ΗΥ220
Εργαστήριο Ψηφιακών Κυκλωμάτων

Εαρινό Εξάμηνο

2026

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_ff @(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
  - fork … join
- 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_ff @(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_ff @(posedge clk)
  ShowValues(counter);
...
endmodule
```

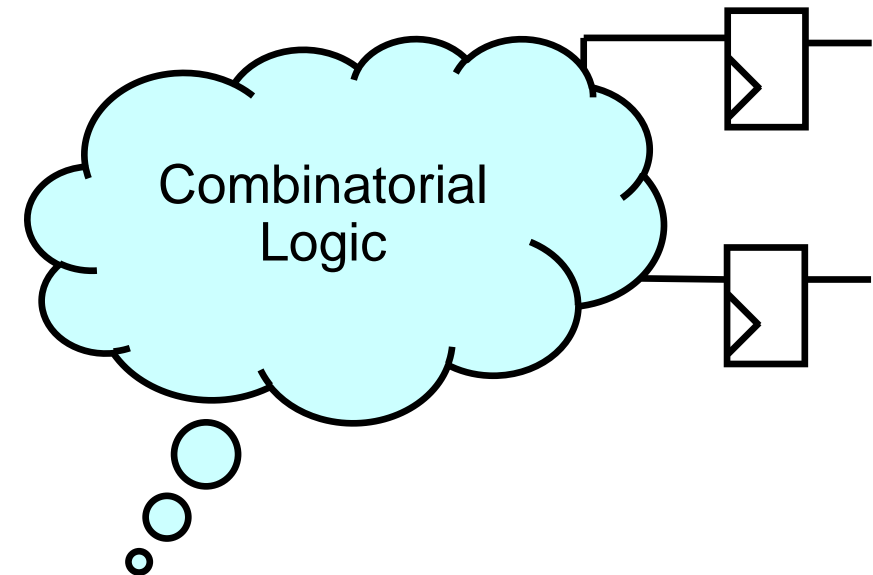
```
'define period 20

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 μοντέλο 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

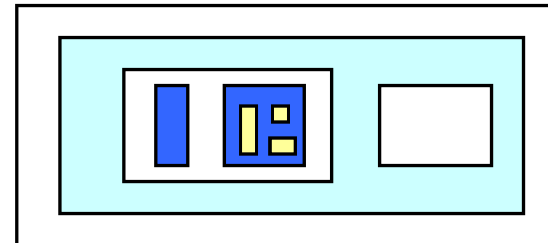
```
module top;
logic clk, reset;
logic [31:0] d_data, I_data;
logic [9:0] d_adr;
logic [5:0] i_adr;

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!
- **Χρησιμοποιήστε indentation**
  - Καλύτερη ομαδοποίηση
  - Αναγνωσιμότητα

```
logic a, controller_data_now_ready;
logic 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)

Synthesizable Verilog Constructs

Construct Type	Keywords	Notes
ports	input, output and inout	
parameters	parameter	
module definition	module, endmodule	
signals and variables	logic, wire, reg, tri	
instantiations	module instances, primitive gates	e.g. mymux(o,i0,i1,s) e.g. nand(out,a,b)
procedural	always_ff, always_comb, if, else, case	initial almost not supported
procedural blocks	begin, end	
data flow	assign	Delay ignored
Operators	+,-, &, |, ~, != , == , etc	Caution: * , / , %
functions / tasks	function, task	Limited support (simple CL)
Loops	for, while	Limited support (assigns)

Register – D Flip Flop

```
module Reg #(
    parameter int N   = 16,
    parameter int C2Q = 1 )
(
    input  logic         clk,
    input  logic [N-1:0] i_d,
    output logic [N-1:0] o_q);
//
   always_ff @(posedge clk)
     o_q <= #C2Q i_d;
//
endmodule
```

Register with Asynchronous Reset

```
module RegARst #(
    parameter int N = 16,
    parameter int C2Q = 1 )
  (
    input  logic         clk,
    input  logic         reset_n,
    input  logic [N-1:0] i_d,
    output logic [N-1:0] o_q)
//
   always_ff @(posedge clk or negedge reset_n) begin
      if (~reset_n)
        o_q <= #C2Q 0;
      else
        o_q <= #C2Q i_d;
   end
endmodule
```

Register with Synchronous Reset

```
module RegSRst #(
    parameter int N = 16,
    parameter int C2Q = 1 )
  (
    input  logic         clk,
    input  logic         reset_n,
    input  logic [N-1:0] i_d,
    output logic [N-1:0] o_q)
//
   always_ff @(posedge clk) begin
      if (~reset_n)
        o_q <= #C2Q 0;
      else
        o_q <= #C2Q i_d;
   end
endmodule
```

