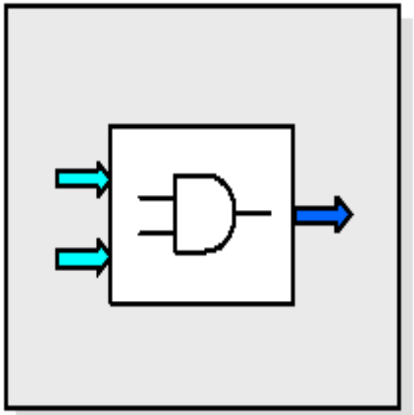
```
always@(a or b or c)
begin
   tmp = #2 B & C;
   Z   = #1 A | tmp;
end
```

Testing

- Ιεραρχικός Έλεγχος
- Κάθε module ξεχωριστά

  Block level simulation

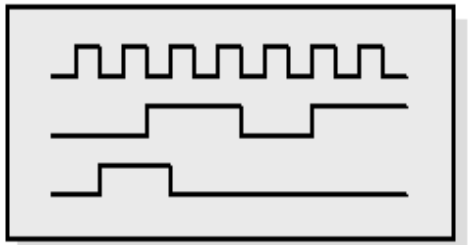
  – Έλεγχος των προδιαγραφών, της λειτουργίας και των χρονισμών των σημάτων



- Όλο το design μαζί

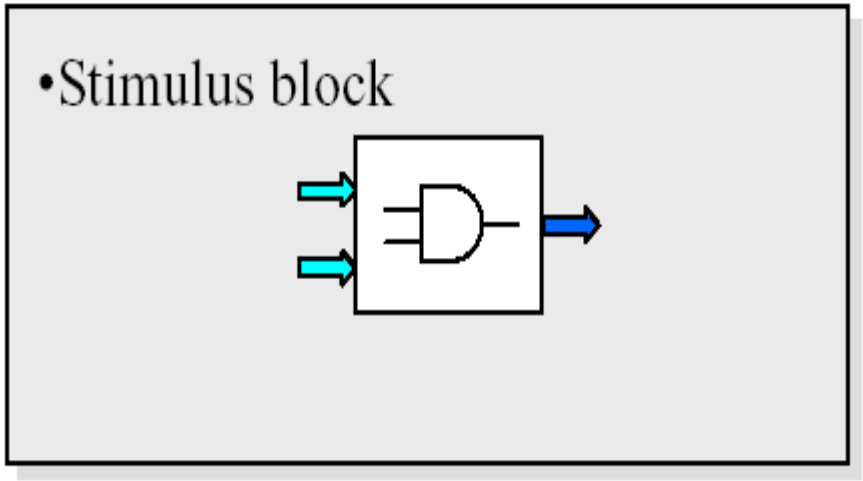
  System level simulation

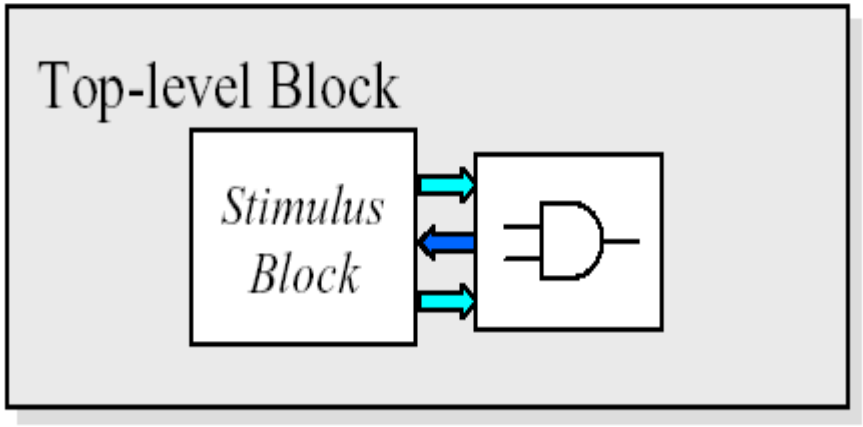
  – Έλεγχος της συνολικής λειτουργίας και των διεπαφών



Έλεγχος σωστής λειτουργίας

- Testbench : top module που κάνει instantiate το module που τεστάρουμε, δημιουργεί τις τιμές των εισόδων του (stimulus) και ελέγχει ότι οι έξοδοί του παίρνουν σωστές τιμές.
- 2 προσεγγίσεις :
  - Έλεγχος εξόδων και χρονισμού με το μάτι
  - Έλεγχος εξόδων και χρονισμού μέσω κώδικα δλδ. αυτόματη σύγκριση των αναμενόμενων εξόδων.



