Latvijas Universitāte
Datorikas fakultāte

Aparatūras apraksta valodas. Verilog.

Kurss "ievads digitālajā projektēšanā"
Lekcija 09.11.2012

Autors: Rinalds Ruskuls

Lekcijas saturs

- HDL valodu pielietojums
- Verilog valodas pamati
  - Veriog programmas sastāvdaļas
  - Objekti
  - Datu tipi
- Ciparu shēmu veidošana izmantojot Verilog

Shēmu aprakstīšanas veidi

- Līdz šim iepazināmies ar ciparu shēmtehnikas pamatiem!
- Ir arī citi veidi kā izveidot ciparu principiālo elektrisko shēmu:
  - Pielietojot programmēšanas valodas (**H**ardware **D**escription **L**anguages – ***HDL***)
  - Kombinējot kopā – HDL un shēmtehniku

Ciparu shēmas projektēšana izmantojot HDL valodu

```
module seven_segment_ctrl                          // Moduļa nosaukums
      (
         input [3:0] data_in,                       // Ieeju definēšana
         output reg [6:0] data_out                  // Izeju definēšana
      );

      Always@(*)                                    // loģikas daļa
            case(data_in)
                  4'b0000:  data_out <= 7'b1111110;
                  4'b0001:  data_out <= 7'b0110000;
                  4'b0010:  data_out <= 7'b1101101;
                  4'b0011:  data_out <= 7'b1111001;
                  4'b0100:  data_out <= 7'b0110011;
                  4'b0101:  data_out <= 7'b1011011;
                  4'b0110:  data_out <= 7'b1011111;
                  4'b0111:  data_out <= 7'b1110000;
                  4'b1000:  data_out <= 7'b1111111;
                  4'b1001:  data_out <= 7'b1111011;
                  default: data_out <= 7'b1111110;
            endcase
endmodule
```

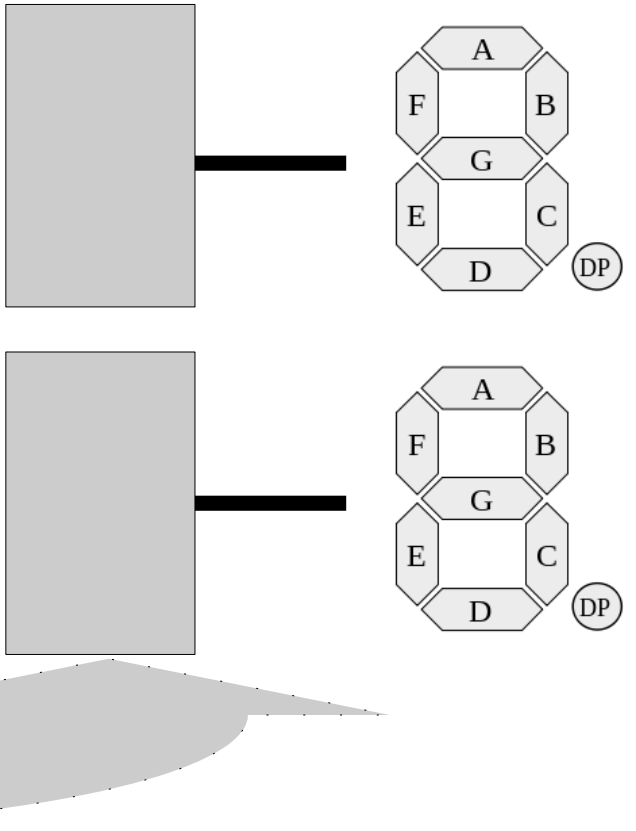
Ciparu shēmas projektēšana kombinējot HDL valodas un shēmtehniku

a

Ir iespējams izmantot dažāda veida ***HDL*** valodas un izveidotos moduļus izmantot kā shēmas blokus!

```
module seven_segment_ctrl
        (
            input [3:0] data_in,
            output reg [6:0] data_out
        );

        Always@(*)
                case(data_in)
                        4'b0000:        data_out <= 7'b1111110;
                        4'b0001:        data_out <= 7'b0110000;
                        4'b0010:        data_out <= 7'b1101101;
                        4'b0011:        data_out <= 7'b1111001;
                        4'b0100:        data_out <= 7'b0110011;
                        4'b0101:        data_out <= 7'b1011011;
                        4'b0110:        data_out <= 7'b1011111;
                        4'b0111:        data_out <= 7'b1110000;
                        4'b1000:        data_out <= 7'b1111111;
                        4'b1001:        data_out <= 7'b1111011;
                        default: data_out <= 7'b1111110;
                endcase
endmodule
```



Verilog

Kāpēc radās nepieciešamība pēc programmēšanas valodām, kas ir paredzētas ciparu elektronikai?

