HY220
Εργαστήριο Ψηφιακών Κυκλωμάτων

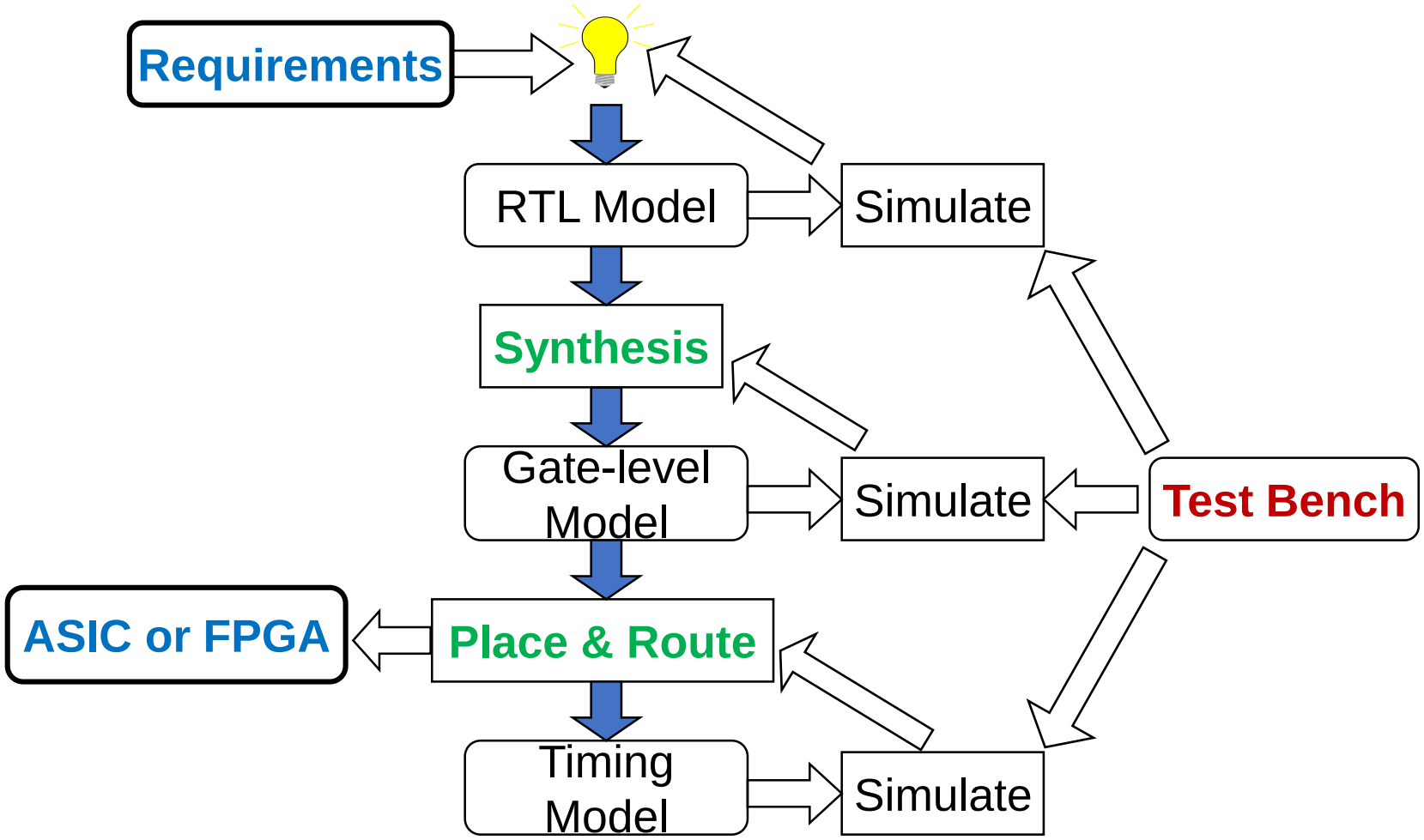
Χειμερινό Εξάμηνο
2019-2020

Verilog: Τα βασικά

Η εξέλιξη στη σχεδίαση ψηφιακών κυκλωμάτων

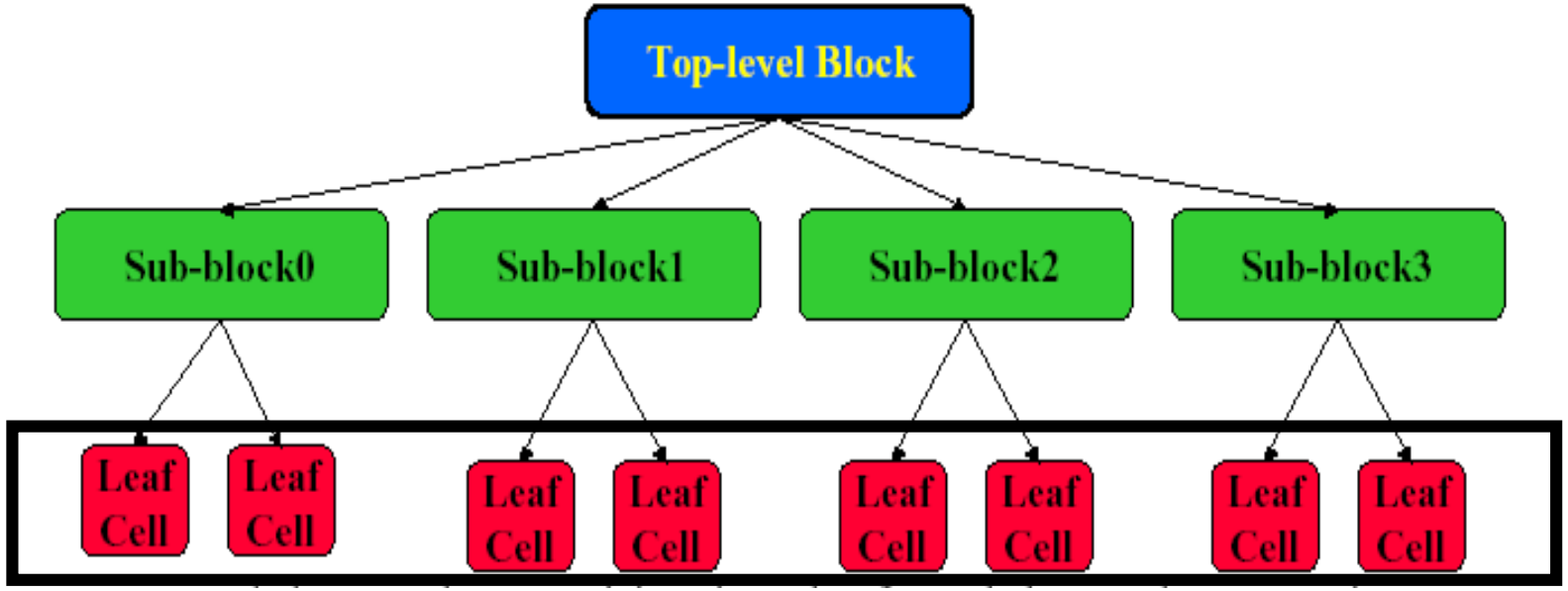
- Μεγάλη εξέλιξη τα τελευταία 40 χρόνια
  - Στις αρχές σχεδιάζαμε με λυχνίες(vacuum tubes) και transistors.
- Μετά ήρθαν τα ολοκληρωμένα (Integrated Circuits – ICs)
  - **SSI:** λίγες πύλες (Small Scale Integration)
  - **MSI:** εκατοντάδες πύλες (Medium Scale Integration)
  - **LSI:** χιλιάδες πύλες (Large Scale Integration)
  - **VLSI:** πολλά εκατομμύρια πύλες (Very Large Scale Integration)
- Ανάγκη για τεχνικές Computer Aided Design (CAD) και γλώσσες περιγραφής υλικού για να μπορούμε να σχεδιάζουμε και να επαληθεύουμε τα κυκλώματα.