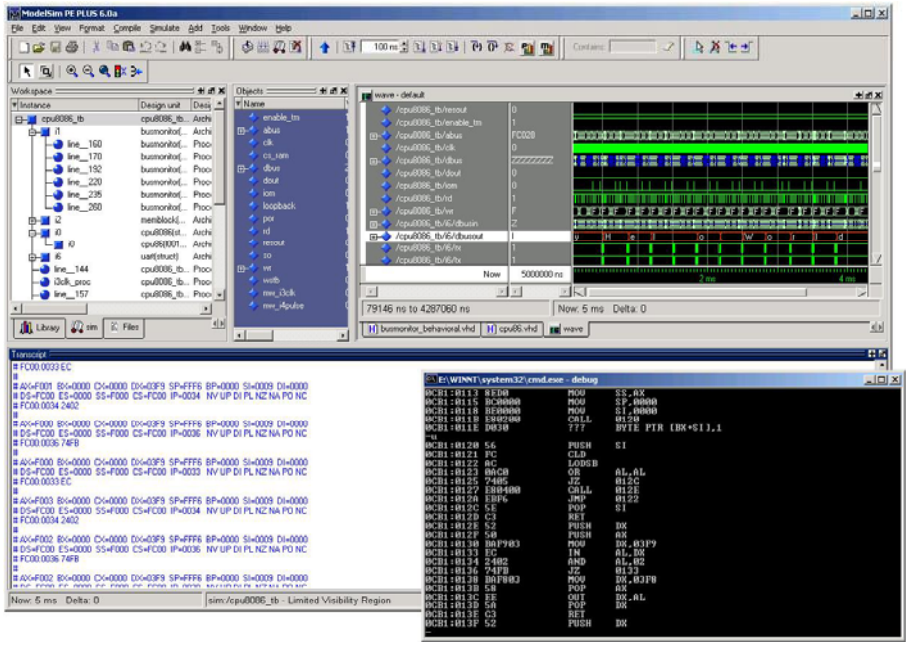


Ένα απλό «test bench»

```
module tb_half_add;
reg  a, b;
wire s, c;

half_add add0(a, b, s, c);

initial begin
 monitor("a: %x, b: %x, s: %x, c: %x", a, b, s, c);
 a = 0; b = 0;
 #5 a = 1;
 #5 b = 1;
 #5 a = 0;

end

endmodule
```

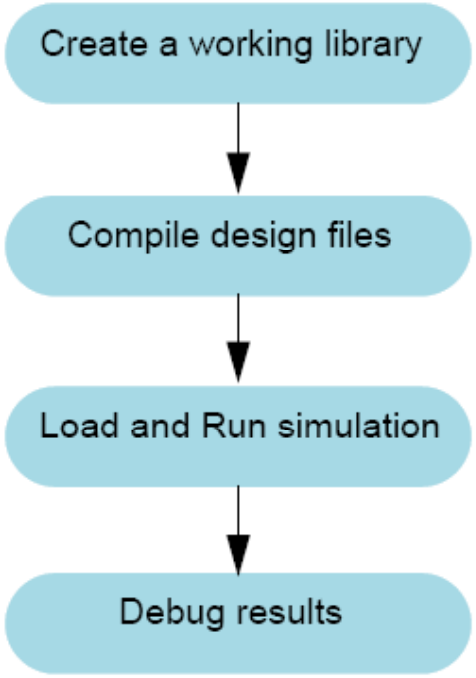
```
module half_add(a, b, sum, cout);
input  a, b;
output sum, cout;

wire sum  = a ^ b;
wire cout = a & b;
endmodule
```

Modelsim



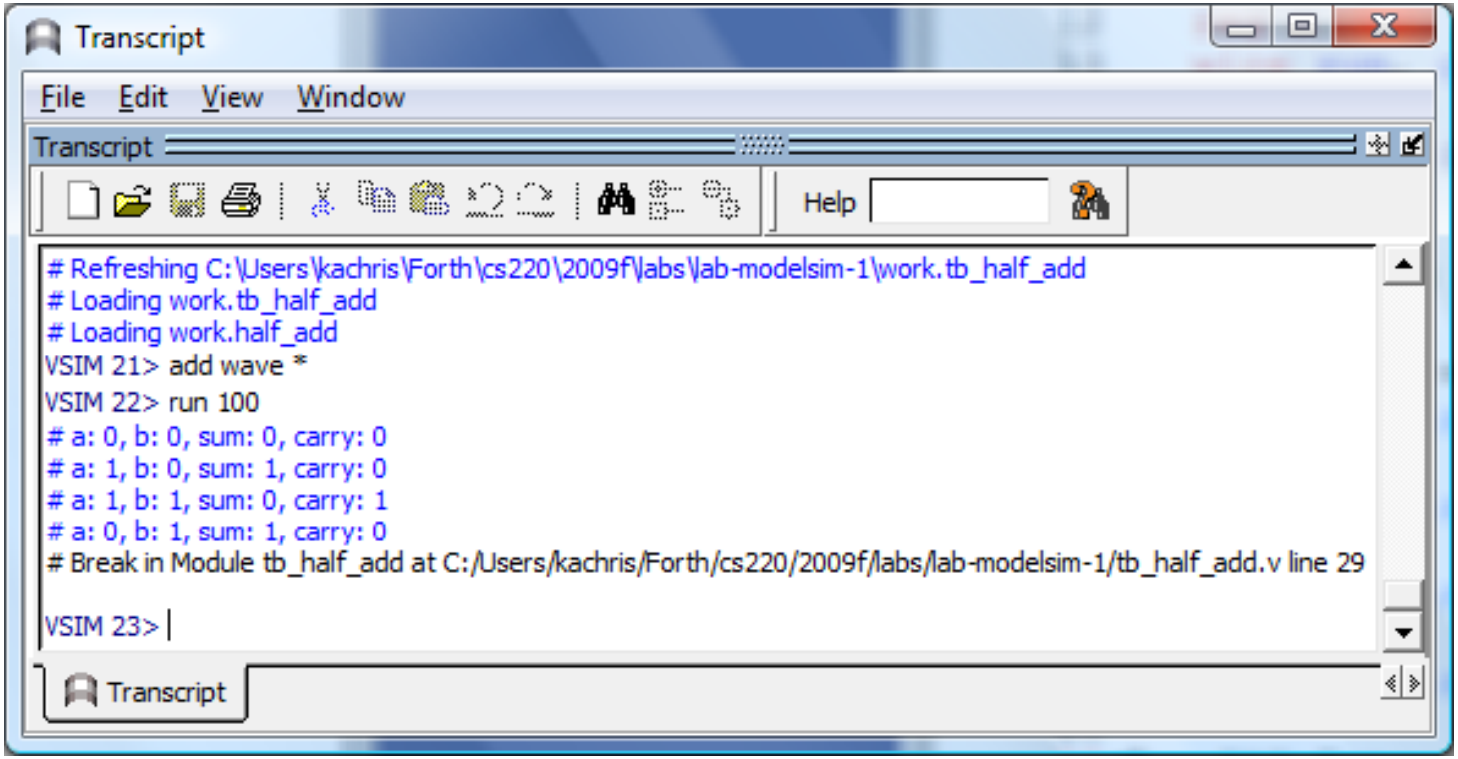
- Χρησιμοποιείται για την προσομοίωση κυκλωμάτων σε Verilog, VHDL
- Ουσιαστικά είναι ένας debugger για HDLs

