

HY220

Εργαστήριο Ψηφιακών Κυκλωμάτων

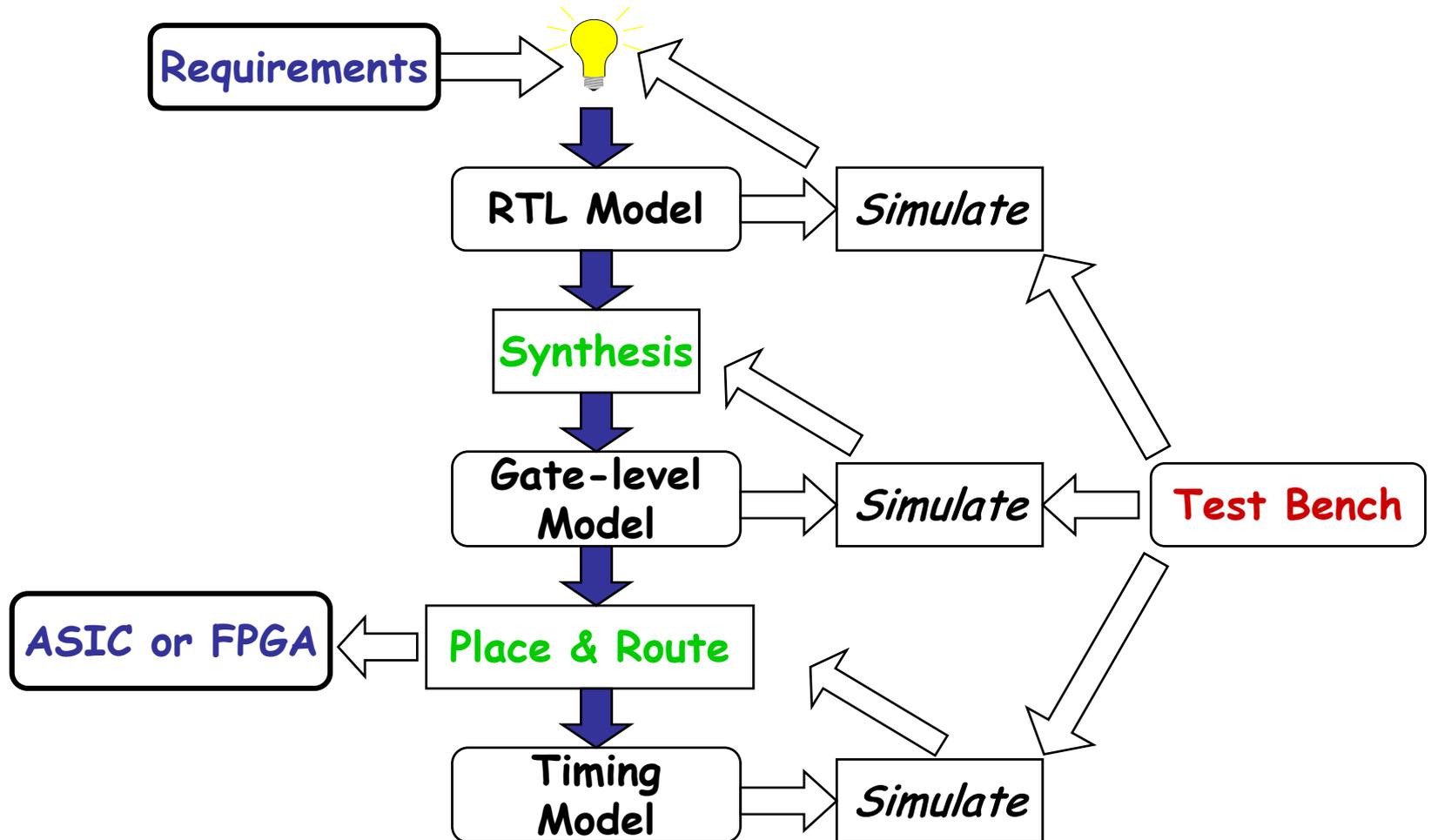
Χειμερινό Εξάμηνο
2014-2015

Verilog: Τα βασικά

Η εξέλιξη στη σχεδίαση ψηφιακών κυκλωμάτων

- Μεγάλη εξέλιξη τα τελευταία 30 χρόνια
 - Στις αρχές σχεδιάζαμε με λυχνίες (vacuum tubes) και transistors.
- Μετα ήρθαν τα ολοκληρωμένα κυκλώματα (Integrated Circuits - ICs)
 - SSI - λίγες πύλες (Small Scale Integration)
 - MSI - εκατοντάδες πύλες (Medium Scale Integration)
 - LSI - χιλιάδες πύλες (Large Scale Integration)
 - VLSI - εκατοντάδες χιλιάδες έως πολλά εκατομμύρια (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)



Τι είναι η Verilog;

- Verilog Hardware Description Language (HDL)
 - Μία υψηλού επιπέδου γλώσσα που μπορεί να αναπαραστεί και να προσομοιώνει ψηφιακά κυκλώματα.
 - 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 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

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

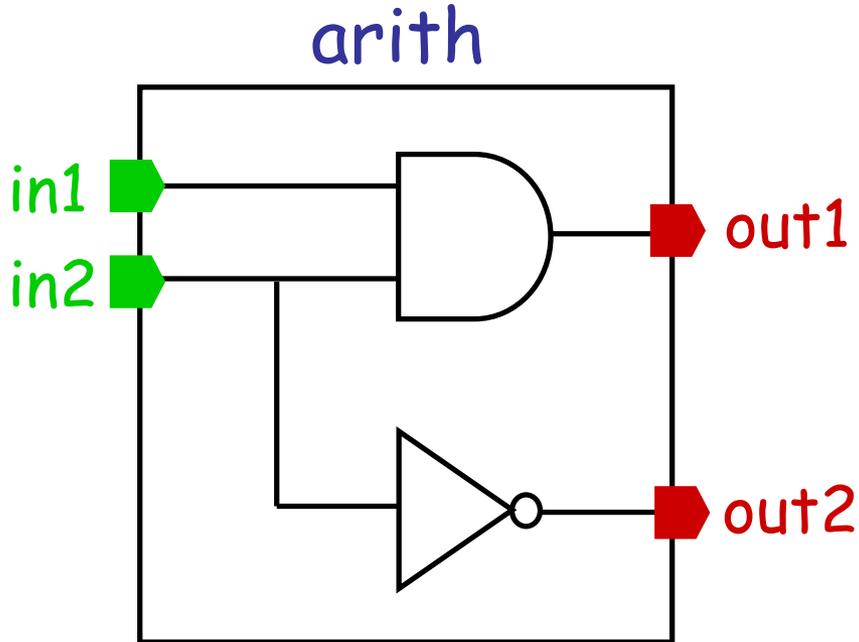
- Αναπαράσταση αριθμών
 - `<size>' <base_format> <number>`
`<size>` δείχνει τον αριθμό απο bits
`<base_format>` μπορεί να είναι : d, h, b, o (default: d)
 - Όταν το `<size>` λείπει το μέγεθος καθορίζεται από τον compiler
 - Όταν το `<number>` έχει πολλά ψηφία μπορούμε να το χωρίζουμε με `_` (underscore)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