Τυπική Ροή Σχεδίασης (Design Flow)



Ιεραρχικές Μεθοδολογίες Σχεδίασης

- Top-Down ή Bottom-Up
  - Συνήθως μια μίξη
- Το τελικό σύστημα αποτελείται από τα Leaf blocks που τρέχουν 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

Αναπαράσταση Ψηφιακών Συστημάτων

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

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

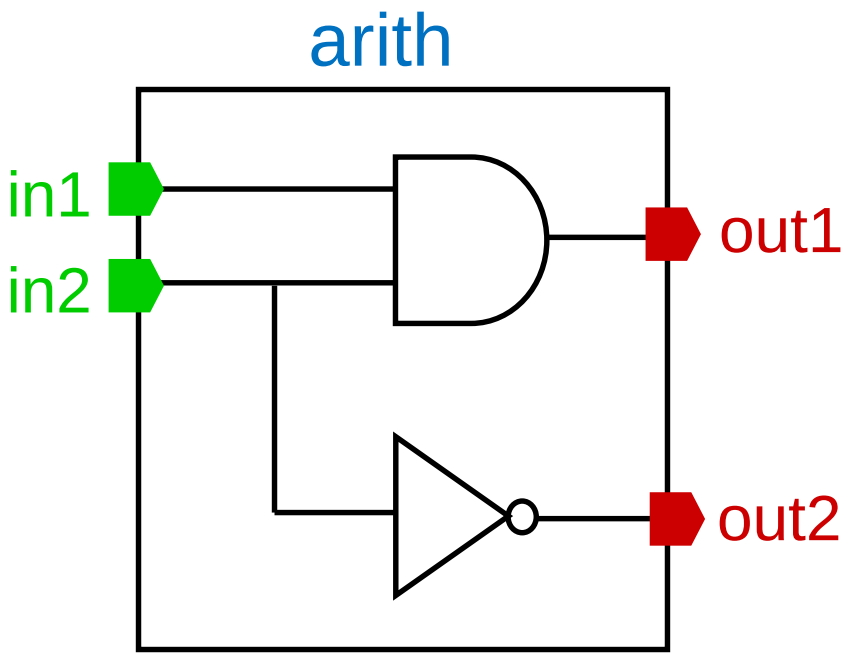
- Αναπαράσταση αριθμών
  - <size>’ <base_format> <number>
    - <size> δείχνει τον αριθμό απο bits
    - <base_format> μπορεί να είναι : d, h, b, o (default: d)
  - Όταν το <size> λείπει το μέγεθος καθορίζεται από τον compiler
  - Όταν το <number> έχει πολλά ψηφία μπορούμε να το χωρίζουμε με _ (underscore) 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 ~ | & ^
- Reduction & | ^ (εφαρμόζεται σε έναν τελεστέο)
- Shift << >>
- Concatenation/Replication {A,B,…} {4{A}} (πολλούς τελεστές)
- Conditional x ? y : z (3 τελεστές)

Βασικό Block: Module

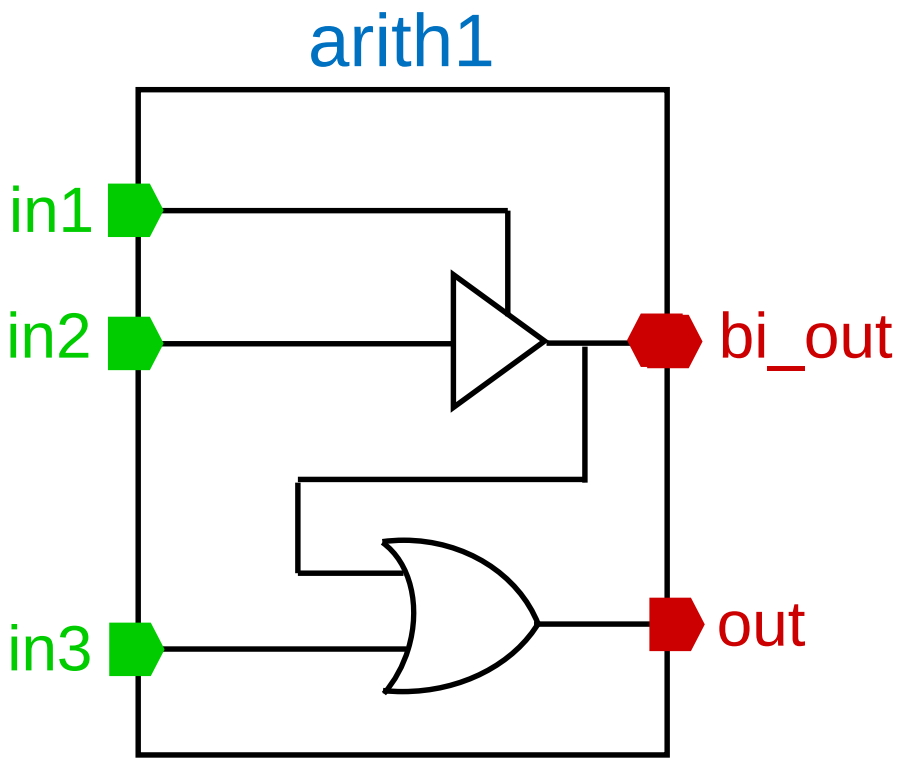



```
module arith (out1, out2, in1, in2);
output out1, out2;
input in1, in2;
……
endmodule
```