Modelsim Introduction

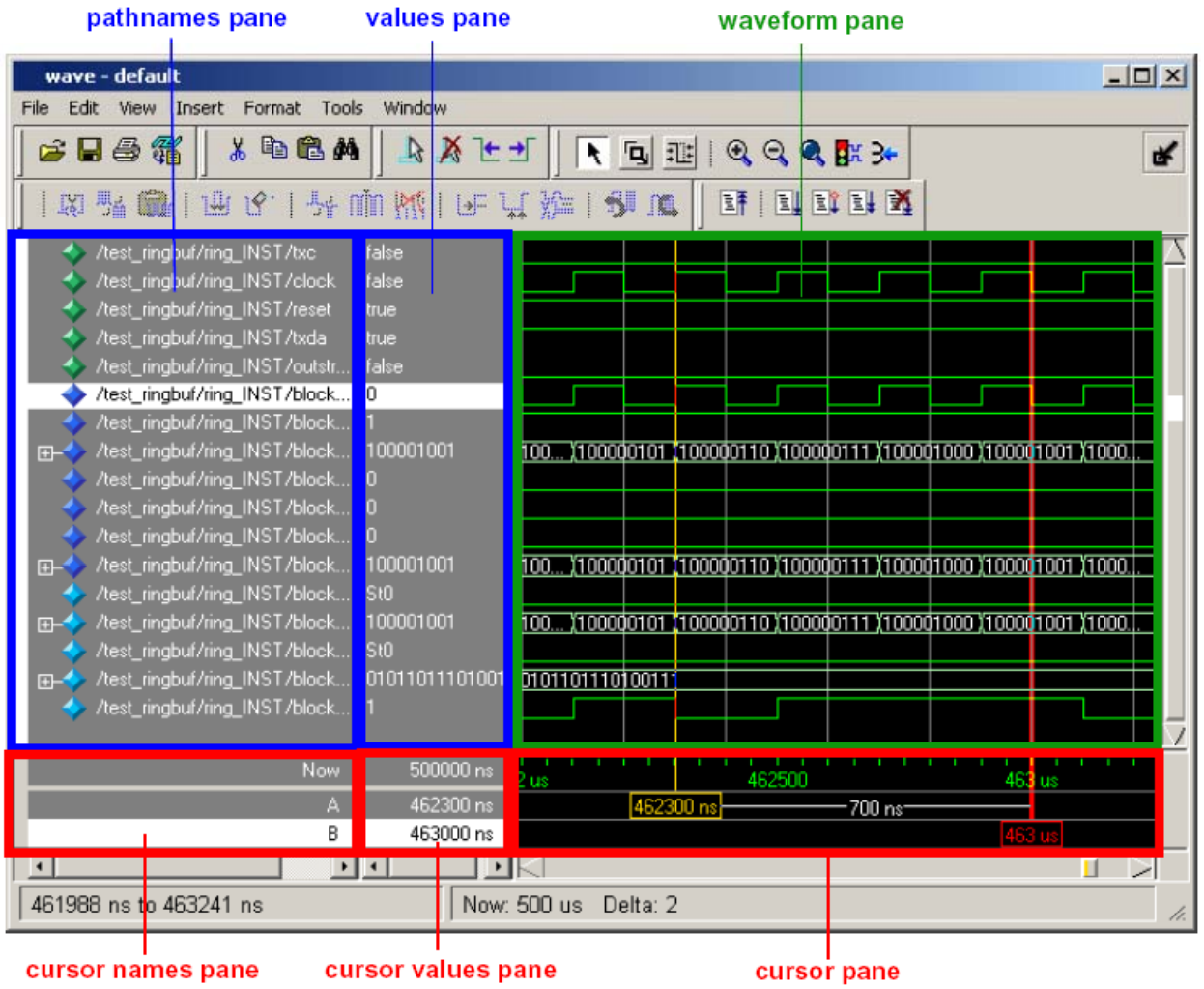


1. Δημιουργία βιβλιοθήκης
2. Compile του κώδικα
3. Εκτέλεση και προσομοίωση

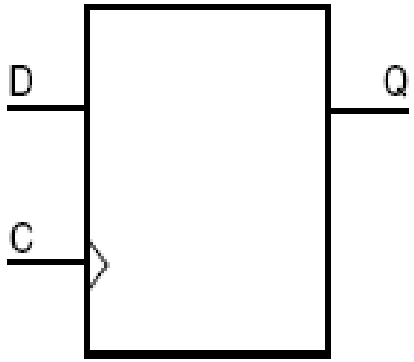
Simulator


Transcript
File Edit View Window
Transcript
Help
Refreshing C:\Users\kachris\Forth\cs220\2009f\labs\lab-modelsim-1\work.tb_half_add
Loading work.tb_half_add
Loading work.half_add
VSIM 21> add wave *
VSIM 22> run 100
a: 0, b: 0, sum: 0, carry: 0
a: 1, b: 0, sum: 1, carry: 0
a: 1, b: 1, sum: 0, carry: 1
a: 0, b: 1, sum: 1, carry: 0
Break in Module tb_half_add at C:/Users/kachris/Forth/cs220/2009f/labs/lab-modelsim-1/tb_half_add.v line 29
VSIM 23>
Transcript

Modelsim Window



DFF: Behavioral Module



D	CLK	Q
0	↑	0
1	↑	1

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

always @(posedge clk)
begin
    q <= d;
end
endmodule
```

Sensitivity lists

- Λογικές εκφράσεις με or
- posedge και negedge
  - Ρολόγια, reset

```
always @(posedge clk or negedge rst_)
 ...

always @(opcode or b or c)
  if (opcode == 32'h52A0234E)
    a = b ^ (~c);

always @(posedge a or posedge b)
 ...
always @(posedge a, posedge b)
always #5 clk=~clk
```

• Παράλειψη παραγόντων RHS και αυτών που γίνονται “read” δίνουν λάθη στην προσομοίωση

• Προσοχή στο hardware που θέλουμε να περιγράψουμε…

Conditional Statements – If… Else …

- Το γνωστό
  if … else …
- Μόνο μεσα σε blocks !
- Επιτρέπονται πολλαπλά και nested ifs
  - Πολλά Else if …
- Αν υπάρχει μόνο 1 πρόταση δεν χρειάζεται begin … end

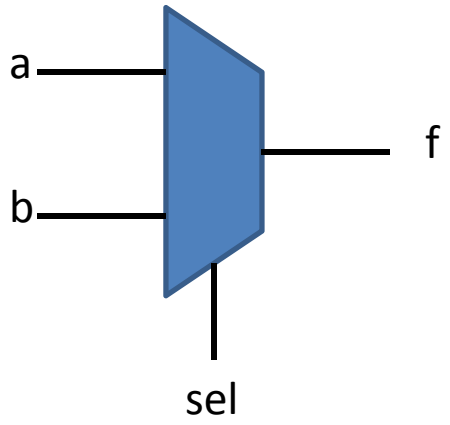
```
module mux(a , b , sel ,
out );

input [4:0] a, b;
input sel;
output [4:0] out;

reg [4:0] out;

always @(a or b or sel) begin
  if ( sel == 0 ) begin
       out <= a;
  end
  else
       out <= b;
end

endmodule
```

Case Statement

- Το γνωστό case
- Μόνο μεσα σε blocks !
- Μόνο σταθερές εκφράσεις
- Δεν υπάρχει break !
- Υπάρχει default !

```
module mux (a , b , c , d ,
sel , out );
input [4:0] a, b, c ,d;
input [1:0] sel;
output [4:0] out;

reg [4:0] out;
always @(a or b or c or d or
sel ) begin
  case (sel)
    2'b00: out <= a;
    2'b01: out <= b;
    2'b10: out <= c;
    2'b11: out <= d;
    default: out <= 5'bx;
  endcase
end
endmodule
```

Mux: Behavioral Module



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

```
module sel(a, b, sel, f);
input  a, b, sel;
output f;
reg f;
always @(sel or a or b)
begin
  if (sel==0)
    f <= a;
  else
    f <= b;
end
endmodule
```

Initial and always block

```
initial
begin
   // run once
   a=0; b=0;
   #5; a=1; b=1;
end
```

- Runs when simulation starts
- Terminates when control reaches the end
- Good for providing stimulus
- not synthesizable

```
always@(...)
begin
    // run always
    a <= b & c;
end
```

- Runs when simulation starts
- Restart when control reaches the end
- Good for modeling hardware
- maybe synthesizable

Παραμετρικά modules

