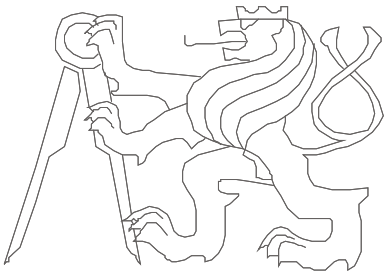
Pokročilé architektury počítačů

– Podklady pro cvičení –

Verilog - rychlokurz

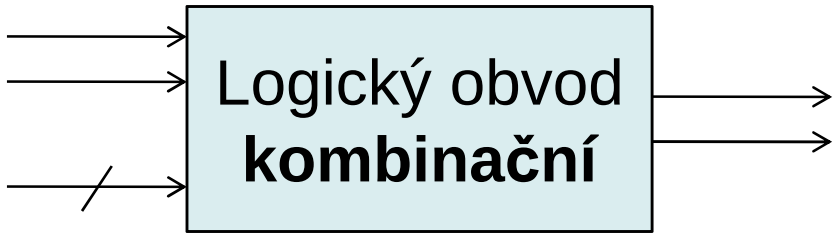


České vysoké učení technické, Fakulta elektrotechnická

HDL – *Hardware Description Language*

- Dva hlavní HDL:
  - **VHDL** (*VHSIC Hardware Description Language; VHSIC = Very High Speed Integrated Circuits)*
  - **Verilog**
  - Oba obsahují příkazy **ne**syntetizovatelné do hardvéru, ale užitečné při simulaci...
- Popis:
  - Behaviorální (jak se to bude chovat)
  - Strukturální (jak to bude vypadat / z čeho je sestaven)

Verilog – behaviorální popis

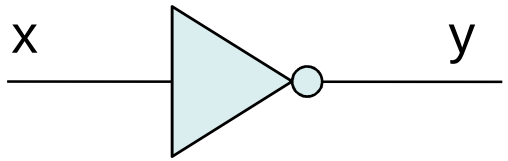



```
module název (input seznam_vstupů,
              output seznam_výstupů);

   wire deklarace_vnitřních_proměnných;

   assign přiřazení_výstupu;
   assign přiřazení_výstupu;
endmodule
```

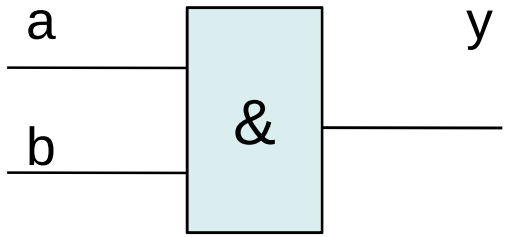
Modul – blok HW s vstupy a výstupy

Verilog – příklad

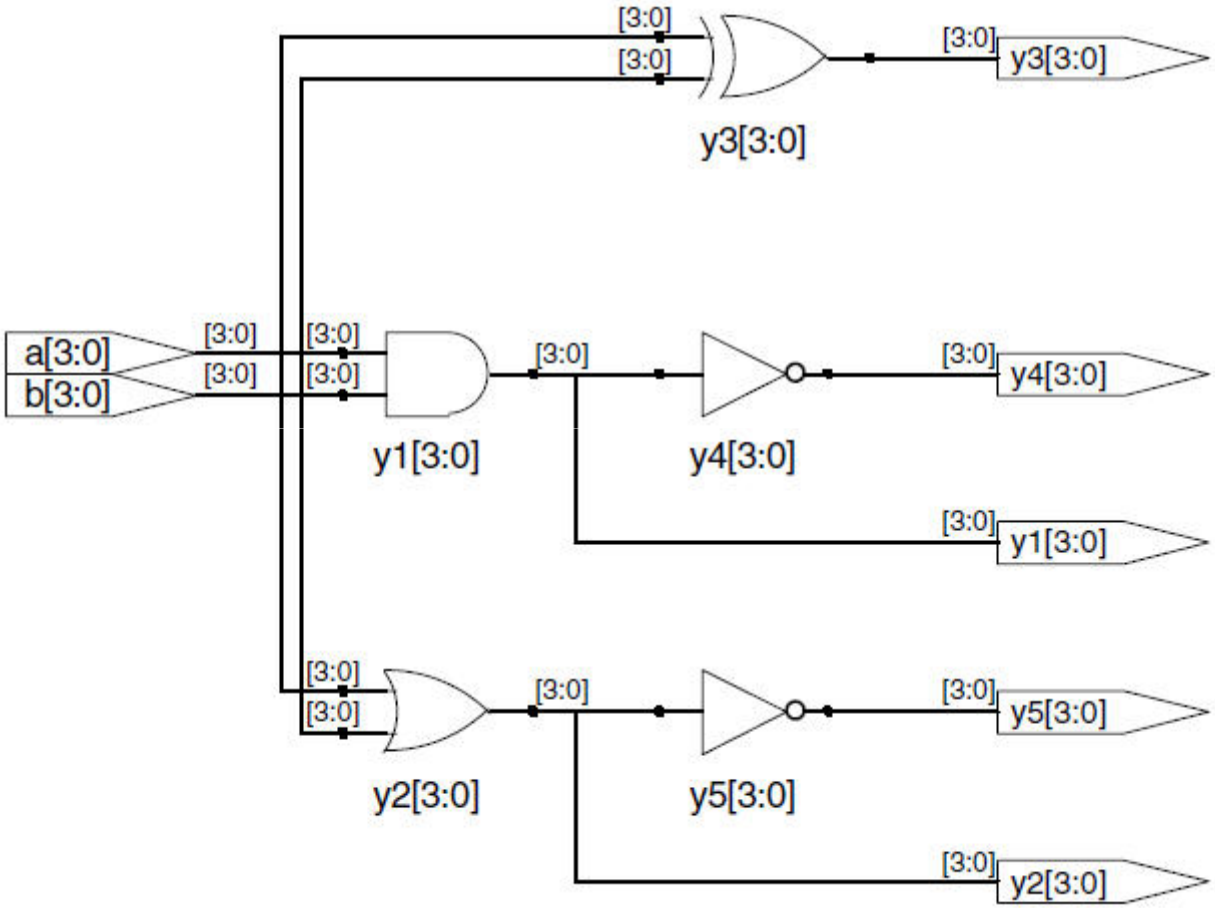
```
module inv (input x,
            output y);
     assign y = ~x;
endmodule
```




```
module and (input a, b,
            output y);
     assign y = a & b;
endmodule
```