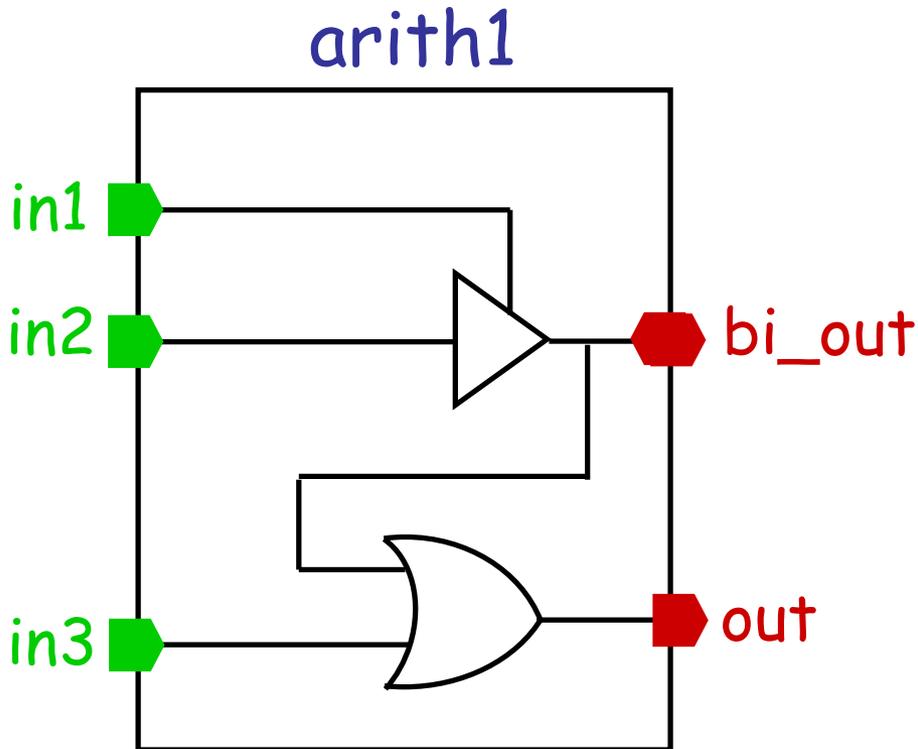
* Εφαρμόζεται μόνο σε έναν τελεστέο

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

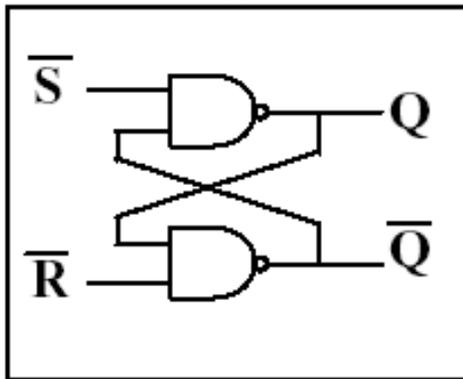


```
module arith1 (  
    inout  bi_out,  
    output out,  
    input  in1,  
    input  in2,  
    input  in3);  
...  
...  
endmodule
```

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

```
module test(input a,  
output 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(  
    input a,  
    input b,  
    output sum,  
    output cout);  
  
assign sum = a ^ b;  
assign cout = a & b;  
  
endmodule
```

Regs και ακολουθιακή λογική

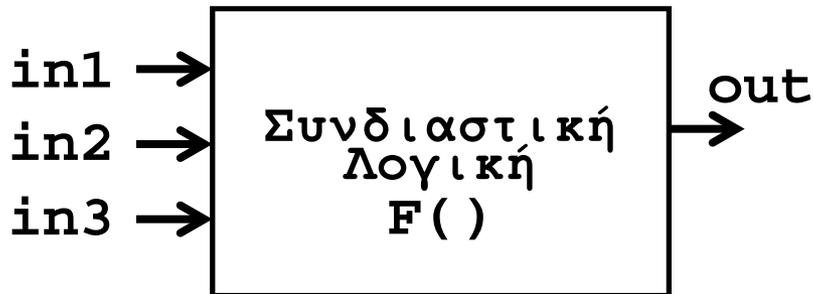
- Στοιχεία μνήμης
... κάτι ανάλογο με μεταβλητές στη C
- Μόνο regs (οχι wires) παίρνουν τιμή σε initial και always blocks.
 - Χρήση των begin και end για grouping πολλών προτάσεων
- Όπου χρησιμοποιούμε reg δεν σημαίνει οτι θα συμπεριφέρεται σαν καταχωρητής !!!

```
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

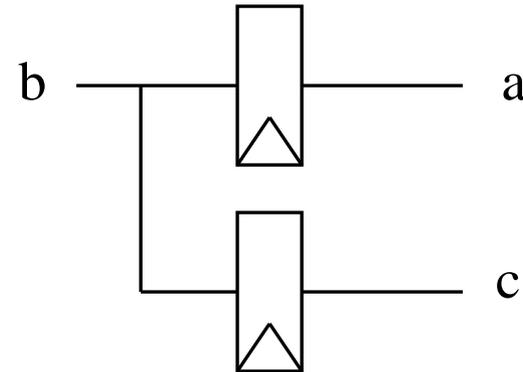
```
reg out;  
  
always @(in1 or in2 or in3)  
    out = in1 | (in2 & in3);
```

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

Αναθέσεις (Assignments)

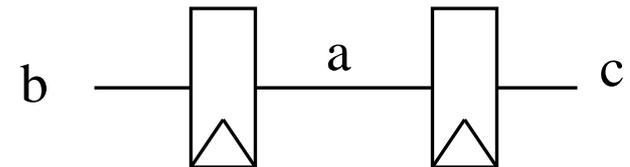
- **blocking =**

```
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)
```

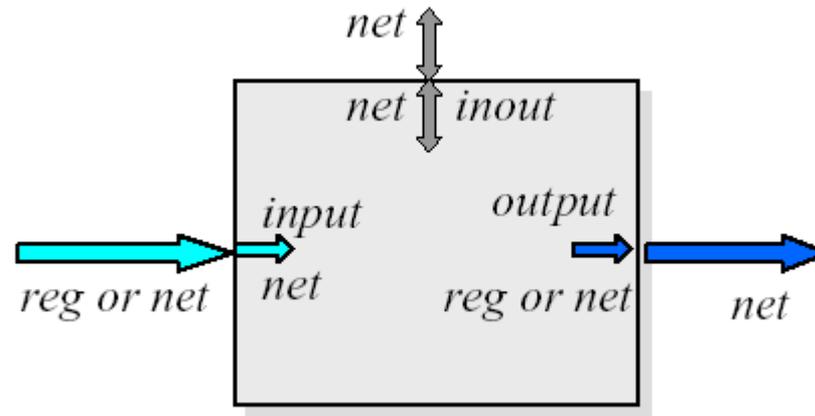
```
time 0 : #10
time 10 : a = b
time 10 : c = a

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

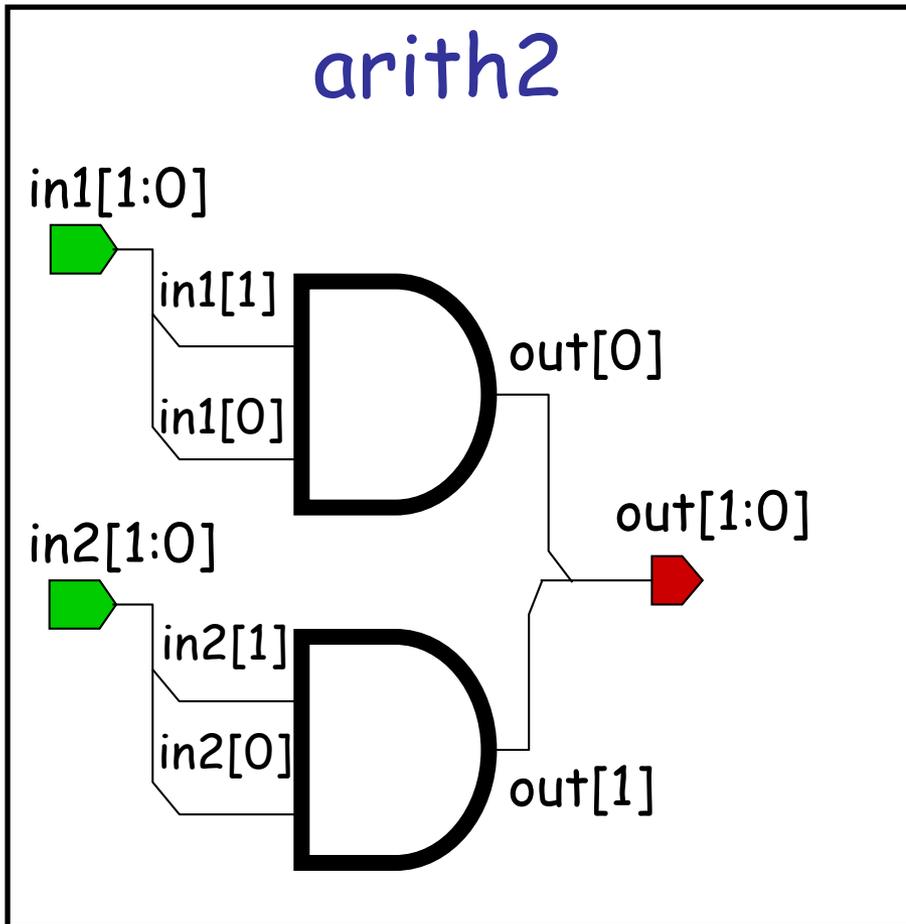


- Τα input και inout έχουν τύπο wire μέσα στο module
- Τα outputs μπορεί να έχουν τύπο wire ή reg

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`

Buses (1/2)



```
module arith2 (  
    input [1:0] in1,  
    input [1:0] in2,  
    output [1:0] out);  
...  
...  
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(  
  input  [7:0] a, b,  
  output [7:0] sum,  
  output          cout);  
  
  wire [8:0] tmp = a + b;  
  
  assign [7:0] sum  = tmp[7:0];  
  assign          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      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, 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
```

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.