ή

```
module arith (
output out1,
output out2,
input in1,
input in2);
……
endmodule
```

Πόρτες ενός Module

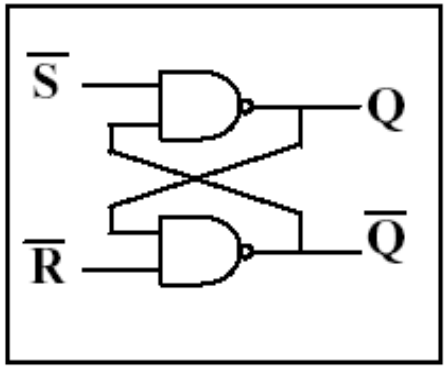



```
module arith1 (
inout bi_out,
output out,
input in1,
input in2,
input in3);
…
…
endmodule
```

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(input a, input b,
output out);

assign out = ~ (a & b);

endmodule
```

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

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


endmodule
```

Primitives

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

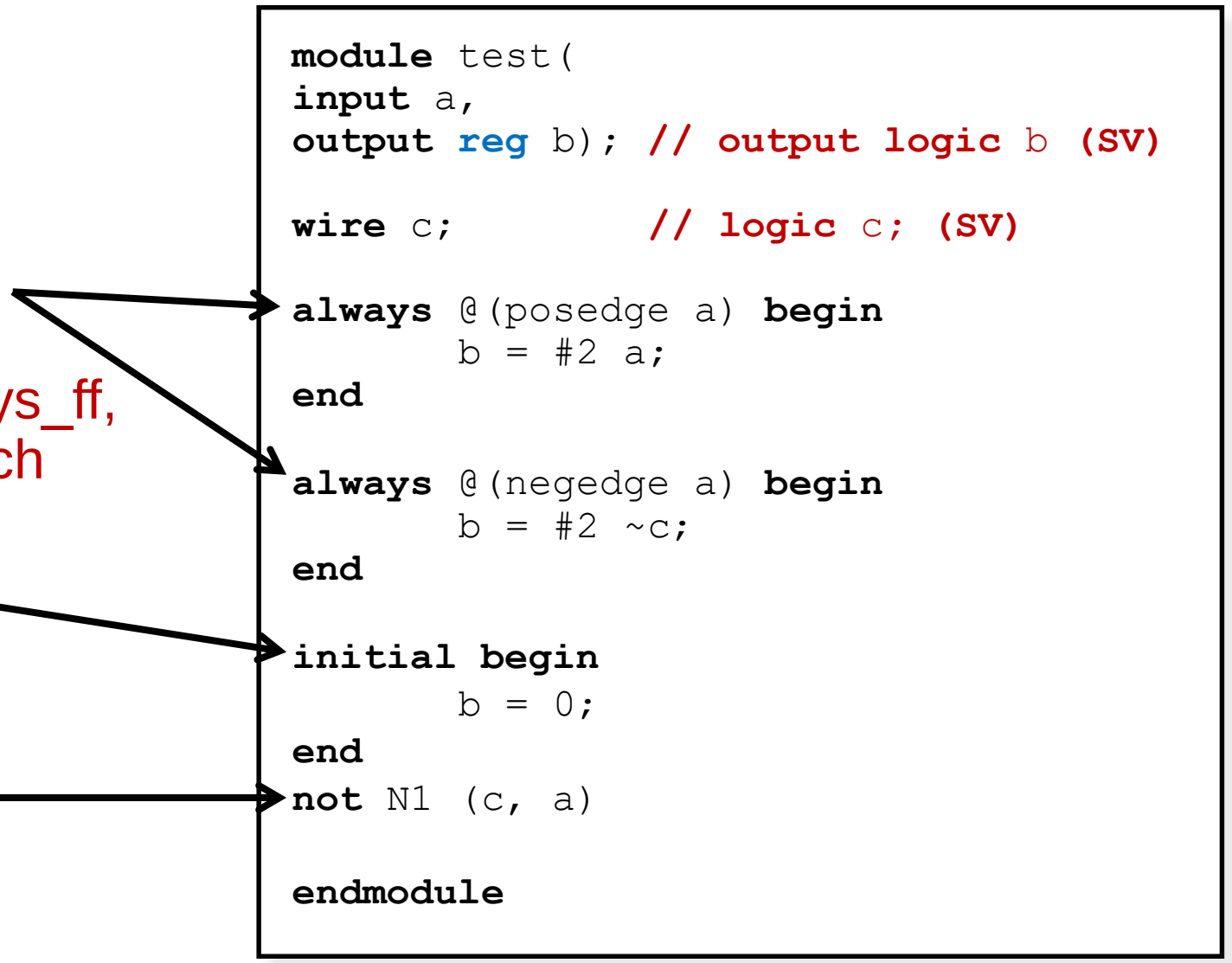
- Παράδειγμα:
  - and N25 (out, A, B) // instance name
  - and #10 (out, A, B) // delay
  - or #15 N33(out, A, B) // name + delay