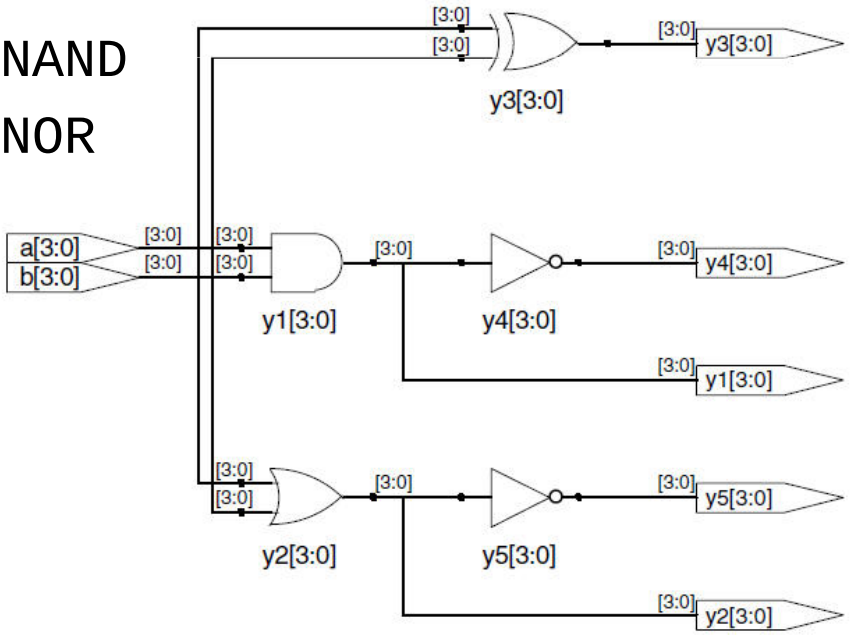


Verilog – příklad č.2

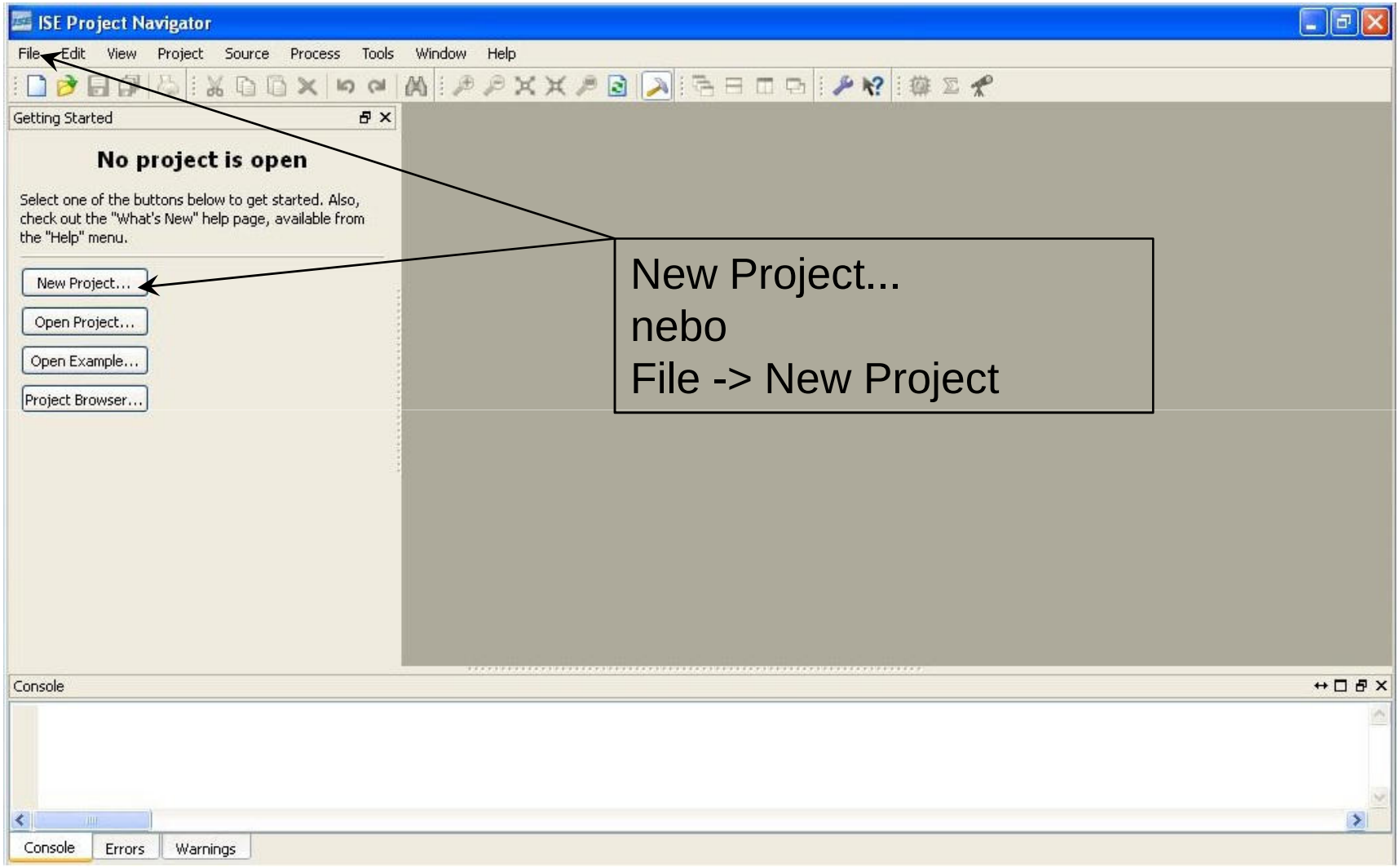


Verilog – příklad – behaviorální popis

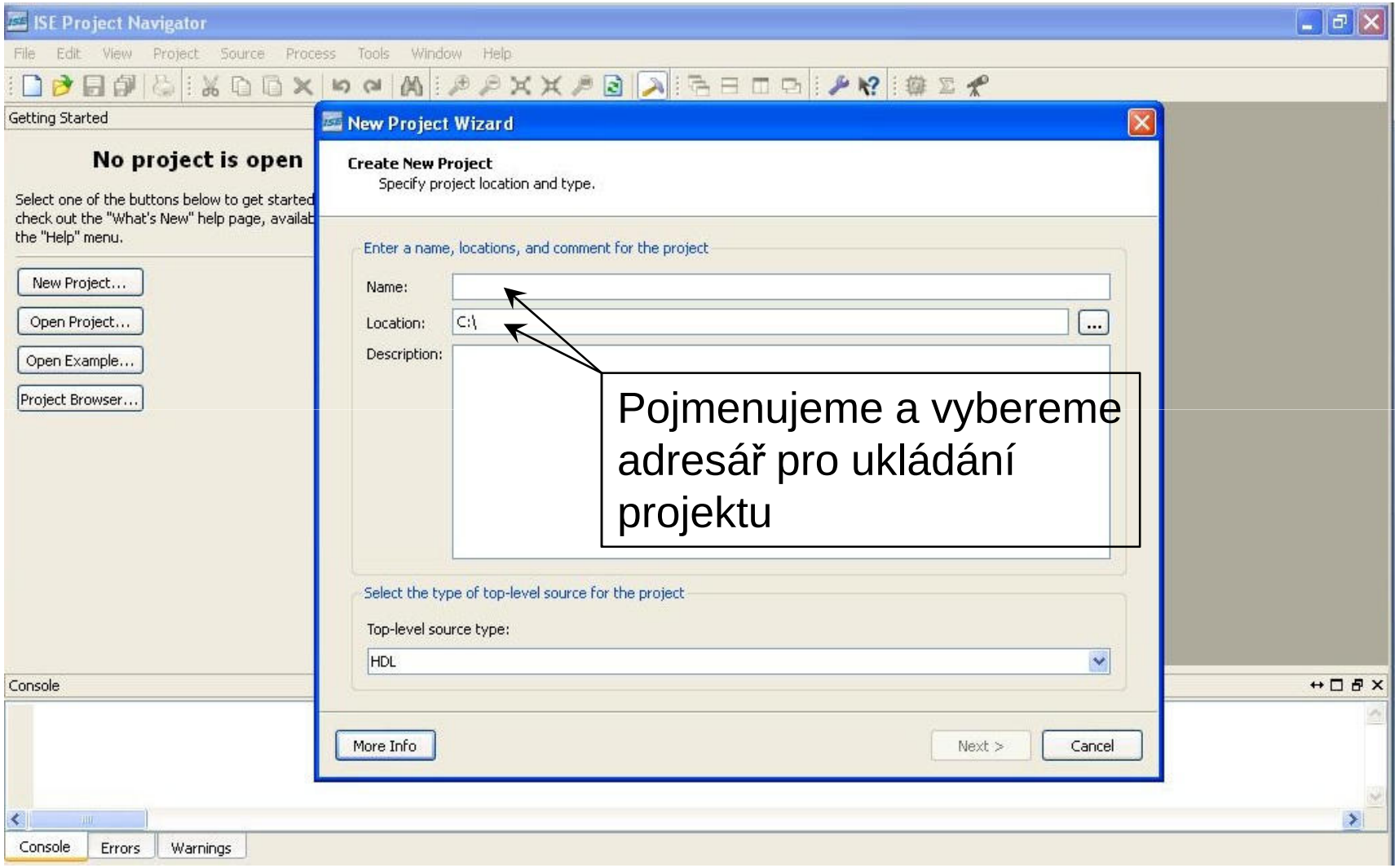
```
module gates (input [3:0] a, b,
              output [3:0] y1, y2,y3, y4, y5);

  assign y1 = a & b;     // AND
  assign y2 = a | b;     // OR
  assign y3 = a ^ b;     // XOR
  assign y4 = ~(a & b);  // NAND
  assign y5 = ~(a | b);  // NOR

endmodule
```