π.χ.

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

- Gate level/structural (επίπεδο κοντά στο hardware)

Ο κώδικας δείχνει πως πραγματικά υλοποιείται σε πύλες η λογική.

π.χ.

```
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.

π.χ.

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

- Non-synthesizable κώδικας χρησιμοποιείται μόνο για προσομοίωση και πετιέται 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 συνδιαστική λογική δεν έχει καθυστέρηση.π.χ.

```
wire a = (b & c) | d;
```

```
// μόνο την λειτουργία όχι καθυστέρηση πυλών
```

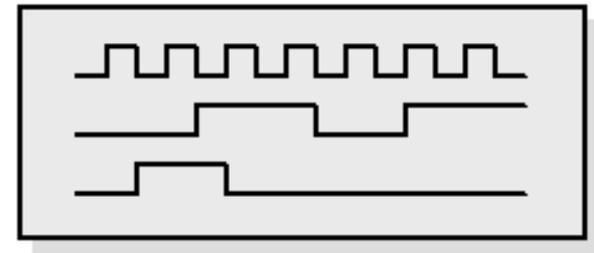
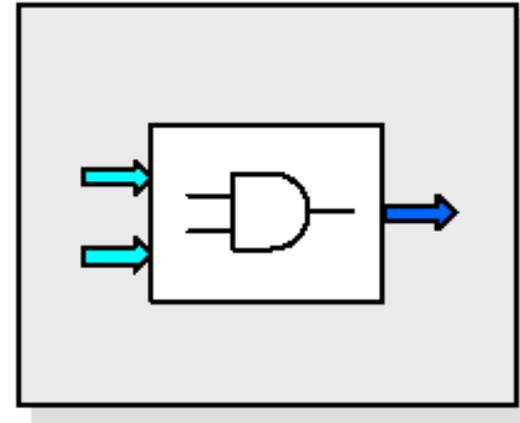
- Η καθυστέρηση χρησιμοποιείται κυρίως στο testbench κώδικα για να φτιάξουμε τα inputs.

- 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

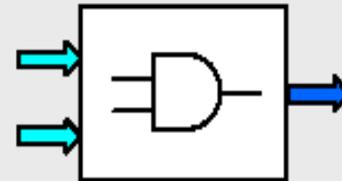
- Ιεραρχικός Έλεγχος
- Κάθε module ξεχωριστά
Block level simulation
 - Έλεγχος των προδιαγραφών, της λειτουργίας και των χρονισμών των σημάτων
- Όλο το design μαζί
System level simulation
 - Έλεγχος της συνολικής λειτουργίας και των διεπαφών



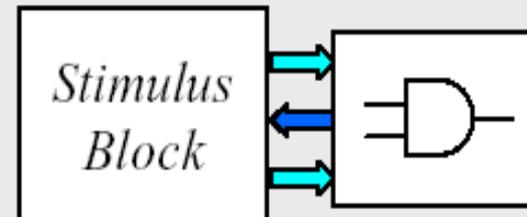
Έλεγχος σωστής λειτουργίας

- **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 δλδ. αυτόματη σύγκριση των αναμενόμενων εξόδων.

• Stimulus block



Top-level Block



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, b,
  output sum, cout);

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

endmodule
```

Μετρητής 8 bits (1/3)

```
module counter(  
input          clk, 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
```

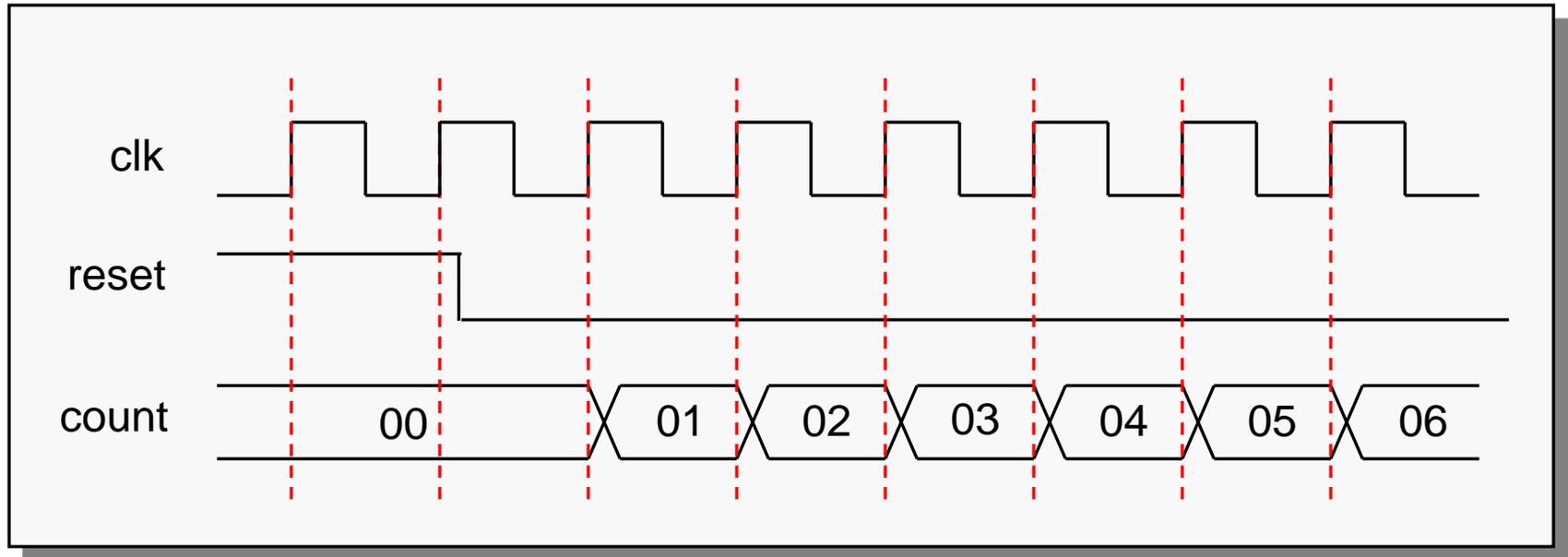
Μετρητής 8 bits (2/3)

```
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`

Verilog: Μια πιο κοντινή ματιά

Δομή της γλώσσας

- Μοιάζει αρκετά με τη C
 - Preprocessor
 - Keywords
 - Τελεστές

```
=  
==, !=  
<, >, <=, >=  
&& ||  
? :
```

```
& and  
| or  
~ not  
^ xor
```

- Γλώσσα «event driven»

```
`timescale 1ns / 1ns  
  
`define dh 2  
(e.g q <= #`dh d)  
  
`undef dh  
  
`ifdef dh / `ifndef dh  
    ...  
`else  
    ...  
`endif  
  
`include "def.h"
```

Events in Verilog (1/2)

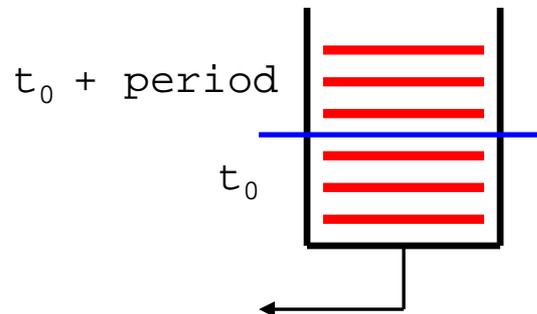
- Δουλεύει μόνο όταν κάτι αλλάξει
- Όλο το `simulation` δουλεύει γύρω από μια ουρά από γεγονότα (`event queue`)
 - Περιέχει `events` και ετικέτες με το χρόνο στον οποίο θα εκτελεστούν
 - Καμιά εγγύηση για τη σειρά εκτέλεσης γεγονότων που πρέπει να γίνουν στον ίδιο χρόνο!!!