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
  - SystemVerilog (SV): always_ff, always_comb, always_latch
- initial block:
  - Μπορεί να περιέχει ένα ή κανένα.
- modules/primitives instantiations



```
module test(
input a,
output reg b); // output logic b (SV)

wire c;          // logic c; (SV)

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

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

initial begin
        b = 0;
end
not N1 (c, a)

endmodule
```

Τύποι μεταβλητών στην Verilog

- integer // αριθμός
- wire // καλώδιο – σύρμα
- reg // register
- tri // tristate
- logic // SystemVerilog equivalent for reg and wire

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(input  a, input b, output sum, output cout);

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

endmodule
```

Regs και ακολουθιακή λογική

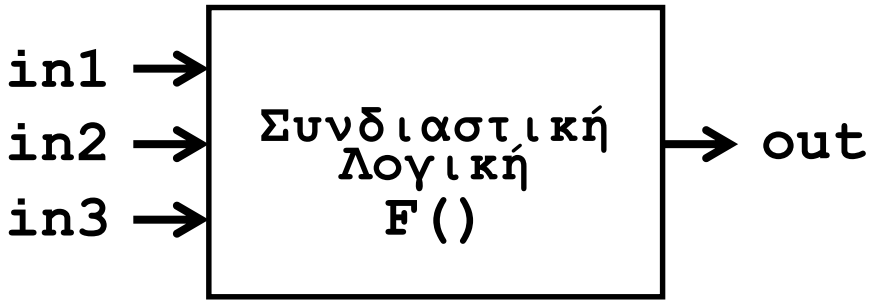
- Στοιχεία μνήμης
  - κάτι ανάλογο με μεταβλητές στη C
- Μόνο regs (οχι wires) παίρνουν τιμή σε initial και always blocks.
  - Χρήση των begin και end για grouping πολλών προτάσεων
- Όπου χρησιμοποιούμε reg δεν σημαίνει ότι θα συμπεριφέρεται σαν καταχωρητής (register) !!!

```
reg q;

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

```
reg a;

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

SystemVerilog και χρήση Logic

- Ο τύπος logic μπορεί να χρησιμοποιηθεί αντί wire **και** αντί reg
  Π.χ.:

```
logic x;
assign x = (a & b) | c;
```

- Για συνδυαστική λογική (combinatorial) υπάρχει το always_comb