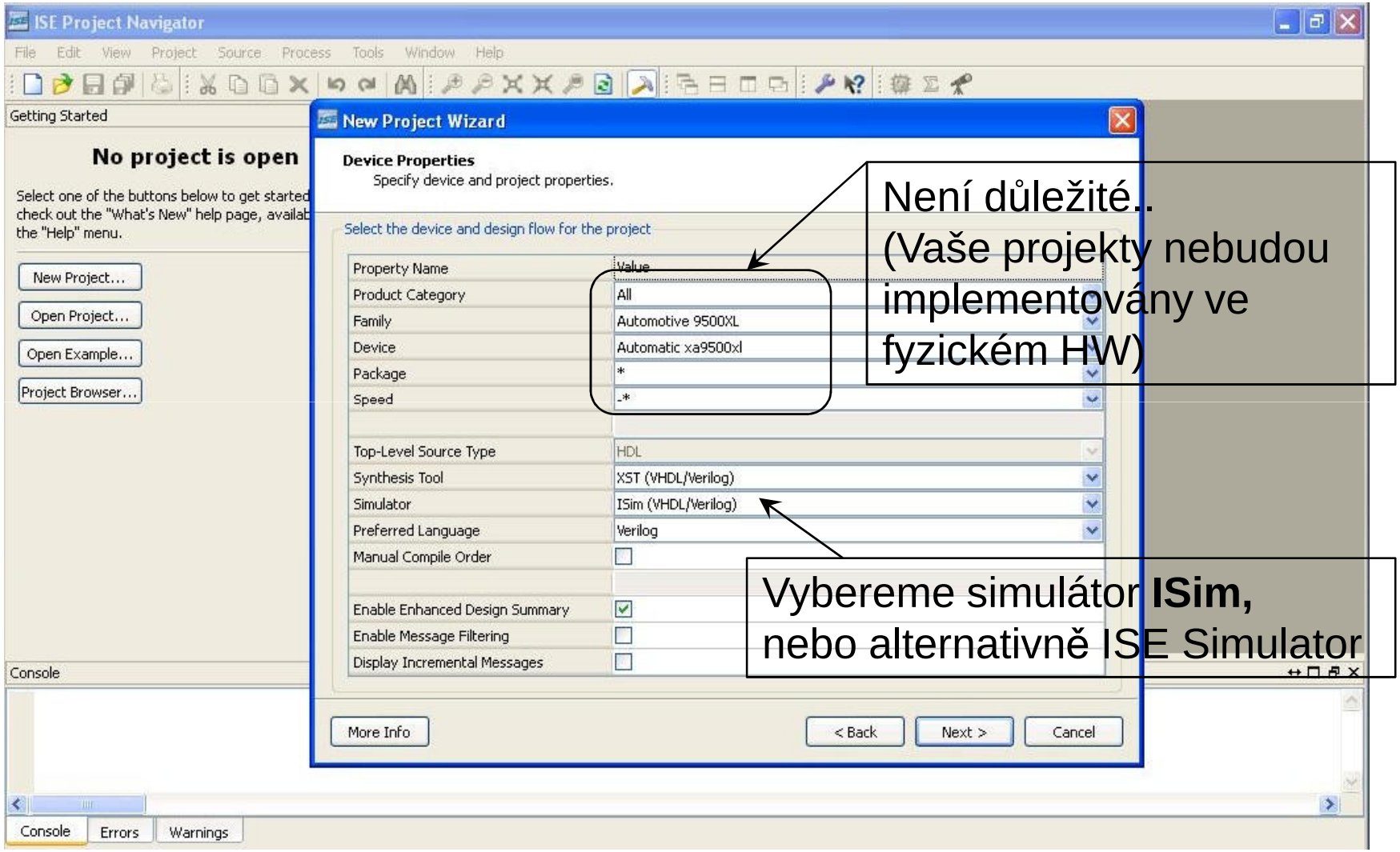
Založení nového projektu v prostředí Xilinx IDE



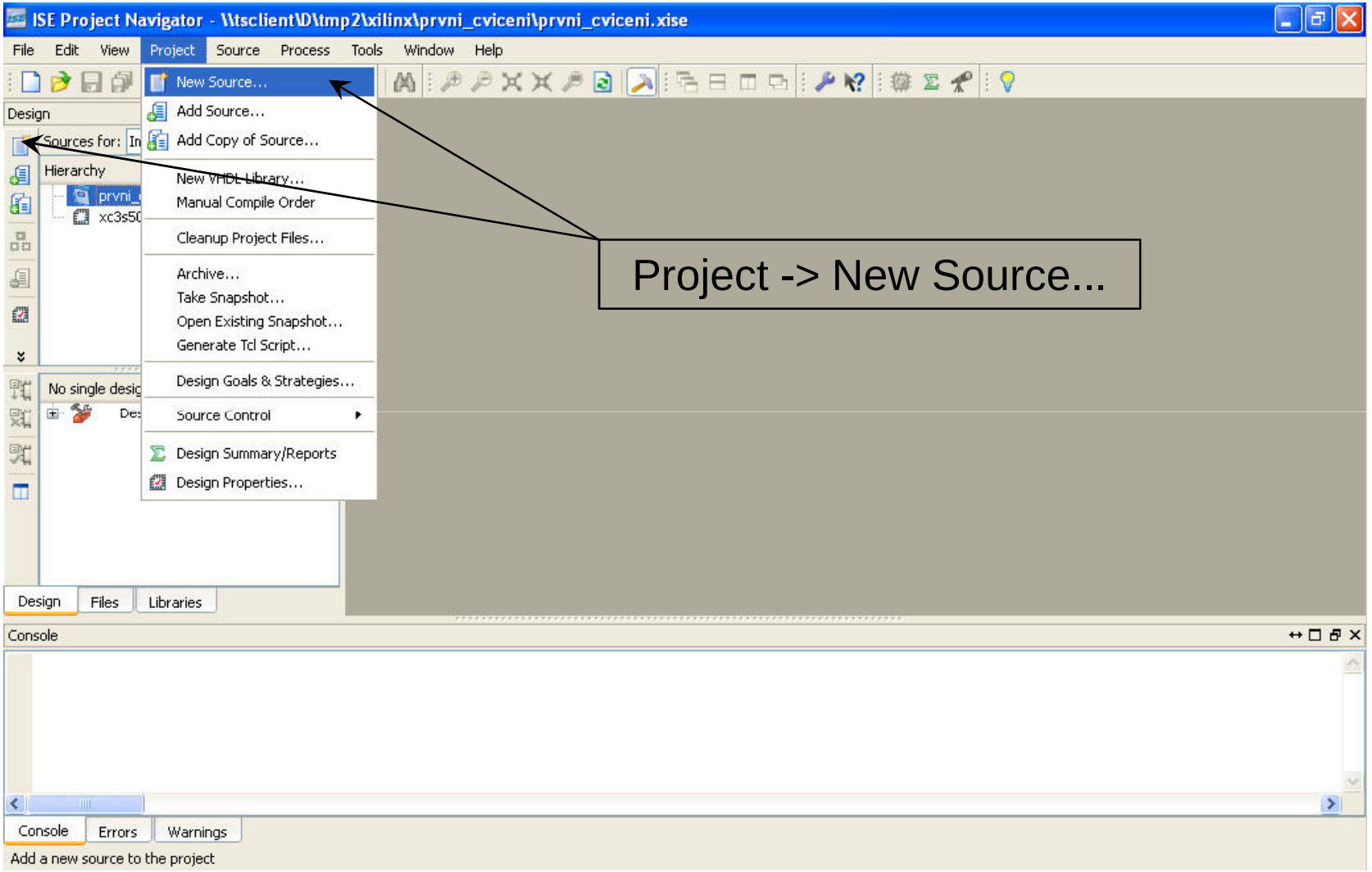
Založení nového projektu v prostředí Xilinx IDE