HDL kopīgais un atšķirīgais ar citām valodām

- Kopīgais ar klasiskajām programmēšanas valodām
  - Ir iespējams aprakstīt secīgas darbības
  - Ļauj strādāt dažādos abstrakciju līmeņos
- Atšķirīgais no klasiskajām programmēšanas valodām:
  - Ir iespējams aprakstīt procesus, kuri izpildās paralēli

HDL – hardware description language

- ***HDL*** valodas izmanto, lai formāli aprakstītu ciparu elektronikas:
  - shēmu uzbūvi
  - shēmas darbību
  - testus darbības simulēšanai
- ***HDL*** iedalās:
  - VHDL
  - ***Verilog***
  - System Verilog

Verilog HDL

- Verilog valoda ir līdzīga ***C/C++***, ***Python*** programmēšanas valodai
- Verilog valoda ir "case sensitive", visi rezervētie vārdi ir ar maziem burtiem

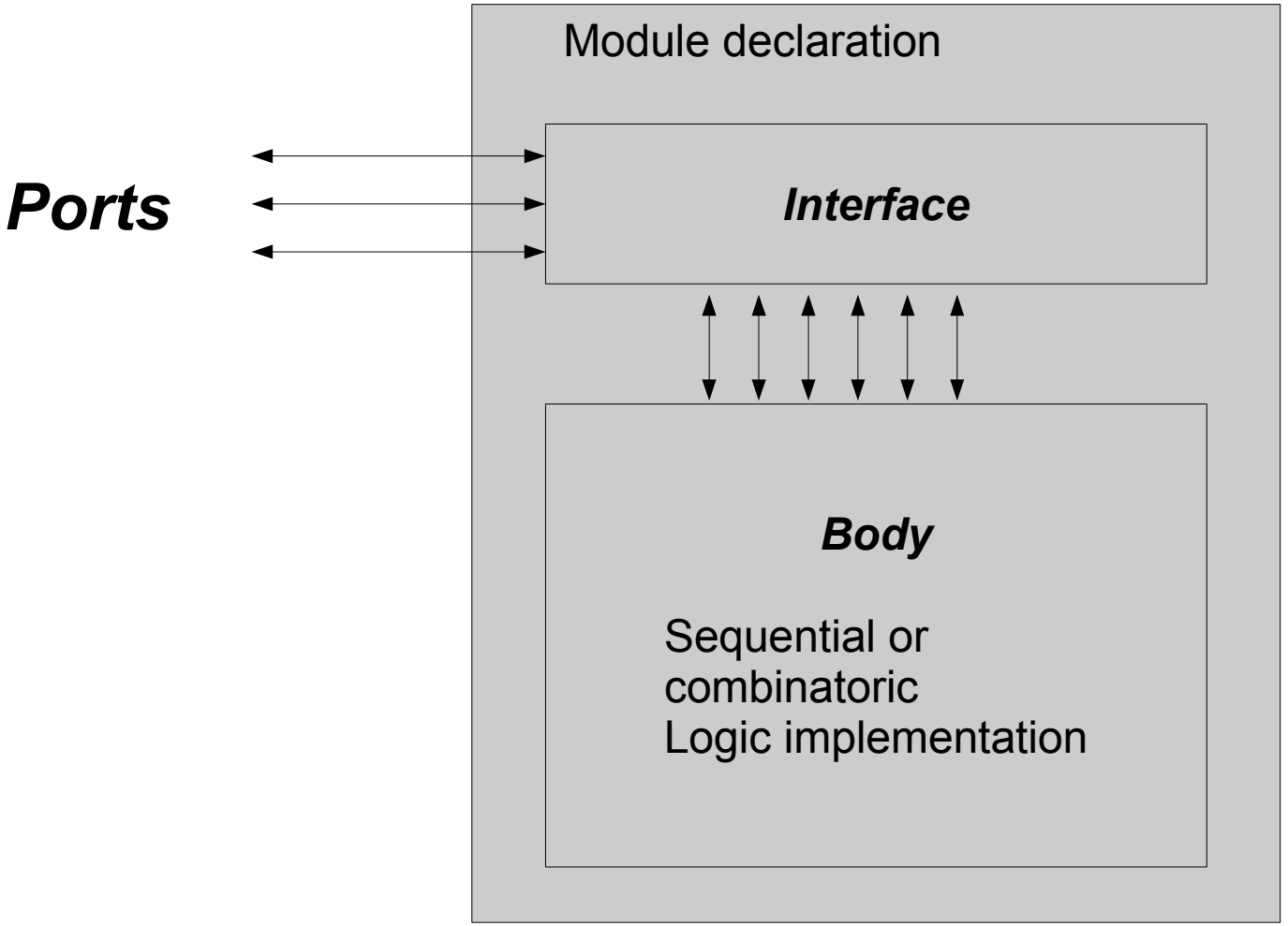
```
input // a Verilog Keyword
wire // a Verilog Keyword
WIRE // a unique name ( not a keyword)
Wire // a unique name (not a keyword)
```

Verilog HDL

- Verilog koda iekomentēšana ir iespējama divos veidos:
  - Vienas rindas komentārs //
  - Bloka komentārs /* ..... */
- Komentēšanas piemērs

```
module my_and (a, b, result)
    input a; // vienas rindas komentārs
    /* input b;    bloka komentārs
    output result; */
endmodule
```