```
module RegLd( D, Q,
              load, clk);
parameter N = 8;
parameter dh = 2;
input  [N-1:0] D;
output [N-1:0] Q;
input  load, clk;
reg [N-1:0] Q;

always @(posedge clk)
 if (load)
  Q = #dh D;

endmodule
```

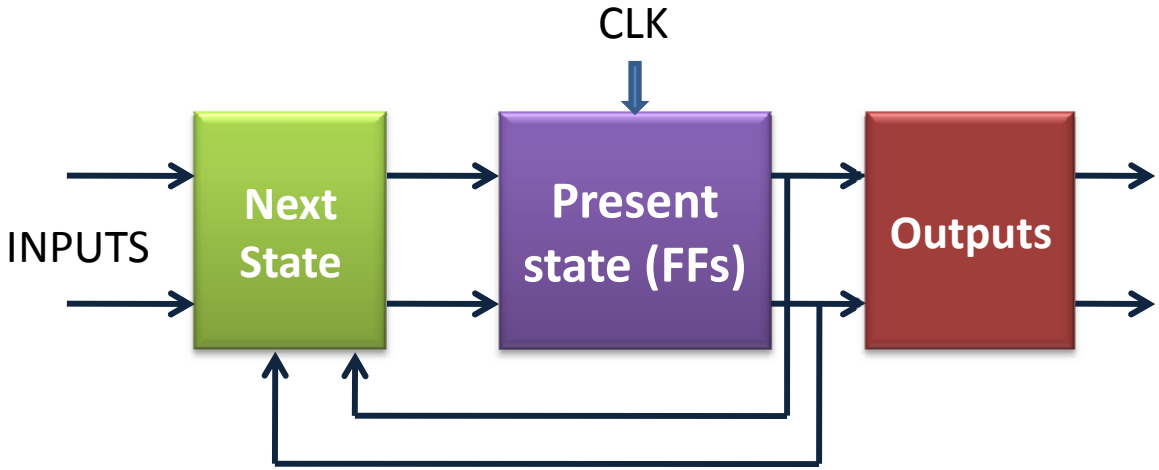
- Μπορούμε να έχουμε παραμέτρους σε ένα module
- Default μέγεθος
- … πολύ βολικό!

```
RegLd     reg0(d0, q0, ld, clk);

RegLd #(16,2) reg1(d1, q1, ld, clk);

RegLd     reg2(d2, q2, ld, clk);
defparam  reg2.N = 4;
defparam  reg2.dh = 4;
```

FSMs

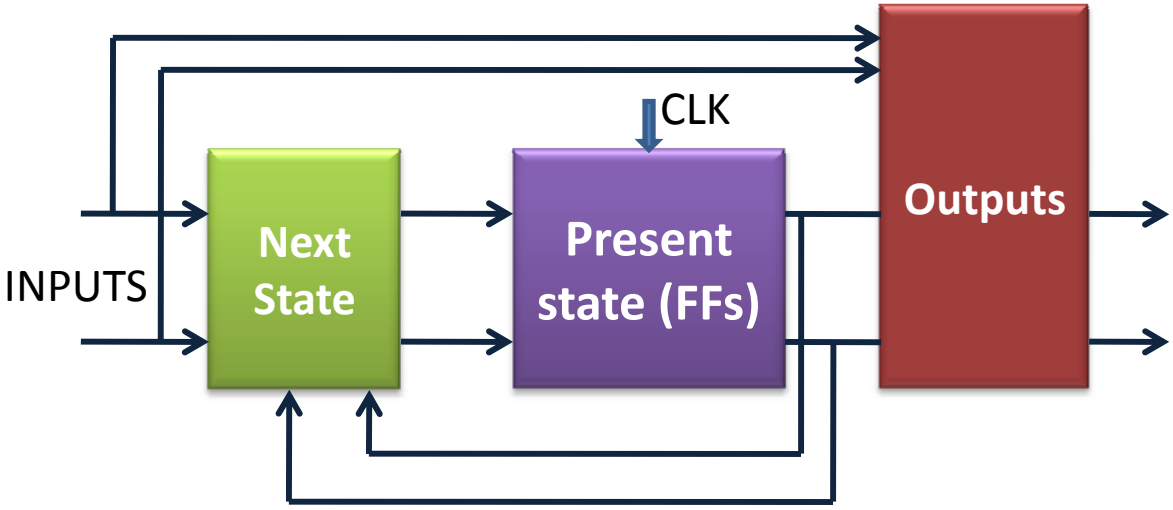
- Οι μηχανές πεπερασμένων καταστάσεων Finite State Machines (FSMs)
  - πιο αφηρημένος τρόπος να εξετάζουμε ακολουθιακά κυκλώματα
  - Είσοδοι, έξοδοι, τρέχουσα κατάσταση, επόμενη κατάσταση
  - Σε κάθε ακμή του ρολογιού συνδυαστική λογική παράγει τις εξόδους και την επόμενη κατάσταση σαν συνάρτησεις των εισόδων και της τρέχουσας κατάστασης.