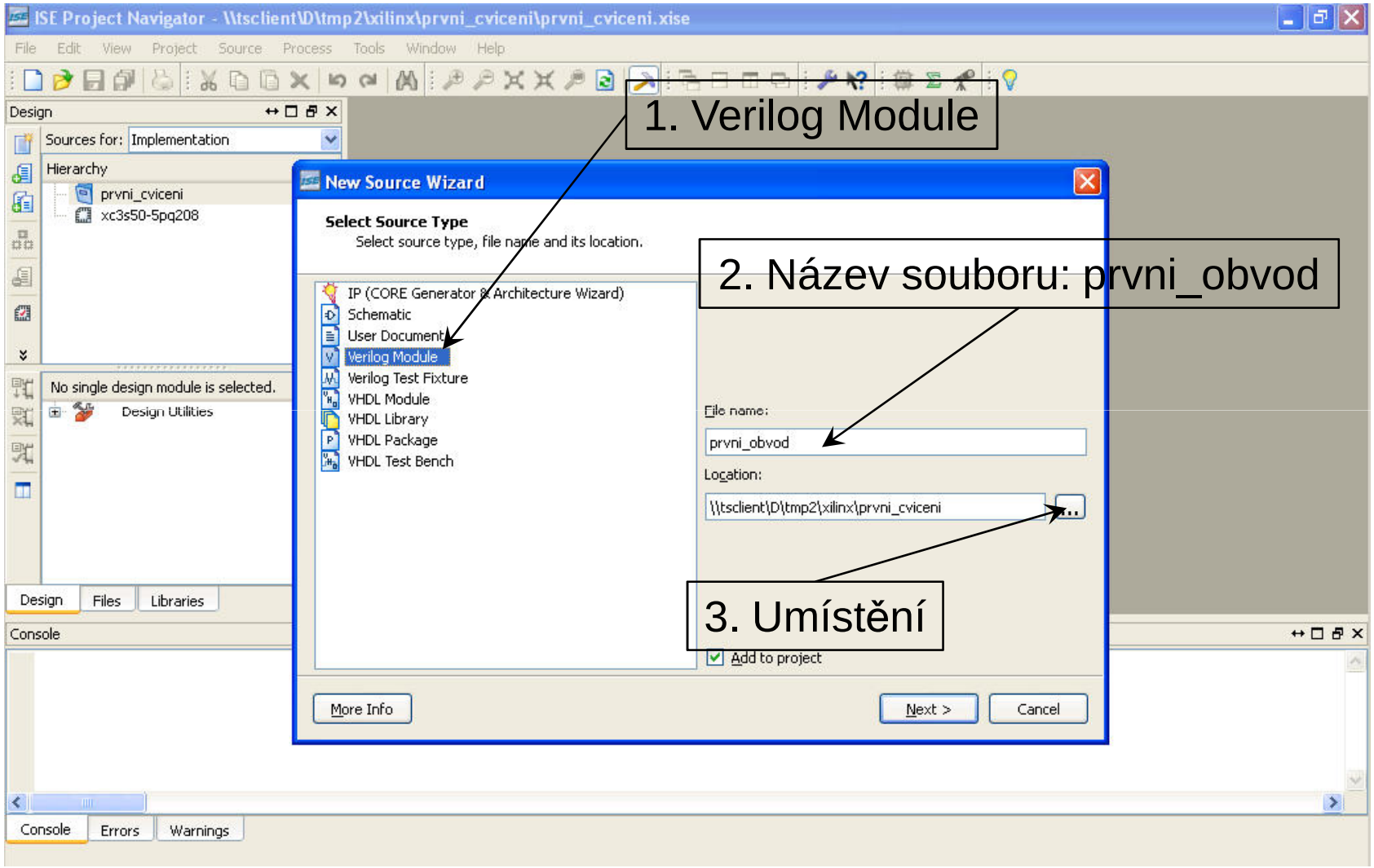
Založení nového projektu v prostředí Xilinx IDE



