HY220
Εργαστήριο Ψηφιακών
Κυκλωμάτων

Χειμερινό Εξάμηνο
2019-2020

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)

- More expressions
  - for / while
  - functions
  - tasks
  - fork ... join

- More types
  - integer
  - real
  - arrays

```vhdl
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
```
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 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;
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...
  - O compiler ξέρει καλύτερα
  - Αναγνωσιμότητα

• Δομή
  - Πολλές οντότητες
  - Ε όχι και τόσες!

• Χρησιμοποιήστε indentation
  - Καλύτερη ομαδοποίηση
  - Αναγνωσιμότητα

wire a, controller_data_now_ready;
wire drc_rx_2, twra_malista;

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

ΗΥ220 - Βασίλης Παπαευσταθίου
... 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

```vhd
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
```

ΗΥ220 - Βασίλης Παπαευσταθίου 9
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
    o Register Inference, combinatorial logic
  – Βελτιστοποίηση του netlist (area,speed)
## Synthesizable Verilog Constructs

<table>
<thead>
<tr>
<th>Construct Type</th>
<th>Keywords</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ports</td>
<td>input, output and inout</td>
<td></td>
</tr>
<tr>
<td>parameters</td>
<td>parameter</td>
<td></td>
</tr>
<tr>
<td>module definition</td>
<td>module, endmodule</td>
<td></td>
</tr>
<tr>
<td>signals and variables</td>
<td>wire, reg, tri</td>
<td></td>
</tr>
<tr>
<td>instantiations</td>
<td>module instances, primitive gates</td>
<td>e.g. mymux(o,i0,i1,s) e.g. nand(out,a,b)</td>
</tr>
<tr>
<td>procedural</td>
<td>always, if, else, case</td>
<td>initial almost not supported</td>
</tr>
<tr>
<td>procedural blocks</td>
<td>begin, end</td>
<td></td>
</tr>
<tr>
<td>data flow</td>
<td>assign</td>
<td>Delay ignored</td>
</tr>
<tr>
<td>Operators</td>
<td>+, -, &amp;,</td>
<td>, ~, != , == , etc</td>
</tr>
<tr>
<td>functions / tasks</td>
<td>function, task</td>
<td>Limited support (simple CL)</td>
</tr>
<tr>
<td>Loops</td>
<td>for, while</td>
<td>Limited support (assigns)</td>
</tr>
</tbody>
</table>
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
module RegARst #(
    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
module RegSRst #(
    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
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

```verilog
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
```
module scff_ss #(  
  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
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**

```verilog
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:0] In0,
    input [N:0] In1,
    input [N:0] In2,
    input [N:0] In3,
    input [1:0] Sel,
    output reg [N: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
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
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
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
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
endmodule
Parallel to Serial Shift Register

module P2Sreg #(
    parameter N = 32,
    parameter dh = 1)
(
    input     Clk,
    input     Reset_n,
    input     l_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 (l_Ld) TmpVal <= #dh i>In;
            else if (i_Shift) TmpVal <= #dh TmpVal>>1;
        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
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) begin
        if (i_In[i] & DetectNot) begin
            o_Out[i]=1;
            DetectNot=0;
        end
        else o_Out[i]=0;
    end
    OneDetected = !DetectNot;
end
endmodule
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;
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
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)