Moore – Mealy Dataflow



Moore

Οι έξοδοι είναι συνάρτηση της κατάστασης



Mealy

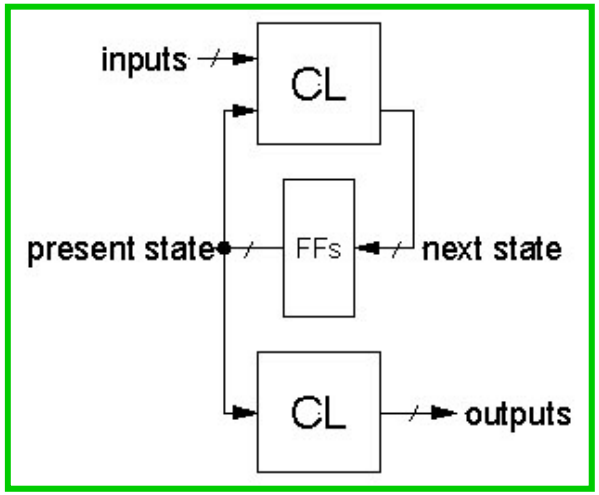
Οι έξοδοι είναι συνάρτηση της κατάστασης και των εισόδων

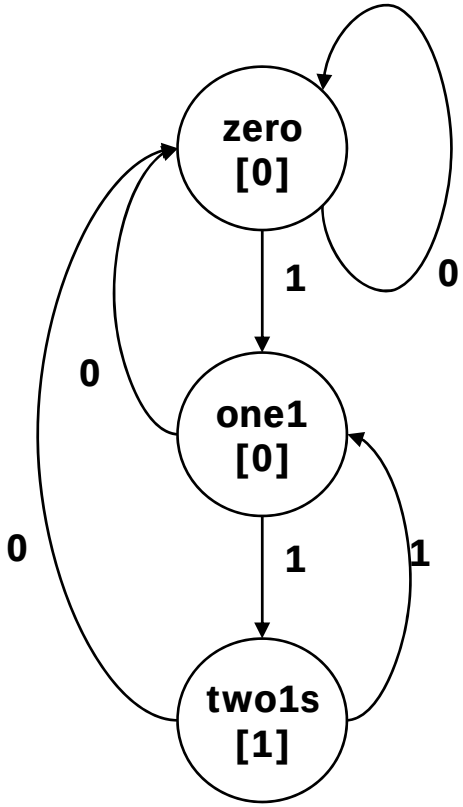
Βήματα Σχεδίασης

1. Περιγραφή λειτουργία του κυκλώματος (functional specification)
2. Διάγραμμα μετάβασης καταστάσεων (state transition diagram)
3. Πίνακας καταστάσεων και μεταβάσεων με συβολικά ονόματα (symbolic state transition table)
4. Κωδικοποίηση καταστάσεων (state encoding)
5. Εξαγωγή λογικών συναρτήσεων
6. Διάγραμμα κυκλώματος
   - FFs για την κατάσταση
   - ΣΛ για την επόμενη κατάσταση και τις εξόδους

Παράδειγμα FSM – Count couples

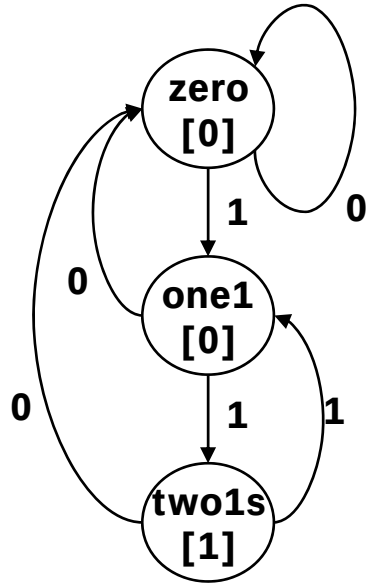
- Η έξοδος γίνεται “1” για κάθε ζευγάρι από 1
  - Moore FSM





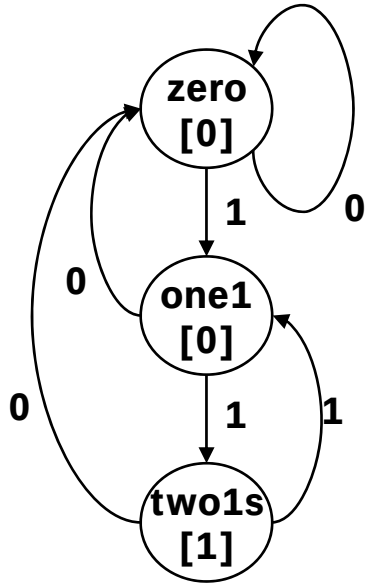
Moore FSM: general & state