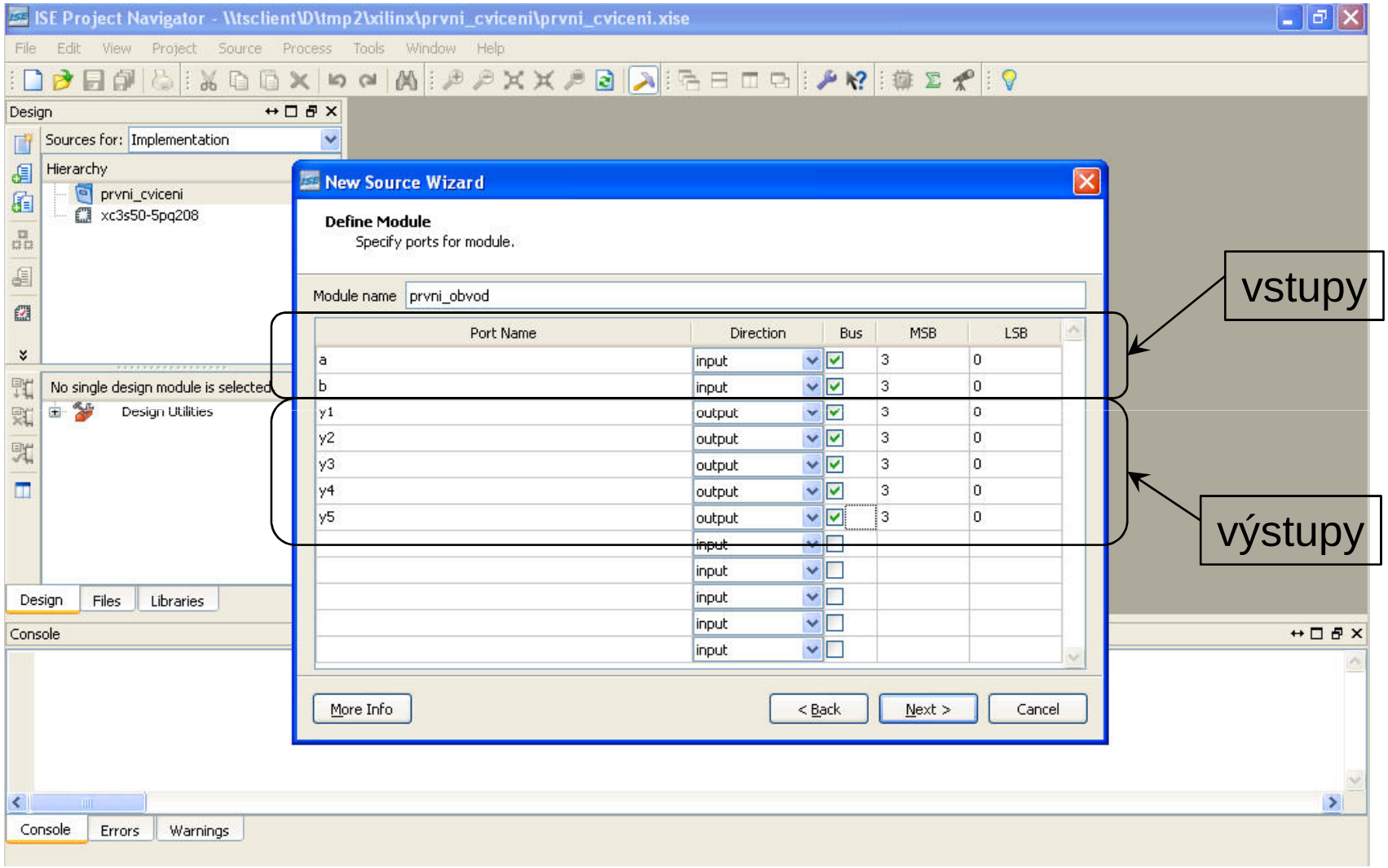
Popis tohoto obvodu v prostředí Xilinx IDE