```
always_comb begin  // no sensitivity list - auto inferred
  x = (a & b) | c;
end
```

- Για ακολουθιακή λογική (flip-flops) υπάρχει το always_ff

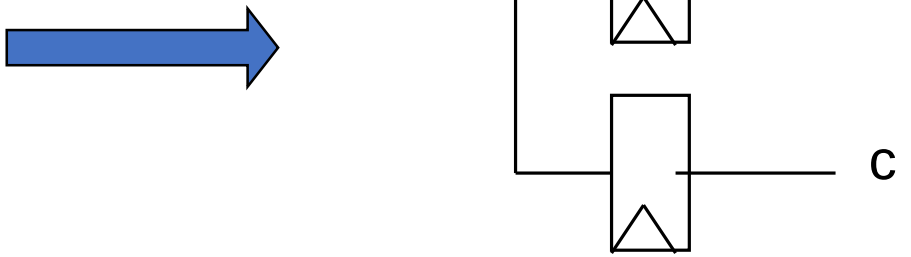
```
always_ff @(posedge clk) begin
  x <= a;
end
```

- Για μανταλωτές (latches) υπάρχει το always_latch

Αναθέσεις (assignments)

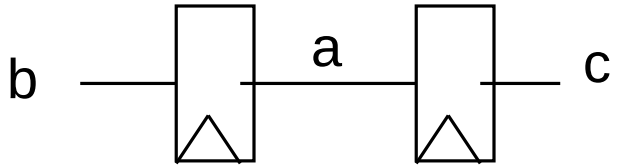
- **blocking =**

```
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
```



Assignments: Example

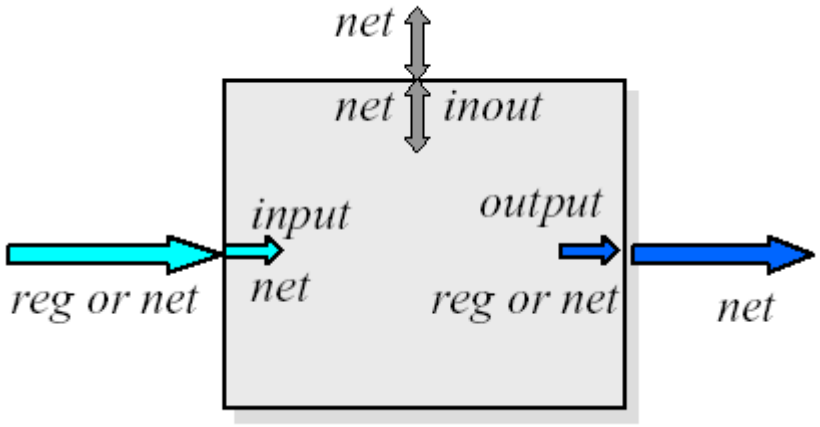
```
time  0 : a = #10 b;
time 10 : c = a;

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)
```

Κανόνες Πορτών Module

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

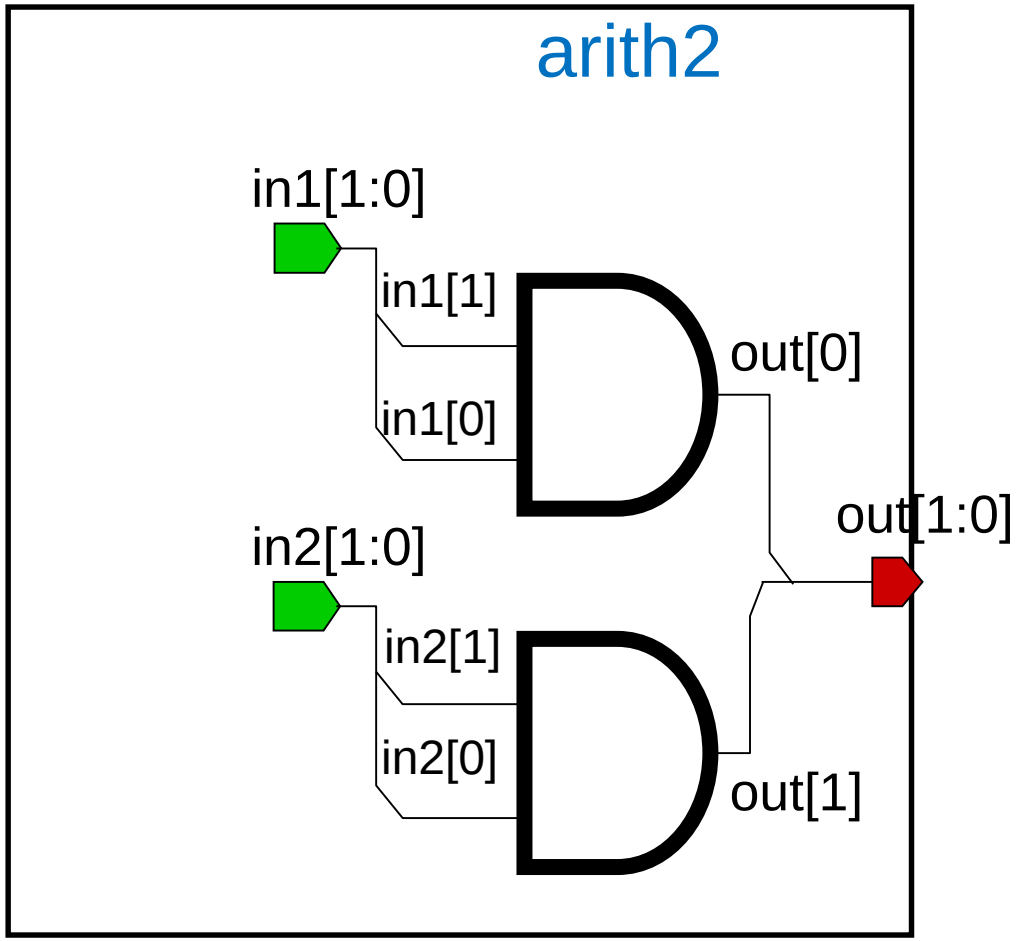


Συνδέσεις μεταξύ Instances

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

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

Multi-Bit Vectors/Busses (1/2)