```
module moore(Out, Clock, Reset, In);
  output  Out;
  input   Clock, Reset, In;

  reg     Out;
  reg [1:0]  CurrentState; // state reg
  reg [1:0]  NextState;

  // State assignment
  parameter
       STATE_Zero = 2'h0,
       STATE_One1 = 2'h1,
       STATE_Two1s =2'h2,
       STATE_X =    2'hX;

  // Implement the state register
  always @( posedge Clock) begin
    if (Reset) CurrentState <= STATE_Zero;
    else CurrentState <= NextState;
  end
```



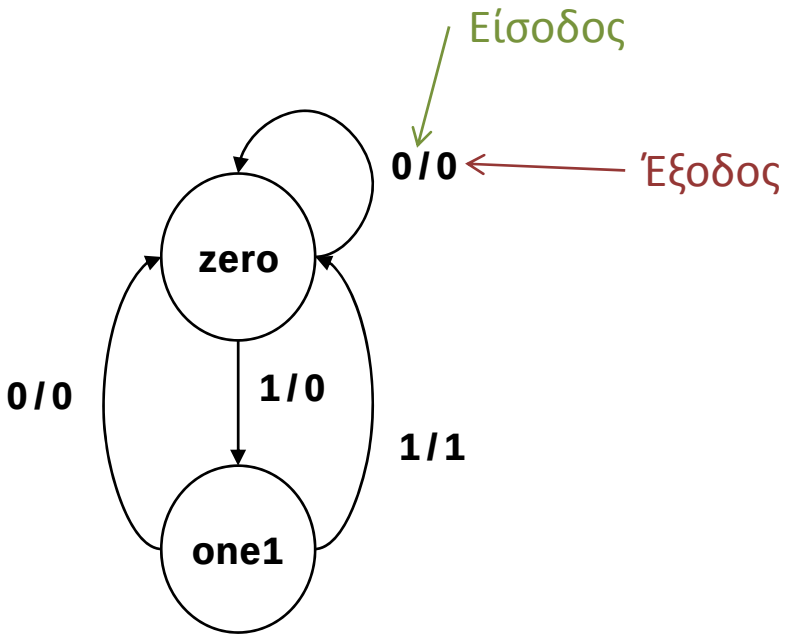
Moore FSM : combinatorial