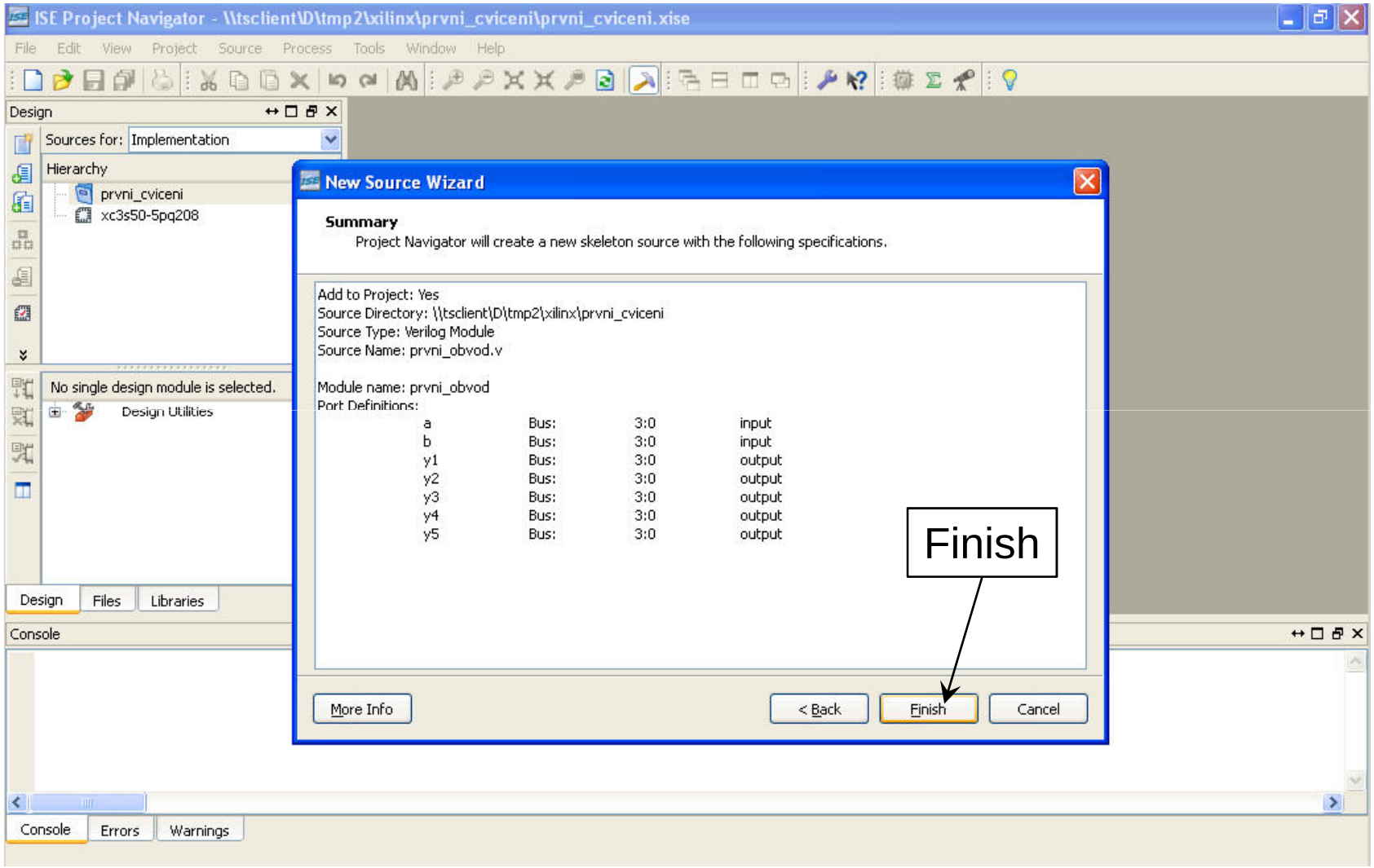
Popis tohoto obvodu v prostředí Xilinx IDE



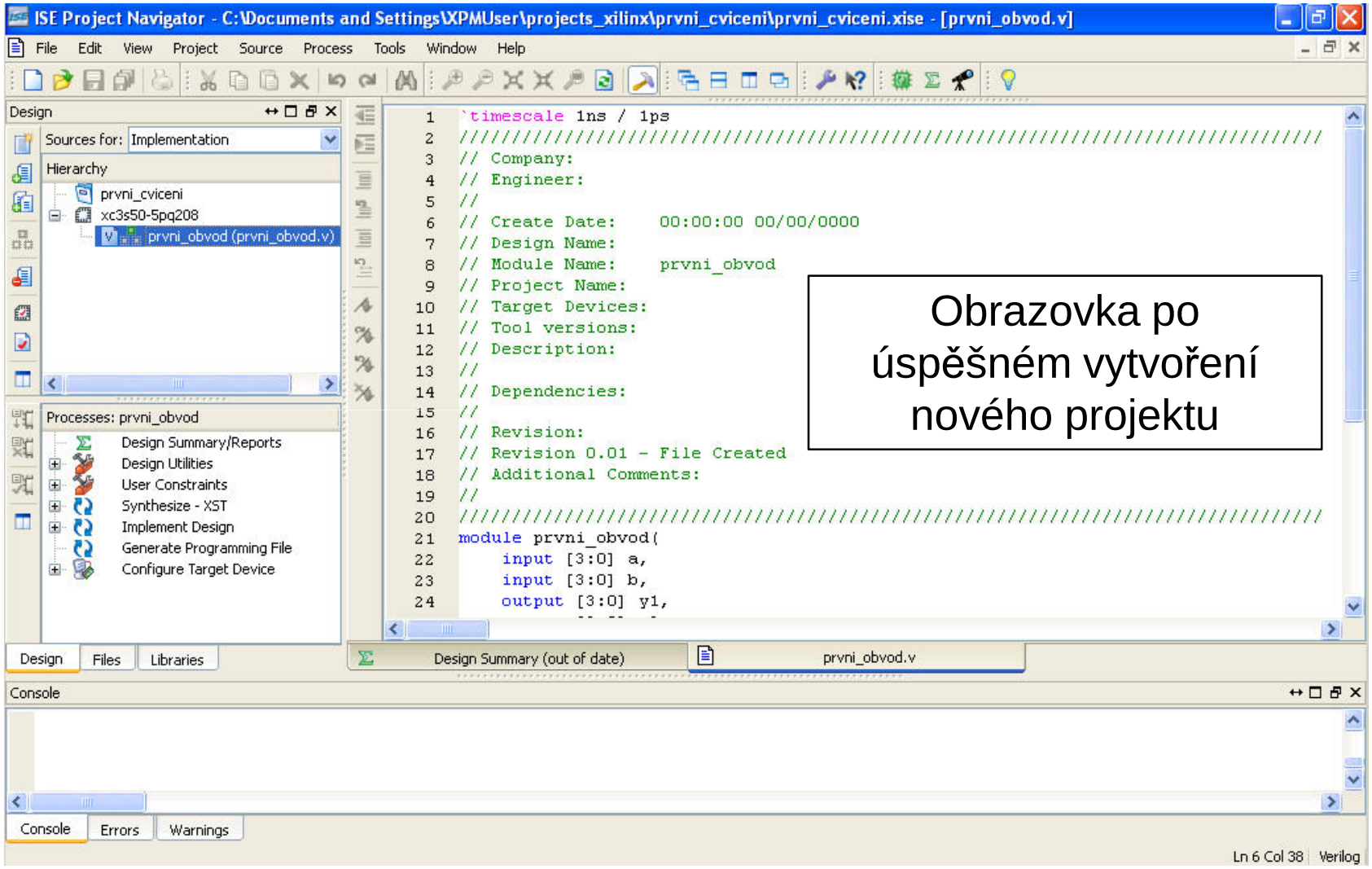
Popis tohoto obvodu v prostředí Xilinx IDE