Verilog satāvdaļas



Module declaration

- Pirmais solis ir definēt moduli (***module_name***)un izmantoto signālu nosaukumus (***sig_name_1, sig_name_2...***)
- Lietotāja definētiem nosaukumiem jāievēro šādi likumi
  - Jāsākas ar burtiem [a-z] vai [A-Z]
  - Nedrīkst sākties ar $ vai [0-9]
  - ***Case sensitivity*** – my_sig ≠ My_sig

```
module module_name (sig_name_1, sig_name_2,..., sig_name_n);

.....................

endmodule
```

Module declaration

- Otrais solis ir definēt signālu virzienu un datu tipu
  - Signālu virzieni var būt – ieeja, izeja, divvirzienu
  - Datu tipi – bits, bitu vektors, utt.

```
module module_name (sig_name_1, sig_name_2,..., sig_name_n);
    input sig_name_1;          // viens bits
    Input [3:0] sig_name_2;    // bitu vektors – 4 biti
    output  sig_name_n;        // viens bits;
endmodule
```

Module declaration - example

Pirmais solis ir definēt moduli un izmantoto signālu nosaukumus

```
module my_and (a, b, result)

.....................

endmodule
```

Otrais solis ir definēt moduļa ieejas un izjas.

```
module my_and (a, b, result)
      input a;
      input b;
      output result;
endmodule
```

Kas mums ir sanācis!?

- Izveidojām moduli, kuram ir:
  - Divas ieejas – a un b;
  - Viena izeja - result

Turpmākie darbi – definēt izveidotajam modulim tā veicamo funkcionalitāti

Funkcionaltitātes definēšana

- Kombinatoriskas shēmas veidošana

```
module my_and (a, b, result)
    input a;
    input b;
    output result;
    assign result = a & b;
endmodule
```

Funkcionaltitātes definēšana

- Kombinatoriskas shēmas veidošana

```
module my_and (a, b, result)
    input a;
    input b;
    output result;
    wire int_sig;
    assign int_sig = a | b;
    assign result = a & b ^ int_sig;
endmodule
```

Bitu līmeņa operandi

- & → AND
- | → OR
- ~ → NOT
- ^ → XOR
- ~^ vai ^~ → XNOR

Loģiskās operācijas

&&	→	AND
||	→	OR
!	→	NOT

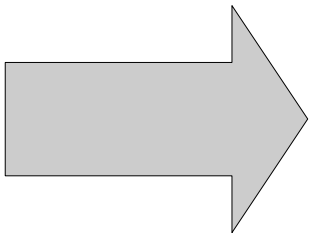
Izejā ir viena bita vērtība – 0,1 vai x

A = 6;	A && B	→	1 && 0	→ 0;
B = 0;	A || !B	→	1 || 1	→ 1;
C = x;	C || B	→	x || 0	→ x;

Salikšanas operācija

- ***{op_1, op_2, ..}*** - saliek signālu no vairākiem atsevišķiem signāliem

```
reg a;
reg [2:0] b, c;
..
a = 1'b 1;
b = 3'b 010;
c = 3'b 101;
```



```
Catx = {a, b, c};        // 1_010_101
Caty = {b, 2'b11, a}; // 010_11_1
```

Secīgas shēmas veidošana – skaitītājs

- Ieejas signālu deklarēšana

```
module counter
    (
        input clk,
        input enable_in,
        output en_out,
        output [3:0] counter_out
    );

    reg [3:0] cnt_reg       = 4'b0000;
    reg [3:0] cnt_next      = 4'b0000;
    reg en_next             = 1'b0;
    reg en_reg              = 1'b0;
```

Secīgas shēmas veidošana – skaitītājs

- Sinhronā daļa

```
// state reg process
always@(posedge clk) begin
    cnt_reg     <= cnt_next;
    en_reg      <= en_next;
end
```

Secīgas shēmas veidošana – skaitītājs

- Asinhronā (kombinatoriskā) daļa

```
// combinatoric logic
always@(*) begin
        if (enable_in == 1'b0) begin
                cnt_next <= cnt_reg;
                en_next   <= 1'b0;
        end
        else begin
                if (cnt_reg == 9) begin
                        cnt_next <= 4'b0000;
                        en_next   <= 1'b1;
                end
                else begin
                        cnt_next <= cnt_reg + 1;
                        en_next   <= 1'b0;
                end
        end
end
```

Secīgas shēmas veidošana – skaitītājs

- Izejas daļa

```
    // output logic
    assign counter_out = cnt_reg;
    assign en_out = en_reg;

endmodule
```

Papildus uzdevumi mājās

- Patstāvīgi iepazīties ar Verilog valodu
  - Secīgu un paralēlu shēmu veidošanas principi
  - Izveidoto moduļu izsaukšana
  - simbola izveidošana no izveidotajiem moduļiem

Praktiskie darbi

- Strādājam pie kursa projekta

- Drīkst strādāt arī pie tiem mājasdarbiem, kuri vēl nav iesniegti

Paldies par uzmanību!
Jautājumi?