```
always @(In or CurrentState) begin
  NextState =  CurrentState;
  Out = 1'b0;
  case (CurrentState)
    STATE_Zero: begin // last input was a zero
      if (In) NextState =     STATE_One1;
    end
    STATE_One1: begin // we've seen one 1
      if (In) NextState =     STATE_Two1s;
      else    NextState =     STATE_Zero;
    end
    STATE_Two1s: begin // we've seen at 2 ones
      Out = 1;
      if (In) NextState = STATE_One1;
      else    NextState = STATE_Zero
    end
    default: begin // in case we reach a bad state
      Out = 1'bx;
      NextState =      STATE_Zero;
    end
  endcase
end
```



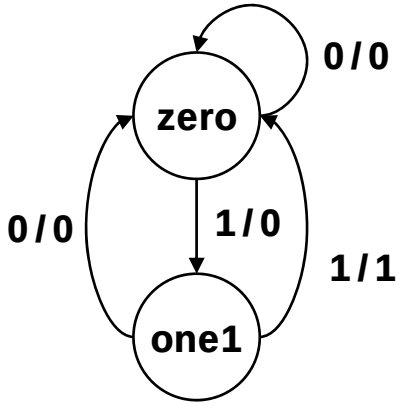
Παράδειγμα FSM – Count couples

- Η έξοδος γίνεται “1” για κάθε ζευγάρι από 1
  - Mealy FSM

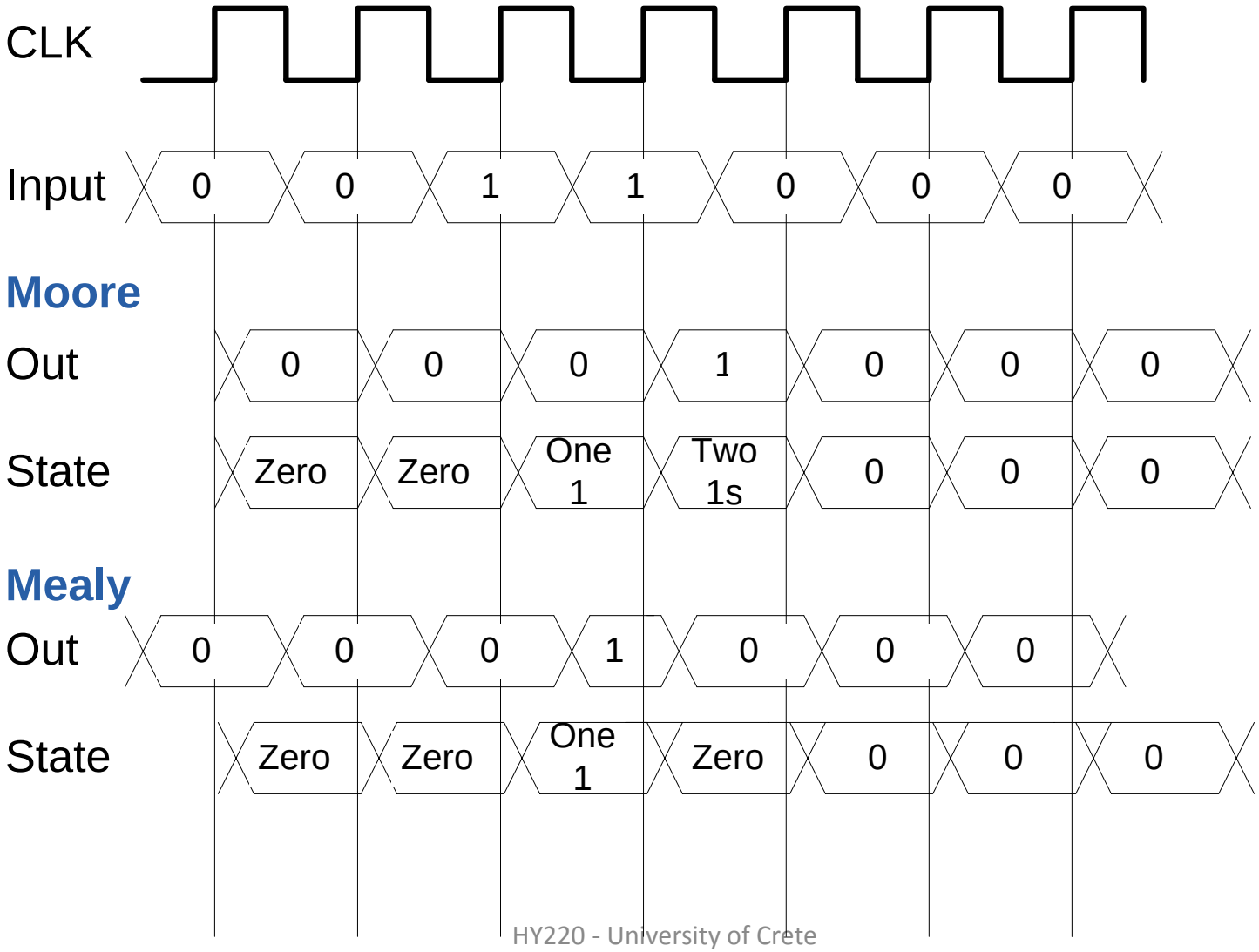


Mealy FSM