```
always clk = #(`period / 2) ~clk;  
always @(posedge clk) a = b + 1;  
always @(posedge clk) b = c + 1;
```



Events in Verilog (2/2)

- Βασική ροή προσομοίωσης
 - Εκτέλεση των events για τον τρέχοντα χρόνο
 - Οι εκτέλεση events αλλάζει την κατάσταση του συστήματος και μπορεί να προκαλέσει προγραμματισμό events για το μέλλον
 - Όταν τελειώσουν τα events του τρέχοντα χρόνου προχωράμε στα αμέσως επόμενα χρονικά!



Τιμές σημάτων

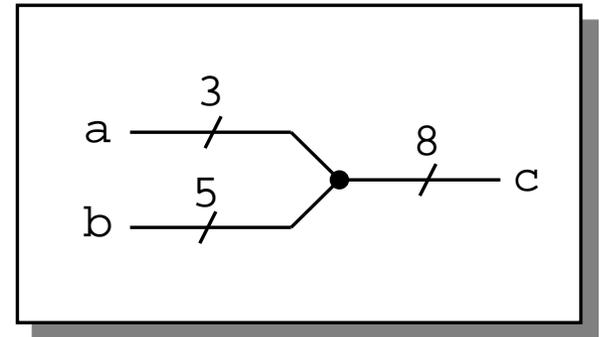
- **Four-valued logic**
- 0 ή 1
- Z
 - Έξοδος τρικατάστατου οδηγητή
 - Καλώδιο χωρίς ανάθεση
- X
 - Αρχική τιμή των regs
 - Έξοδος πύλης με είσοδο/ους Z
 - Ταυτόχρονη ανάθεση 0 και 1 από δύο ή περισσότερες πηγές (multi-source logic)
[πηγή = always block]
- Προσοχή στην αρχικοποίηση (regs)

```
initial ...
```

```
always @(posedge clk)  
  if (reset) ...  
  else ...
```

Concatenation

- «Hardwired» πράξεις...
- ... απαραίτητες σε μια HDL



```
wire [2:0] a;  
wire [4:0] b;  
  
wire [7:0] c = {a , b};
```

```
wire [7:0] unsigned;  
wire [15:0] sign_extend = {  
    (unsigned[7] ? 8'hFF : 8'h0),  
    unsigned  
};
```

For ... While ...

- ... τα γνωστά
- Μόνο μέσα σε blocks !
- Δεν υπάρχει break ούτε continue!!!
- Δεν υπάρχει i++, ++i κτλ!
- Κυρίως για testbenches !!!

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

```
integer j; //reg [3:0] j is OK!  
initial begin  
    j=0;  
    while(j < 10)begin  
        $display ("j= %b",j);  
        j=j+1;  
    end  
end
```

Παραμετρικά modules 1/2

```
module RegLd #(
    parameter N = 8,
    parameter dh = 2)
(
    input          Clk,
    input          i_load,
    input [N-1:0] i_D,
    output reg [N-1:0] o_Q);
//
always @(posedge clk)
    if (i_load)
        o_Q = #dh i_D;

endmodule
```

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

Παραμετρικά modules 2/2

```
RegLd    reg1(d2, q2, ld, clk);  
defparam reg1.N = 4;  
defparam reg1.dh = 4;  
  
RegLd #(  
    .N    ( 8 ),  
    .dh   ( 2 ))  
reg2(  
    .Clk    ( Clk,  
    .i_load ( reg_load ),  
    .i_D    ( wr_data ),  
    .o_Q    ( rd_data ));
```

Τρικατάστατοι οδηγητές

- Εκμετάλλευση της κατάστασης Z
- Χρήση του τύπου inout

```
module tristate(  
    input      en,  
    input      clk,  
    inout [7:0] data);
```

```
wire [7:0] data = (en) ? data_out : 8'bz;
```

```
always @(posedge clk)  
begin  
    if (!en)  
        case (data)  
            ...  
        endmodule
```

```
wire [7:0] bus;
```

```
tristate tr0(en0, clk, bus);  
tristate tr1(en1, clk, bus);  
tristate tr2(en2, clk, bus);
```

Μνήμες

- Αναδρομικά:
array of array
- Συνήθως non-synthesizable
- Ειδική αρχικοποίηση
 - \$readmemh
 - \$readmemb

```
wire [M-1:0]    word_in;
wire [M-1:0]    word_out;
wire [Nlog-1:0] addr;
reg  [M-1:0]    memory [((1<<Nlog)-1):0];

always @(posedge clk) begin
    if (we)
        memory[addr] = word_in;
    else word_out = memory[addr];
end
```

```
always @(negedge rst_n)
    $readmemh("memory.dat", memory);
```

```
memory.dat:
0F00 00F1
0F02
```

Συναρτήσεις - Functions (1/3)

- Δήλωση (declaration):

```
function [ range_or_type ] fname;  
    input_declarations  
    statements  
endfunction
```

- Επιστρεφόμενη τιμή (return value):

- Ανάθεση στο σώμα του function

```
fname = expression;
```

- Κλήση (function call):

```
fname ( expression,... )
```

Συναρτήσεις - Functions (2/3)

- Χαρακτηριστικά συναρτήσεων:
 - Επιστρέφει 1 τιμή (default: 1 bit)
 - Μπορεί να έχει πολλαπλά ορίσματα εισόδου (πρέπει να έχει τουλάχιστον ένα)
 - Μπορούν να καλούν άλλες functions αλλά όχι tasks.
 - Δεν υποστηρίζουν αναδρομή (non-recursive)
 - Εκτελούνται σε μηδέν χρόνο προσομοίωσης
 - **Δεν** επιτρέπονται χρονικές λειτουργίες (π.χ. delays, events)
- Χρησιμοποιούνται για συνδυαστική λογική και συνθέτονται συνήθως έτσι.
 - προσοχή στον κώδικα για να γίνει σωστά σύνθεση

Συναρτήσεις - Functions (3/3)

- Function examples:

```
function calc_parity;  
input [31:0] val;  
begin  
    calc_parity = ^val;  
end  
endfunction
```

```
function [15:0] average;  
input [15:0] a, b, c, d;  
begin  
    average = (a + b + c + d) >> 2;  
end  
endfunction;
```

Verilog Tasks (1/2)

- Τυπικές *procedures*
- Πολλαπλά ορίσματα `input`, `output` και `inout`
- Δεν υπάρχει συγκεκριμένη τιμή επιστροφής (χρησιμοποιεί τα όρισματα `output`)
- Δεν υποστηρίζουν αναδρομή (*non-recursive*)
- Μπορούν να καλούν άλλες *tasks* και *functions*
- Μπορούν να περιέχουν *delays*, *events* και χρονικές λειτουργίες
 - Προσοχή στη σύνθεση

Verilog Tasks (2/2)

- Task example:

```
task ReverseByte;  
    input [7:0] a;  
    output [7:0] ra;  
    integer j;  
begin  
    for (j = 7; j >=0; j=j-1) begin  
        ra[j] = a[7-j];  
    end  
end  
endtask
```

Functions and Tasks

- Ορίζονται μέσα σε `modules` και είναι τοπικές
- Δεν μπορούν να έχουν `always` και `initial blocks` αλλά μπορούν να καλούνται μέσα από αυτά
 - Μπορούν να έχουν ότι εκφράσεις μπαίνουν σε `blocks`

Functions vs Tasks

Functions	Tasks
Μπορούν να καλούν άλλες functions αλλά όχι tasks	Μπορούν να καλούν άλλες tasks και functions
Εκτελούνται σε μηδενικό χρόνο προσομοίωσης	Μπορούν να διαρκούν μη μηδενικό χρόνο προσομοίωσης
Δεν μπορούν περιέχουν χρονικές λειτουργίες (delay, events κτλ)	Μπορούν να περιέχουν χρονικές λειτουργίες (delay, events κτλ)
Έχουν τουλάχιστον 1 είσοδο και μπορούν να έχουν πολλές	Μπορούν να έχουν μηδέν ή περισσότερα ορίσματα εισόδων, εξόδων και inout
Επιστρέφουν μια τιμή, δεν έχουν εξόδους	Δεν επιστρέφουν τιμή αλλά βγάζουν έξοδο από τα ορίσματα εξόδου και inout

System Tasks and Functions

- Tasks and functions για έλεγχο της προσομοίωσης
 - Ξεκινούν με "\$" (e.g., \$monitor)
 - Standard - της γλώσσας
- Παράδειγμα system task: \$display

```
$display("format-string", expr1, ..., exprn);
```

format-string - regular ASCII mixed with formatting characters %d - decimal, %b - binary, %h - hex, %t - time, etc.
other arguments: any expression, including wires and regs