Register with Load Enable

```
module RegLd #(
   parameter int N = 16,
   parameter int C2Q = 1)
  (
   input  logic         clk,
   input  logic         i_ld,
   input  logic [N-1:0] i_d,
   output logic [N-1:0] o_q);
//
   always_ff @(posedge clk)
      if (i_ld)
        o_q <= #C2Q i_d;
//
endmodule
```

Set Clear flip-flop with Strong Clear

```
module scff_sc #(
    parameter int C2Q = 1 )
  (
    input  logic clk
    input  logic i_set,
    input  logic i_clear,
    output logic o_out);
//
 always_ff @(posedge clk)
      o_out <= #C2Q (o_out | i_set) & ~i_clear;
//
endmodule
```

```
// the simpler equivalent version
always_ff @(posedge clk)
   if (i_clear)      o_out <= #C2Q 0;
   else if (i_set) o_out <= #C2Q 1;
```

Set Clear flip-flop with Strong Set

```
module scff_ss #(
    parameter int C2Q = 1 )
  (
    input  logic clk
    input  logic i_set,
    input  logic i_clear,
    output logic o_out);
//
 always_ff @(posedge clk)
      o_out <= #C2Q i_set | (o_out & ~i_clear);
//
endmodule
```

```
// the simpler equivalent version
always_ff @(posedge clk)
   if (i_set)          o_out <= #C2Q 1;
   else if (i_clear) o_out <= #C2Q 0;
```

T Flip Flop

```
module Tff #(
    parameter int C2Q = 1 )
  (
    input  logic clk,
    input  logic rst,
    input  logic i_toggle,
    output logic o_out);
//
always_ff @(posedge clk)
    if (rst)
        o_out <= #C2Q 0;
    else if (i_toggle)
        o_out <= #C2Q ~o_out;
//
endmodule
```

Multiplexor 2 to 1

```
module mux2 #(
   parameter int N = 16)
(
   output logic [N-1:0] o_out,
   input  logic [N-1:0] i_in0,
   input  logic [N-1:0] i_in1,
   input  logic         i_sel);
//
 assign o_out = i_sel ? i_in1 : i_in0;
//
endmodule
```

Multiplexor 4 to 1

```
module mux4 #(
   parameter int N  = 32)
  (
   input  logic [N-1:0] i_in0,
   input  logic [N-1:0] i_in1,
   input  logic [N-1:0] i_in2,
   input  logic [N-1:0] i_in3,
   input  logic [  1:0] i_sel,
   output logic [N-1:0] o_out);
//
   always_comb 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
```

Positive Edge Detector

```
module PosEdgDet #(
   parameter int C2Q = 1)
 (
   input  logic clk,
   input  logic i_in,
   output logic o_out);
//
 logic tmp;
 always_ff @(posedge clk)
    tmp <= #C2Q i_in;
//
 assign o_out = ~tmp & i_in;
//
endmodule
```

Negative Edge Detector

```
module NegEdgDet #(
    parameter int C2Q = 1)
  (
    input  logic clk,
    input  logic i_in,
    output logic o_out);
//
 logic tmp;
 always_ff @(posedge clk)
    tmp <= #C2Q i_in;
//
 assign o_out = tmp & ~i_in;
//
endmodule
```

Edge Detector

```
module EdgDet #(
   parameter int C2Q = 1)
  (
   input  logic clk,
   input  logic i_in,
   output logic o_out);
//
 logic tmp;
 always_ff @(posedge clk)
   tmp <= #C2Q i_in;
//
 assign o_out = tmp ^ i_in;
//
endmodule
```

Tristate Driver

```
module Tris #(
   parameter int N = 32)
(
input  logic [N-1:0] i_tris_in,
input  logic         i_tris_oen_n,
inout  logic [N-1:0] o_tris_out);
//
 assign o_tris_out = ~i_tris_oen_n ? i_tris_in : 'bz;
//
endmodule
```

Up Counter

```
module Cnt #(
   parameter int N      = 32,
   parameter int MAXCNT = 100,
   parameter int C2Q    = 1)
  (
   input  logic         clk,
   input  logic         i_en,
   input  logic         i_clear,
   output logic         o_zero,
   output logic [N-1:0] o_out);
//
  always_ff @(posedge clk) begin
      if(i_clear) begin
          o_out  <= #C2Q 0;
          o_zero <= #C2Q 1;
      end
      else if (i_en) begin
         if (o_out==MAXCNT) begin
             o_out  <= #C2Q 0;
             o_zero <= #C2Q 1;
         end
         else begin
             o_out  <= #C2Q o_out + 1'b1;
             o_zero <= #C2Q 0;
         end
      end
  end
endmodule
```

```
// a cleaner SystemVerilog RTL version
logic zero_d, zero_q;
logic [N-1:0] out_d, out_q;
// flip-flops below with non-blocking assignments
always_ff @(posedge clk) begin
  out_q  <= #C2Q out_d;
  zero_q <= #C2Q zero_d;
end
// combinatorial logic below with blocking assignments
always_comb begin
  out_d  = out_q;   // keep previous value - default
  zero_d = zero_q;  // keep previous value - default

  if(i_clear) begin
    out_d  = 0;
    zero_d = 1;
  end
  else if (i_en) begin
    if (out_q == MAXCNT) begin
      out_d  = 0;
      zero_d = 1;
    end
    else begin
      out_d  = out_q + 1;
      zero_d = 0;
    end
  end
end
// output assignments
assign o_out  = out_q;
assign o_zero = zero_q;
```