```
module arith2 (
output [1:0] out,
input  [1:0] in1,
input  [1:0] in2);
…
…
endmodule
```

Multi-Bit Vectors/Busses (2/2)

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

```
module adder(
input  [7:0] a,
input  [7:0] 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
```

Conditional Statements – If… Else …

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

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

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

endmodule
```

Branch Statement – Case

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

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

Επίπεδα Αφαίρεσης Κώδικα

- Η λειτουργία ενός module μπορεί να οριστεί με διάφορους τρόπους
- **Behavioral** (επίπεδο πιο κοντά στην λογική)
  - Παρόμοια με την C – ο κώδικας δεν έχει άμεση σχέση με το hardware.

```
wire a = b + c
```

- **Gate level/structural** (επίπεδο κοντά στο hardware)
  - Ο κώδικας δείχνει πως πραγματικά υλοποιείται σε πύλες η λογική.

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

Συνθέσιμος Κώδικας

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

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

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

```
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
```

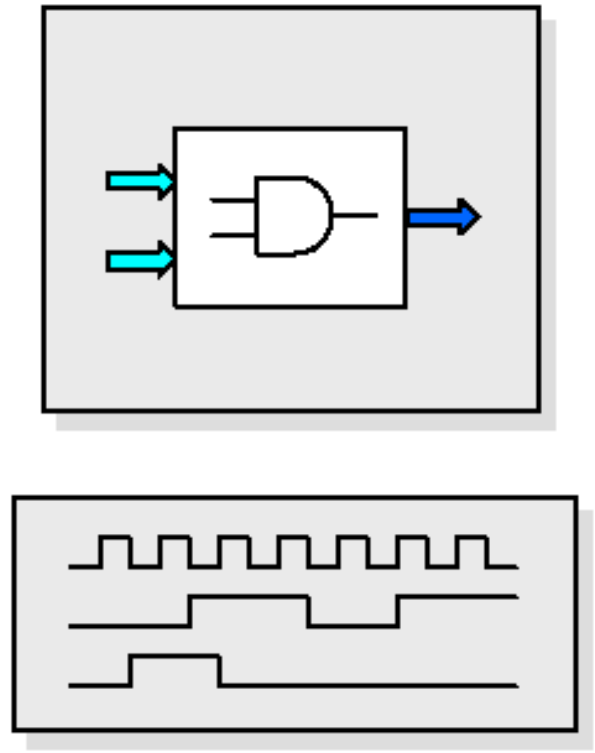
  - Συνήθως θεωρούμε ότι η συνδυαστική λογική δεν έχει καθυστέρηση π.χ.

```
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.