```
$display("Error at time %t: value is %h, expected %h", $time, actual_value, expected_value);
```

Χρήσιμες System Tasks

- `$time` - τρέχον χρόνος προσομοίωσης
- `$monitor` - τυπώνει όταν αλλάζει τιμή ένα όρισμα (1 μόνο κάθε φορά νέες κλήσεις ακυρώνουν τις προηγούμενες)

```
$monitor("cs=%b, ns=%b", cs, ns)
```

- Έλεγχος προσομοίωσης
 - `$stop` - διακοπή simulation
 - `$finish` - τερματισμός simulation
- Υπάρχουν και συναρτήσεις για file I/O (`$fopen`, `$fclose`, `$fwrite` ... etc)

Verilog: Στυλ Κώδικα και Synthesizable Verilog

Τα στυλ του κώδικα

- Τρεις βασικές κατηγορίες
 - Συμπεριφοράς - Behavioral
 - Μεταφοράς Καταχωρητών - Register Transfer Level (RTL)
 - Δομικός - Structural
- Και εμάς τι μας νοιάζει;
 - Διαφορετικός κώδικας για διαφορετικούς σκοπούς
 - Synthesizable ή όχι;

Behavioral (1/3)

- Ενδιαφερόμαστε για την συμπεριφορά των blocks
- Αρχικό simulation
 - Επιβεβαίωση αρχιτεκτονικής
- Test benches
 - Απο απλά ...
 - ... μέχρι εκλεπτυσμένα

```
initial begin
    // reset everything
end

always @(posedge clk) begin
    case (opcode)
        8'hAB: RegFile[dst] = #2 in;
        8'hEF: dst = #2 in0 + in1;
        8'h02: Memory[addr] = #2 data;
    endcase

    if (branch)
        dst = #2 br_addr;
end
```

Behavioral (2/3)

- Περισσότερες εκφράσεις
 - for / while
 - functions
 - tasks
- Περισσότεροι τύποι
 - integer
 - real
 - πίνακες



```
integer sum, i;
integer opcodes [31:0];
real average;

initial
  for (i=0; i<32; i=i+1)
    opcodes[i] = 0;

always @(posedge clk) begin
  sum = sum + 1;
  average = average + (c / sum);
  opcodes[d] = sum;
  $display("sum: %d, avg: %f",
    sum, average);
end
```

Behavioral (3/3)

```
module test;  
  
    task ShowValues;  
        input [7:0] data;  
        $display(..., data);  
    endtask  
  
    ...  
    always @(posedge clk)  
        ShowValues(counter);  
  
    ...  
endmodule
```

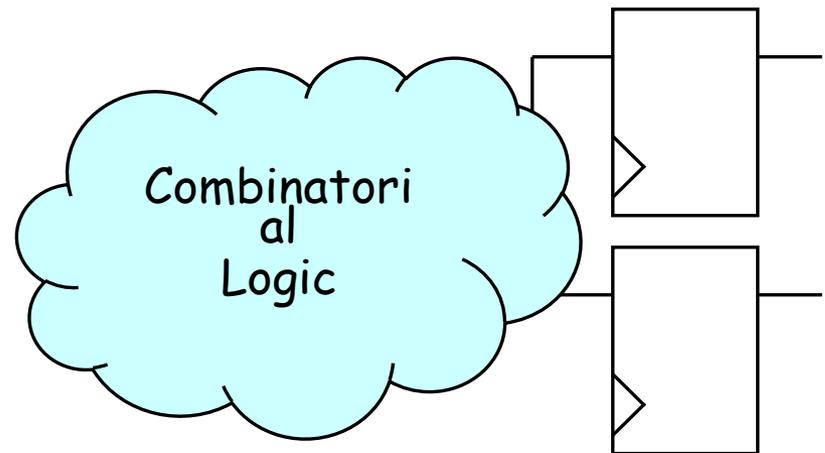
```
`define period 20  
  
initial begin  
    reset_ = 1'b0;  
    reset_ = #(2*`period + 5) 1'b1;  
  
    @(branch);  
    reset_ = 1'b0;  
    reset_ = #(2*`period + 5) 1'b1;  
end
```

```
always @(negedge reset_) begin  
    fork  
        a = #2 8'h44;  
        b = #(4*`period + 2) 1'b0;  
        c = #(16*`period + 2) 8'h44;  
    join  
end
```

Register Transfer Level - RTL

- Το πιο διαδεδομένο και υποστηριζόμενο μοντελο για **synthesizable** κώδικα
- Κάθε block κώδικα αφορά την είσοδο λίγων καταχωρητών
- Σχεδιάζουμε **κύκλο-κύκλο** με «οδηγό» το ρολόι
- Εντολές:
 - Λιγότερες
 - ... όχι τόσο περιοριστικές

Think Hardware!



Structural

- Αυστηρότατο μοντέλο
 - Μόνο module instantiations
- Συνήθως για το top-level module
- Καλύτερη η αυστηρή χρήση ΤΟΥ

```
module top;
wire clk, reset;
wire [31:0] d_data, I_data;
wire [9:0] d_adr;
wire [5:0] i_adr;

clock clk0(clk);

processor pr0(clk, reset,
             d_adr, d_data,
             i_adr, i_data,
             ...);

memory #10 mem0(d_adr,
               d_data);

memory #6 mem1(i_adr,
               i_data);

tester tst0(reset, ...);

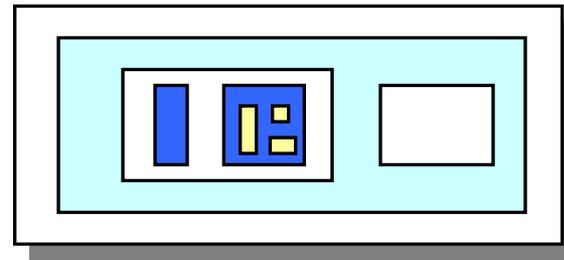
endmodule
```

... και μερικές συμβουλές

- **Ονοματολογία**
 - Όχι πολύ μεγάλα / μικρά ονόματα
 - ... με νόημα
- **Συνδυαστική λογική**
 - Όχι όλα σε μια γραμμή...
 - Ο compiler ξέρει καλύτερα
 - Αναγνωσιμότητα
- **Δομή**
 - Πολλές οντότητες
 - Ε όχι και τόσες!

```
wire a,  
    controller_data_now_ready;  
wire drc_rx_2,  
    twra_malista;
```

```
if (~req &&  
    ((flag & prv_ack) |  
     ~set) &&  
    (count-2 == 0))  
    ...
```



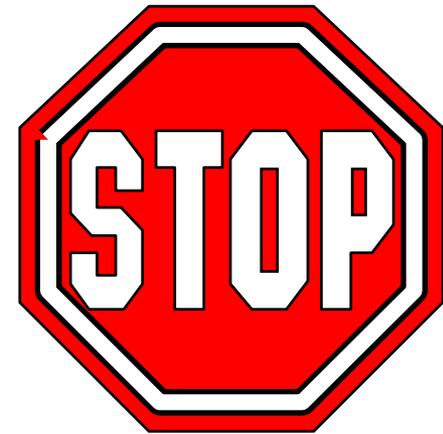
... περισσότερες συμβουλές

- Διευκολύνουν την ανάγνωση και την χρήση του κώδικα (filters, tools etc)
 - Είσοδοι ξεκινούν με `i_*`
 - Οι έξοδοι με `o_*`
 - Οι τρικατάστατες με `io_*`
 - Εκτός από ρολόι και `reset`
 - Τα **active low** σήματα τελειώνουν με `*_n`
- Συνδέσεις πορτών συσχετίζοντας ονόματα

```
module adder(o_Sum, i_In1, i_In2);
adder i0_adder ( // instance names i0_adder, i1_adder ...
    .i_In2(B),
    .i_In1(A),
    .o_Sum(C)
) // o_Sum = C, i_In1 = A, i_In2 = B
```

Σχόλια

- Ακούγεται μονότονο, αλλά...
 - Κώδικας hardware πιο δύσκολος στην κατανόηση
 - Ακόμα και ο σχεδιαστής ξεχνάει γρήγορα
 - Αν δε μπουν στην αρχή, δε μπαίνουν ποτέ
- Σημεία κλειδιά
 - Σε κάθε module
 - Σε κάθε block