```
module mealy(Clock, Reset, In, Out);
    input  Clock, Reset, In;
    output Out;
    reg    Out;
    reg    CurrentState;// state register
    reg    NextState;

    parameter STATE_Zero =    1'b0,
              STATE_One1 =    1'b1;

    always @(posedge Clock) begin
        if (Reset) CurrentState <= STATE_Zero;
        else CurrentState <= NextState;
    end

    always @ (In or CurrentState) begin
        NextState = CurrentState;
        Out = 1'b0;
        case (CurrentState)
            STATE_Zero: if (In) NextState =  STATE_One1;
            STATE_One1: begin          // we've seen one 1
                         NextState = STATE_Zero;
                         if (In) Out = 1;
                        end
        endcase
    end
endmodule
```



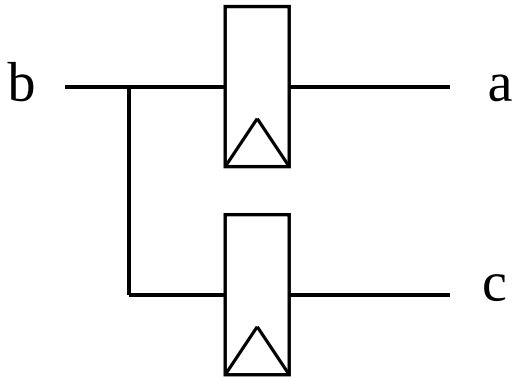
Moore vs Mealy



Αναθέσεις (Assignments)

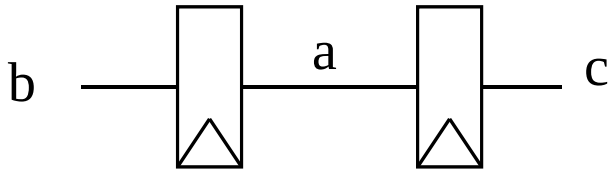
- blocking =

```
always @(posedge clk)
begin
    a = b;
    c = a; // c παίρνει τιμή του b
end
```



- non blocking <=

```
always @(posedge clk)
begin
    a <= b;
    c <= a; // c παίρνει παλιά τιμή του a
end
```



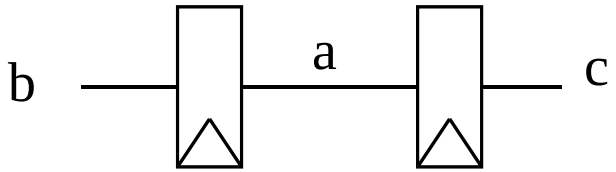
Αναθέσεις (Assignments)

- blocking =
  (η σειρα εχει σημασια!)

```
always @( posedge clk)
begin
Z=Y; Y=X;
y=x; z=y;
```

- non blocking <=
  η σειρα ΔΕΝ εχει σημασια

```
always @(posedge clk)
begin
    a <= b;
    c <= a; // c παίρνει παλιά τιμή του a
end
```



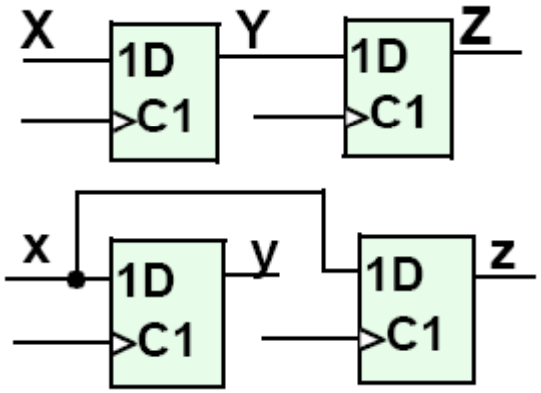
Αναθέσεις (Assignments)

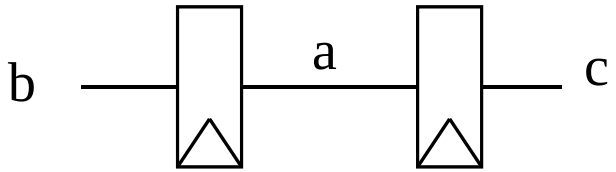
- blocking =
  η σειρα εχει σημασια!

```
always @( posedge clk)
begin
Z=Y; Y=X; // shift register
y=x; z=y; //parallel ff.
```

- non blocking <=
  η σειρα ΔΕΝ εχει σημασια !

```
always @(posedge clk)
begin
    a <= b;
    c <= a; // c παίρνει παλιά τιμή του a
end
```





Assignments: Example

```
time 0  : a = #10 b
time 10 : c = a

a(t=10) = b(t=0)
c(t=10) = a(t=10) = b(t=0)
```

```
time  0 : #10
time 10 : a = b
time 10 : c = a

a(t=10) = b(t=10)
c(t=10) = a(t=10) = b(t=10)
```

```
time 0  : a <= #10 b
time 0  : c <= a

a(t=10) = b(t=0)
c(t=0)  = a(t=0)
```