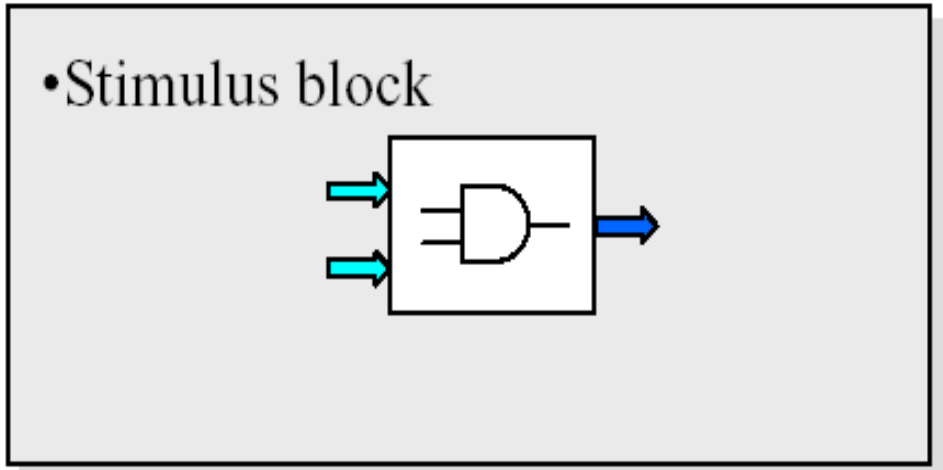
Testing

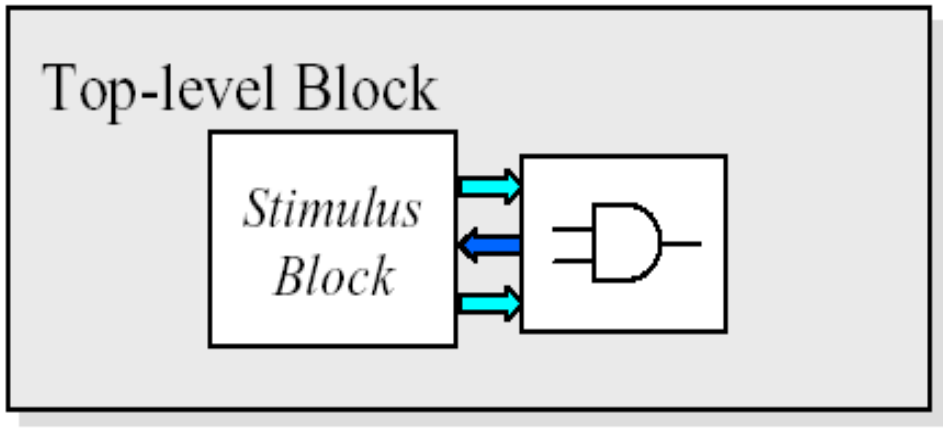
- Ιεραρχικός Έλεγχος
- Κάθε module ξεχωριστά
  - Block level simulation
  - Έλεγχος των προδιαγραφών, της λειτουργίας και των χρονισμών των σημάτων
- Όλο το design μαζί (System level simulation)
  - Έλεγχος της συνολικής λειτουργίας και των διεπαφών



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 δηλαδή αυτόματη σύγκριση των αναμενόμενων εξόδων.





Ένα απλό «test bench»

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

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

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

endmodule
```

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

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

endmodule
```

Μετρητής 8 bits (1/3)

```
module counter(
input  clk,
input reset,
output reg [7:0] out);

wire [7:0] next_value = out + 1;

always @(posedge clk) begin
 if (reset)
  out <= #2 8'b0;
 else
  out <= #2 next_value;
end

endmodule
```

```
module clk(
output reg out);

initial out = 1'b0;

always
 out = #10 ~out;

endmodule
```

Μετρητής 8 bits (2/3)

```
module test;

wire        clk;
reg         reset;
wire [7:0]  count;

clock   clk0(clk);

counter cnt0(clk, reset, count);
```

```
initial begin
 reset = 1;
 @(posedge clk);
 @(posedge clk);

 reset = #2 0;
 @(posedge clk);
 #300;
 $stop;
end

endmodule
```

Μετρητής 8 bits (3/3)

- counter.v
- clock.v
- test.v