```
/******  
 * Comments on module test:  
 * Module test comprises of  
 * the following components...  
 *****/  
module test;  
 // Line comment
```

Verilog and Synthesis

- Χρήσεις της Verilog
 - Μοντελοποίηση και event-driven προσομοίωση
 - Προδιαγραφές κυκλώματος για σύνθεση (logic synthesis)
- Logic Synthesis
 - Μετατροπή ενός υποσυνόλου της Verilog σε netlist
 - Register Inference, combinatorial logic
 - Βελτιστοποίηση του netlist (area,speed)

Register - D Flip Flop

```
module Reg #(
    parameter N = 16,
    parameter dh = 1)
(
    input          Clk,
    input          [N-1:0] i_D,
    output reg [N-1:0] o_Q);
//
    always @(posedge Clk)
        o_Q <= #dh i_D;
//
endmodule
```

Register with Asynchronous Reset

```
module RegRst #(
    parameter N = 16,
    parameter dh = 1)
(
    input          Clk,
    input          Reset_n,
    input          [N-1:0] i_D,
    output reg [N-1:0] o_Q)
//
    always @(posedge Clk or negedge Reset_n) begin
        if (~Reset_n)
            o_Q <= #dh 0;
        else
            o_Q <= #dh i_D;
    end
endmodule
```

Register with Synchronous Reset

```
module RegRst #(
    parameter N = 16,
    parameter dh = 1)
(
    input                Clk,
    input                Reset_n,
    input                [N-1:0] i_D,
    output reg [N-1:0] o_Q)
//
    always @(posedge Clk) begin
        if (~Reset_n)
            o_Q <= #dh 0;
        else
            o_Q <= #dh i_D;
    end
endmodule
```

Register with Load Enable

```
module RegLd #(
    parameter N = 16,
    parameter dh = 1)
(
    input          Clk,
    input          i_Ld,
    input [N-1:0] i_D,
    output reg [N-1:0] o_Q);
//
    always @(posedge Clk)
        if (i_Ld)
            o_Q <= #dh i_D;
//
endmodule
```

Set Clear flip-flop with Strong Clear

```
module scff_sc #(
    parameter dh = 1)
(
    input Clk
    input i_Set,
    input i_Clear,
    output o_Out);
//
    always @(posedge Clk)
        o_Out <= #dh (o_Out | i_Set) & ~i_Clear;
//
endmodule
```

Set Clear flip-flop with Strong Set

```
module scff_s #(
    parameter dh = 1)
(
    input Clk
    input i_Set,
    input i_Clear,
    output o_Out);
//
    always @(posedge Clk)
        o_Out <= #dh i_Set | (o_Out & ~i_Clear);
//
endmodule
```

T Flip Flop

```
module Tff #(
    parameter dh = 1)
(
    input Clk,
    input Rst,
    input i_Toggle,
    output o_Out);
//
always @(posedge Clk)
    if(Rst)
        o_Out <= #dh 0
    else if (i_Toggle)
        o_Out <= #dh ~o_Out;
//
endmodule
```

Multiplexor 2 to 1

```
module mux2 #(
    parameter N = 16)
(
    output [N-1:0] o_Out,
    input [N-1:0] i_In0,
    input [N-1:0] i_In1,
    input          i_Sel);
//
wire [N-1:0] o_Out = i_Sel ? i_In1 : i_In0;
//
endmodule
```

Multiplexor 4 to 1

```
module mux4 #(
    parameter N = 32)
(
    input      [N-1:0] In0,
    input      [N-1:0] In1,
    input      [N-1:0] In2,
    input      [N-1:0] In3,
    input      [ 1:0] Sel,
    output reg [N-1:0] Out);
//
    always @(i_In0 or i_In1 or i_In2 or i_In3 or i_Sel) begin
        case ( i_Sel )
            2'b00 : o_Out <= i_In0;
            2'b01 : o_Out <= i_In1;
            2'b10 : o_Out <= i_In2;
            2'b11 : o_Out <= i_In3;
        endcase
    end
endmodule
```

Multiplexor 4 to 1

```
module mux4 #(
    parameter N = 32)
(
    input [N-1:0] i_In0,
    input [N-1:0] i_In1,
    input [N-1:0] i_In2,
    input [N-1:0] i_In3,
    input          i_Sel0,
    input          i_Sel1,
    input          i_Sel2,
    input          i_Sel3,
    output [N-1:0] o_Out);
//
    assign o_out = i_In0 & {N{i_Sel0}} |
                  i_In1 & {N{i_Sel1}} |
                  i_In2 & {N{i_Sel2}} |
                  i_In3 & {N{i_Sel3}};
//
endmodule
```

Positive Edge Detector

```
module PosEdgDet #(
    parameter dh = 1)
(
    input  Clk,
    input  i_In,
    output o_Out);
//
    reg Tmp;
    always @(posedge Clk)
        Tmp <= #dh i_In;
//
    assign o_Out = ~Tmp & i_In;
//
endmodule
```

Negative Edge Detector

```
module NegEdgDet #(
    parameter dh = 1)
(
    input  Clk,
    input  i_In,
    output o_Out);
//
    reg Tmp;
    always @(posedge Clk)
        Tmp <= #dh i_In;
//
    assign o_Out = Tmp &~i_In;
//
endmodule
```

Edge Detector

```
module EdgDet #(
    parameter dh = 1)
(
    input  Clk,
    input  i_In,
    output o_Out);
//
    reg Tmp;
    always @(posedge Clk)
        Tmp <= #dh i_In;
//
    wire Out = Tmp ^ i_In;
//
endmodule
```

Tristate Driver

```
module Tris #(
    parameter N = 32)
(
input  [N-1:0] i_TrIsIn,
input          i_TrIsOen_n,
output [N-1:0] o_TrIsOut);
//
    assign o_TrIsOut = ~i_TrIsOen_n ? i_TrIsIn : `bz;
//
endmodule
```

Up Counter

```
module Cnt #(
    parameter N      = 32,
    parameter MaxCnt = 100,
    parameter dh     = 1)
(
    input          Clk,
    input          i_En,
    input          i_Clear,
    output reg     o_Zero,
    output reg [N-1:0] o_Out);
//
always @(posedge Clk) begin
    if(i_Clear) begin
        o_Out  <= #dh 0;
        o_Zero <= #dh 0;
    end
    else if (i_En) begin
        if (o_Out==MaxCnt) begin
            o_Out  <= #dh 0;
            o_Zero <= #dh 1;
        end
        else begin
            o_Out  <= #dh o_Out + 1'b1;
            o_Zero <= #dh 0;
        end
    end
end
end
endmodule
```

Parallel to Serial Shift Register

```
module P2Sreg #(
    parameter N = 32,
    parameter dh = 1)
(
    input          Clk,
    input          Reset_n,
    input          i_Ld,
    input          i_Shift,
    input  [N-1:0] i_In,
    output         o_Out);
//
    reg [N-1:0] TmpVal;
//
    always @(posedge Clk or negedge Reset_n) begin
        if (~Reset_n) TmpVal <= #dh 0;
        else begin
            if (i_Ld) TmpVal <= #dh i_In;
            else if(i_Shift) TmpVal <= #dh TmpVal>>1;
        end
    end
end
//
    assign o_Out = TmpVal[0];
//
endmodule
```

Serial to Parallel Shift Register

```
module S2Preg #(
    parameter N = 32,
    parameter dh = 1)
(
    input          Clk,
    input          i_Clear,
    input          i_Shift,
    input          i_In,
    output reg [N-1:0] o_Out);
//
    always @(posedge Clk) begin
        if (i_Clear)
            o_Out <= #dh 0;
        else if (i_Shift)
            o_Out <= #dh {o_Out[N-2:0],i_In};
    end
//
endmodule
```

Barrel Shift Register

```
module BarShiftReg(
    parameter N = 32,
    parameter dh = 1)
(
    input          Clk,
    input          Reset_n,
    input          i_Ld,
    input          i_Shift,
    input [N-1:0] i_In,
    output reg [N-1:0] o_Out);
//
always @(posedge Clk) begin
    if (~Reset_n) o_Out <= #dh 0;
    else begin
        if (i_Ld)
            o_Out <= #dh i_In;
        else if (i_Shift)
            o_Out <= #dh {o_Out[N-2:0],o_Out[N-1]};
    end
end
//
endmodule
```

3 to 8 Binary Decoder

```
module dec #(
    parameter Nlog = 3)
(
    input      [      Nlog-1:0] i_in,
    output reg [((1<<Nlog))-1:0] o_out);
//
Integer i;
//
always @(i_in) begin
    for (i=0; i<(1<<Nlog); i=i+1) begin
        if (i_In==i)
            o_out[i]=1;
        else o_out[i] = 0;
    end
end
//
endmodule
```

8 to 3 Binary Encoder