Parallel to Serial Shift Register

```
module P2Sreg #(
    parameter int N   = 32,
    parameter int C2Q = 1)
 (
    input  logic         clk,
    input  logic         reset_n,
    input  logic         i_ld,
    input  logic         i_shift,
    input  logic [N-1:0] i_in,
    output logic         o_out);
//
   logic [N-1:0] tmp_val;
//
 always_ff @(posedge clk or negedge reset_n) begin
    if (~reset_n) tmp_val <= #C2Q 0;
    else begin
       if (i_ld) tmp_val <= #C2Q i_in;
       else if(i_shift) tmp_val <= #C2Q tmp_val >> 1;
    end
 end
//
 assign o_out = tmp_val[0];
//
endmodule
```

Serial to Parallel Shift Register

```
module S2Preg #(
    parameter int N   = 32,
    parameter int C2Q = 1)
 (
    input  logic         clk,
    input  logic         i_clear,
    input  logic         i_shift,
    input  logic         i_in,
    output logic [N-1:0] o_out);
//
 always_ff @(posedge clk) begin
    if (i_clear)
       o_out <= #C2Q 0;
    else if (i_shift)
       o_out <= #C2Q {o_out[N-2:0],i_in};
 end
//
endmodule
```

Barrel Shift Register

```
module BarShiftReg(
   parameter int N   = 32,
   parameter int C2Q = 1)
  (
   input  logic         clk,
   input  logic         reset_n,
   input  logic         i_ld,
   input  logic         i_shift,
   input  logic [N-1:0] i_in,
   output logic [N-1:0] o_out);
//
 always_ff @(posedge clk) begin
    if (~reset_n) o_out <= #C2Q 0;
    else begin
       if (i_ld)
          o_out <= #C2Q i_in;
       else if (i_shift)
          o_out <= #C2Q {o_out[N-2:0],o_out[N-1]};
    end
 end
//
endmodule
```

3 to 8 Binary Decoder

```
module dec #(
    parameter int NLOG = 3)
  (
    input  logic [NLOG-1:0]        i_in,
    output logic [((1<<NLOG))-1:0] o_out);
//
int i;
//
 always_comb begin
    for (i=0; i<(1<<NLOG); i++) 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 int NLOG = 3)
 (
  input  logic [((1<<NLOG)-1):0] i_in,
  output logic [NLOG-1:0]        o_out);
//
int i;
//
 always_comb begin
    o_out = 'x;
    for (i=0; i<(1<<NLOG); i++) begin
       if (i_in[i]) o_out = i;
    end
 end
//
endmodule
```

Priority Enforcer Module

```
module PriorEnf #(
    parameter int N = 8)
  (
    input  logic [N-1:0] i_in,
    output logic [N-1:0] o_out,
    output logic         o_found);
//
int i;
always_comb begin
    o_found = 0;
    for (i=0; i<N; i++) begin
       if (i_in[i] & ~o_found) begin
            o_found = 1;
            o_out[i] = 1;
       end
       else o_out[i] = 0;
    end
end
endmodule
```

Latch

```
module Latch #(
   parameter int N = 16,
   parameter int D2Q = 1)
 (
   input  logic         i_ld,
   input  logic [N-1:0] i_in,
   output logic [N-1:0] o_out);
//
 always_latch begin
      if (i_ld)
        o_out = #D2Q i_in;
 end
//
endmodule:
```

Combinatorial Logic and Latches (1/3)

```
module mux3 #(
        parameter int N   = 32 )
(       input  logic [  1:0] sel,
        input  logic [N-1:0] in2,
        input  logic [N-1:0] in1,
        input  logic [N-1:0] in0,
        output logic [N-1:0] out);
   always_comb 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 int N  = 32 )
(       input  logic [  1:0] sel,
        input  logic [N-1:0] in2,
        input  logic [N-1:0] in1,
        input  logic [N-1:0] in0,
        output logic [N-1:0] out);
    always_comb 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_comb** blocks και **logic** τότε πρέπει να αναθέτουμε τιμές στις εξόδους της λογικής για όλες τις πιθανές περιπτώσεις εισόδων (κλήσεις του **always_comb**) !!!
  - 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)