```
module enc #(
    parameter Nlog = 3)
    (
        input      [((1<<Nlog)-1):0] i_In,
        output reg [      Nlog-1:0] o_Out);
//
integer i;
//
always @(i_In) begin
    o_Out = x;
    for (i=0; i<(1<<Nlog); i=i+1) begin
        if (i_In[i]) o_Out=i;
    end
end
//
endmodule
```

Priority Enforcer Module

```
module PriorEnf #(
    parameter N = 8)
(
    input      [N-1:0] In,
    output reg [N-1:0] Out,
    output reg          OneDetected);
//
integer i;
reg      DetectNot;
always @(i_In) begin
    DetectNot=1;
    for (i=0; i<N; i=i+1)
        if (i_In[i] & DetectNot) begin
            o_Out[i]=1;
            DetectNot=0;
        end
    else o_Out[i]=0;
    OneDetected = !DetectNot;
end
endmodule
```

Latch

```
module Latch #(
    parameter N = 16,
    parameter dh = 1)
(
    input      [N-1:0] i_In,
    input      i_Ld,
    output reg [N-1:0] o_Out);
//
    always @(i_In or i_Ld)
        if (i_Ld) o_Out = #dh i_In;
//
endmodule;
```

Combinatorial Logic and Latches (1/3)

```
module mux3 #(
    parameter N = 32 )
(
    input [ 1:0 ] Sel,
    input [N-1:0] In2,
    input [N-1:0] In1,
    input [N-1:0] In0,
    output reg [N-1:0] Out);
//
    always @(In0 or In1 or In2 or Sel) begin
        case ( Sel )
            2'b00 : Out <= In0;
            2'b01 : Out <= In1;
            2'b10 : Out <= In2;
        endcase
    end
endmodule
```

Γιατί είναι λάθος;



Combinatorial Logic and Latches (2/3)

```
module mux3 #(
    parameter N = 32 )
(
    input [ 1:0 ] Sel,
    input [N-1:0] In2,
    input [N-1:0] In1,
    input [N-1:0] In0,
    output reg [N-1:0] Out);
    always @(In0 or In1 or In2 or Sel) begin
        case ( Sel )
            2'b00 : Out <= In0;
            2'b01 : Out <= In1;
            2'b10 : Out <= In2;
            default : Out <= x;
        endcase
    end
endmodule
```

Το σωστό !!!



Combinatorial Logic and Latches (3/3)

- Όταν φτιάχνουμε συνδυαστική λογική με always blocks και regs τότε πρέπει να αναθέτουμε τιμές στις εξόδους της λογικής για όλες τις πιθανές περιπτώσεις εισόδων (κλήσεις του always) !!!
 - Για κάθε if ένα else
 - Για κάθε case ένα default
- Παραλείψεις δημιουργούν latches κατά τη σύνθεση
 - Οι περιπτώσεις που δεν καλύπτουμε χρησιμοποιούνται για το «σβήσιμο» του load enable του latch. (θυμάται την παλιά τιμή)

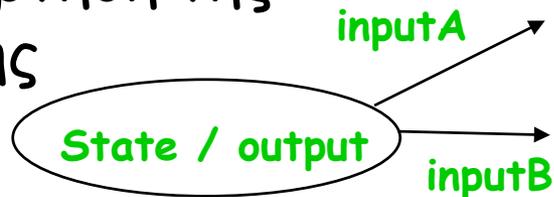
Μηχανές Πεπερασμένων Καταστάσεων

FSMs

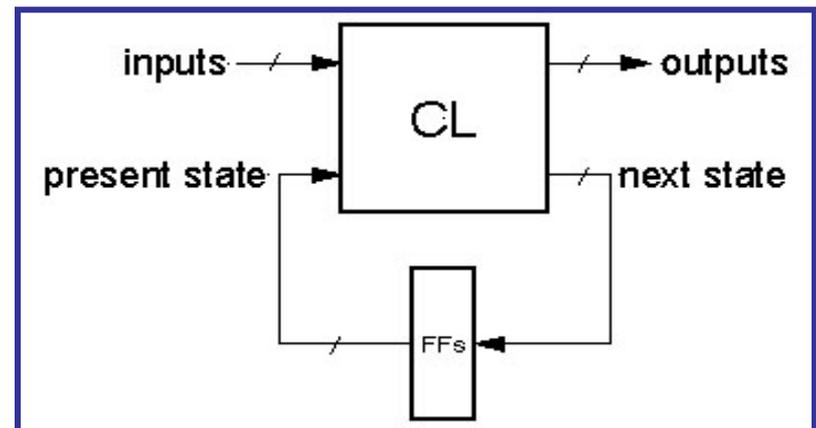
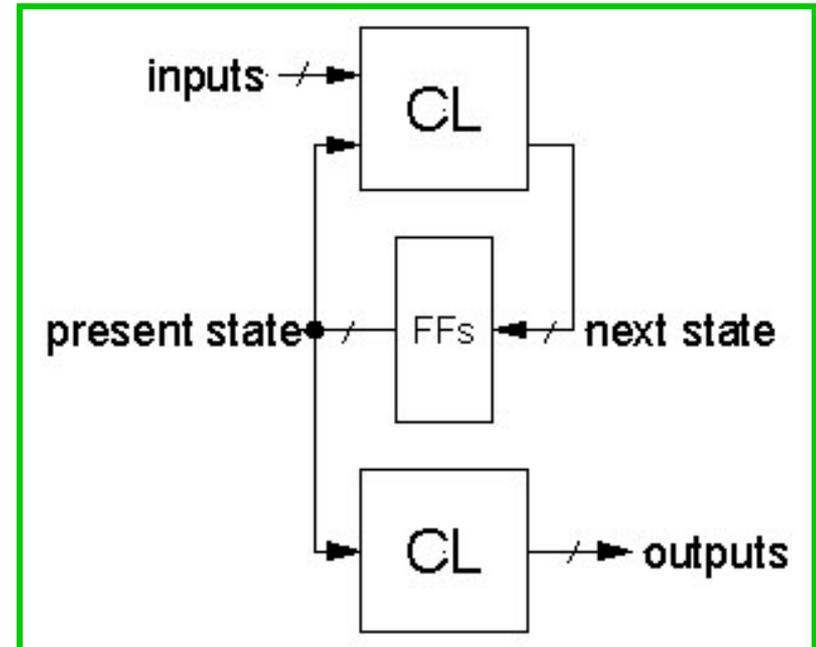
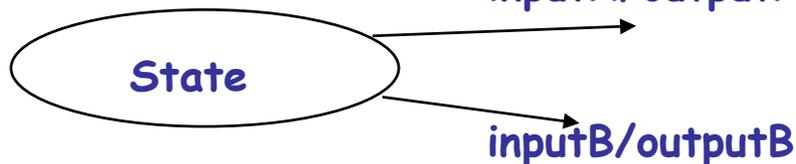
- Οι μηχανές πεπερασμένων καταστάσεων
Finite State Machines (FSMs)
 - πιο αφηρημένος τρόπος να εξετάζουμε ακολουθιακά κυκλώματα
 - Είσοδοι, έξοδοι, τρέχουσα κατάσταση, επόμενη κατάσταση
 - Σε κάθε ακμή του ρολογιού συνδυαστική λογική παράγει τις εξόδους και την επόμενη κατάσταση σαν συνάρτησης των εισόδων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

Χαρακτηριστικά των FSM

- Η επόμενη κατάσταση είναι συνάρτηση της τρέχουσας κατάστασης και των εισόδων
- **Moore Machine:** Οι έξοδοι είναι συνάρτηση της κατάστασης



- **Mealy Machine:** Οι έξοδοι είναι συνάρτηση της κατάστασης και των εισόδων

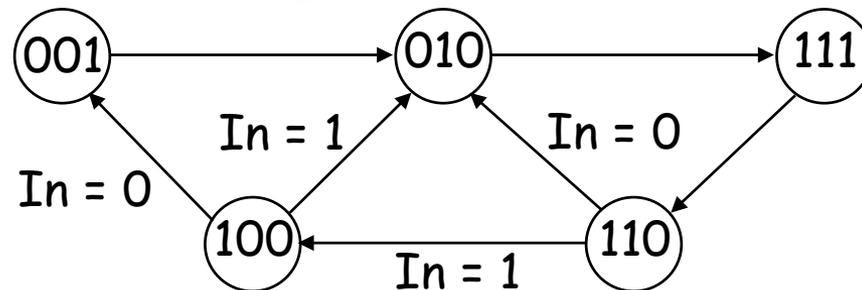


Βήματα Σχεδίασης

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

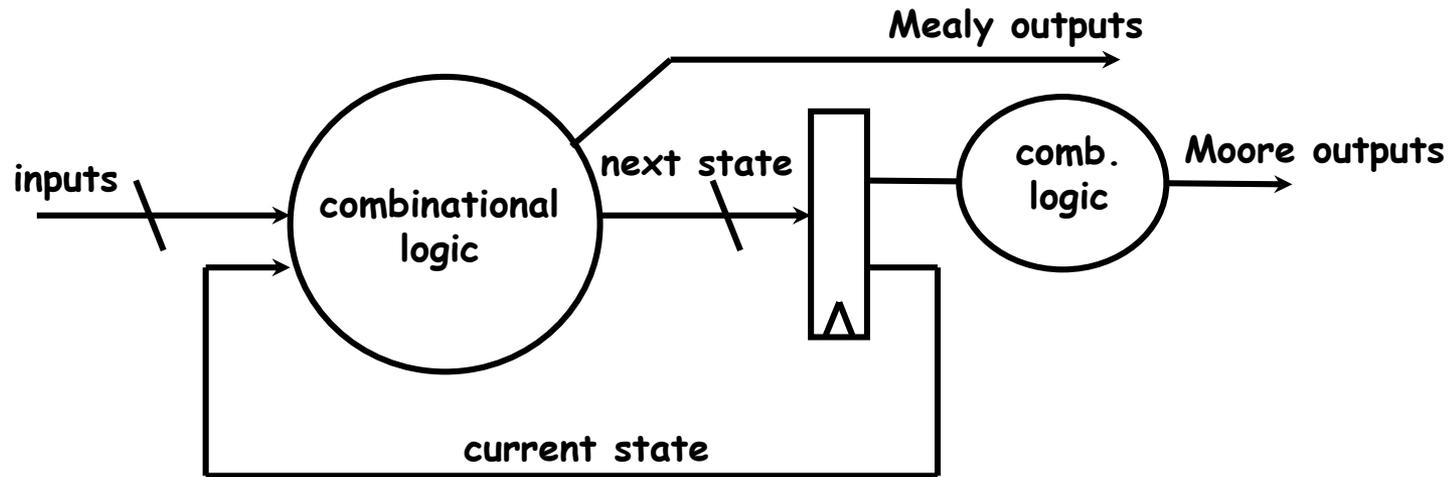
Αναπαράσταση FSM

- Καταστάσεις: όλες οι πιθανές τιμές στα ακολουθιακά στοιχεία μνήμης (FFs)
- Μεταβάσεις: αλλαγή κατάστασης
- Αλλαγή τις κατάστασης με το ρολόι αφού ελέγχει την φόρτωση τιμής στα στοιχεία μνήμης (FFs)



- Ακολουθιακή λογική
 - Ακολουθία μέσω μιας σειράς καταστάσεων
 - Βασίζεται στην ακολουθία των τιμών στις εισόδους

Υλοποίηση FSMs



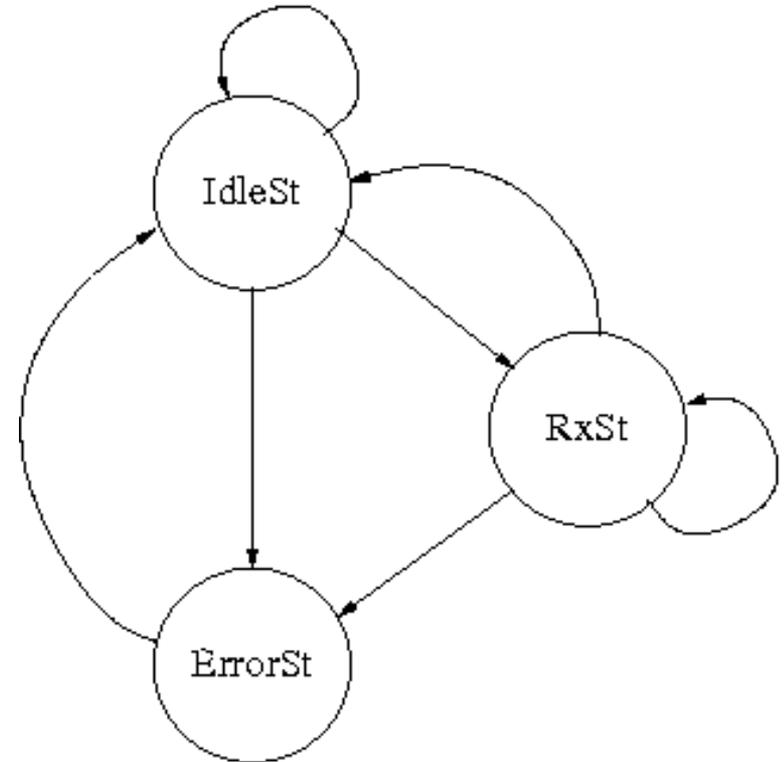
- Προτεινόμενο στυλ υλοποίησης FSM
 - Η συνδυαστική λογική καταστάσεων σε always block (πάντα default)
 - Ο καταχωρητής κατάστασης σε ένα ξεχωριστό always block (clocked - πάντα reset)
 - Έξοδοι είτε από το always της CL είτε από wires

Απλή FSM (1/2)

```
module fsm #(
    parameter dh = 1)
(
    input    Clk,
    input    Reset_n,
    input    i_start,
    input    i_stop,
    input    i_error,
    output   o_receive,
    output   o_error);
//
localparam IdleSt  = 3'b001,
             RxSt   = 3'b010,
             ErrorSt = 3'b100;
//
reg [2:0] State;
//
```

Απλή FSM (2/2)

```
always @(posedge Clk) begin
  if (~Reset_n) State <= #dh IdleSt;
  else begin
    case(State)
    IdleSt : begin
      if(i_error)
        State <= #dh ErrorSt;
      else if(i_start)
        State <= #dh RxSt;
      else State <= #dh IdleSt;
    RxSt : begin
      if(i_error)
        State <= #dh ErrorSt;
      else if(i_stop)
        State <= #dh IdleSt;
      else State <= #dh RxSt;
    end
    ErrorSt : State <= #dh IdleSt;
    default : State <= #dh IdleSt;
  endcase
end
assign o_receive = (State== RxSt);
assign o_error = (State==ErrorSt);
endmodule
```



hex2ascii

```
module hex2ascii(  
    input  [3:0] i_in,  
    output [7:0] o_out);  
//  
    assign o_out = (i_in>9) ? (8'h37 + i_in) : (8'h30 + i_in);  
//  
endmodule
```

ascii2hex

```
module ascii2hex(  
    input      [7:0] i_in,  
    output reg [3:0] o_out,  
    output reg   o_error);  
//  
always @(i_in) begin  
    if((i_in>8'h2F)&&(i_in<8'h3A)) begin  
        o_out  = i_in - 8'h30;  
        o_error = 1'b0;  
    end  
    else if((i_in>8'h40)&&(i_in<8'h47)) begin  
        o_out  = i_in - 8'h37;  
        o_error = 1'b0;  
    end  
    else o_error = 1'b1;  
end  
//  
endmodule
```

7segm_dec

```
module 7segm_dec(  
    input    [3:0] i_digit,  
    output reg [7:0] o_out);  
//  
always @(i_digit) begin  
    case(i_digit)  
        4'h0 : o_out = 7'b0000001; // 0  
        4'h1 : o_out = 7'b1001111; // 1  
        4'h2 : o_out = 7'b0010010; // 2  
        4'h3 : o_out = 7'b0000110; // 3  
        4'h4 : o_out = 7'b1001100; // 4  
        4'h5 : o_out = 7'b0100100; // 5  
        4'h6 : o_out = 7'b0100000; // 6  
        4'h7 : o_out = 7'b0001111; // 7  
        4'h8 : o_out = 7'b0000000; // 8  
        4'h9 : o_out = 7'b0000100; // 9  
        4'hA : o_out = 7'b0001000; // A  
        4'hb : o_out = 7'b1100000; // B  
        4'hC : o_out = 7'b0110001; // C  
        4'hd : o_out = 7'b1000010; // D  
        4'hE : o_out = 7'b0110000; // E  
        4'hF : o_out = 7'b0111000; // F  
        default: o_out = 7'b0000001; // 0  
    endcase  
end  
endmodule
```