Popis tohoto obvodu v prostředí Xilinx IDE

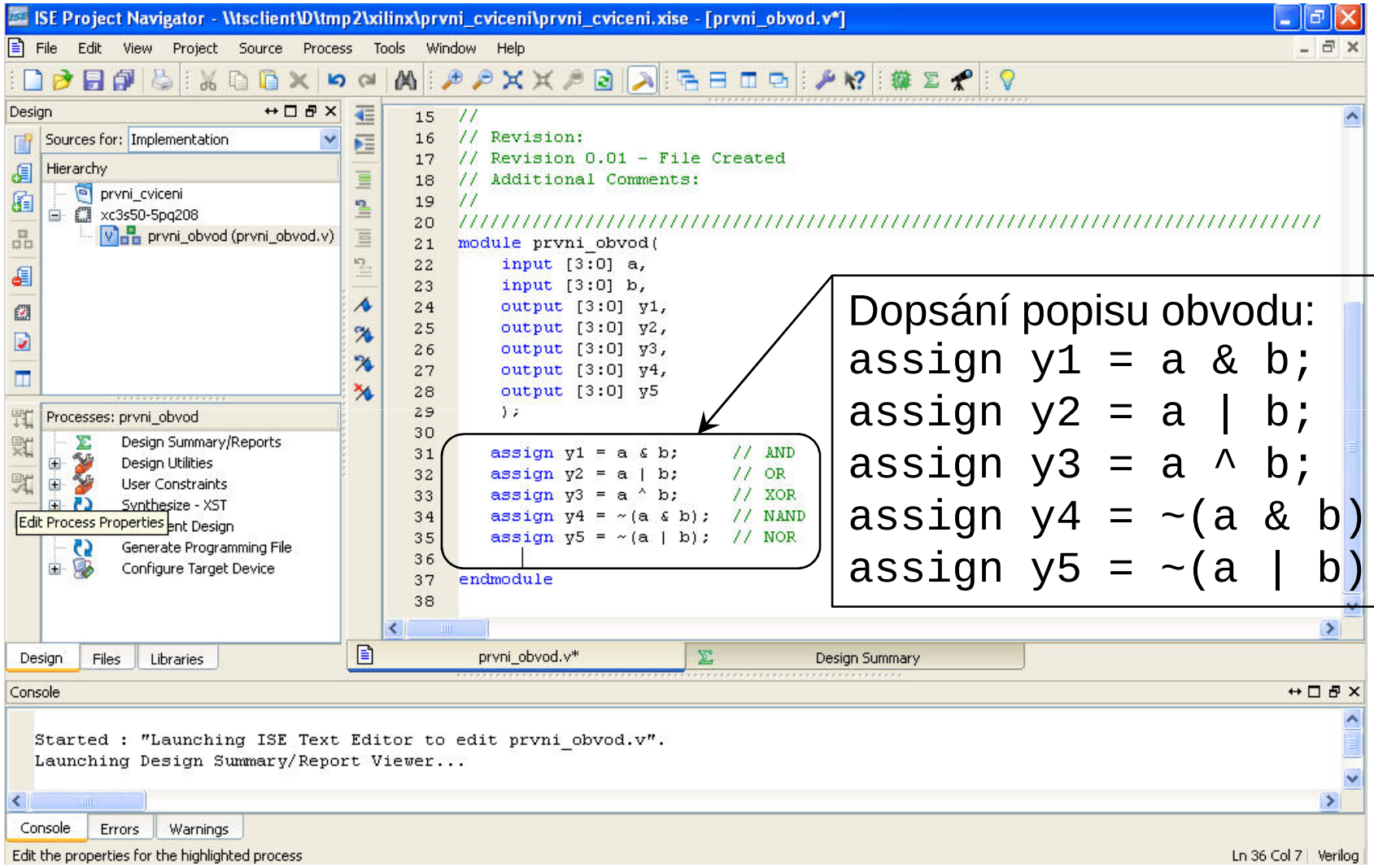


Popis tohoto obvodu v prostředí Xilinx IDE



Obrazovka po úspěšném vytvoření nového projektu

Popis tohoto obvodu v prostředí Xilinx IDE



Verilog – simulace – Obecně

- **Krok první** – „zapouzdřit“ simulovaný obvod do modulu bez vstupů a výstupů
- **Krok druhý** – vytvořit vnitřní proměnné tohoto modulu (reg, wire) pro nastavování vstupů (reg) a sledování výstupů (wire) simulovaného obvodu
- **Krok třetí** – přiřazení vytvořených vnitřních proměnných vstupům a výstupům simulovaného obvodu
- **Krok čtvrtý** – specifikace časové posloupnosti stimulů obvodu

Verilog – simulace – Obecně

```
`timescale 1ns / 1ps
module test();
    reg  a, b;   // vstupy
    wire  c;     // výstupy

    nazev_simulovaneho_obvodu   pojmenovani _instance(
            .a(a),
            .b(b),
            .c(c),
    );

    initial begin
            a = 0;
            b = 0;
    end

    always   #40 a = ~a;
    always   #80 b = ~b;
endmodule
```

žádné vstupy ani výstupy

vytvoření vnitřních proměnných

vytvoření instance simulovaného obvodu a přirazení vnitřních proměnných vstupům a výstupům simulovaného obvodu

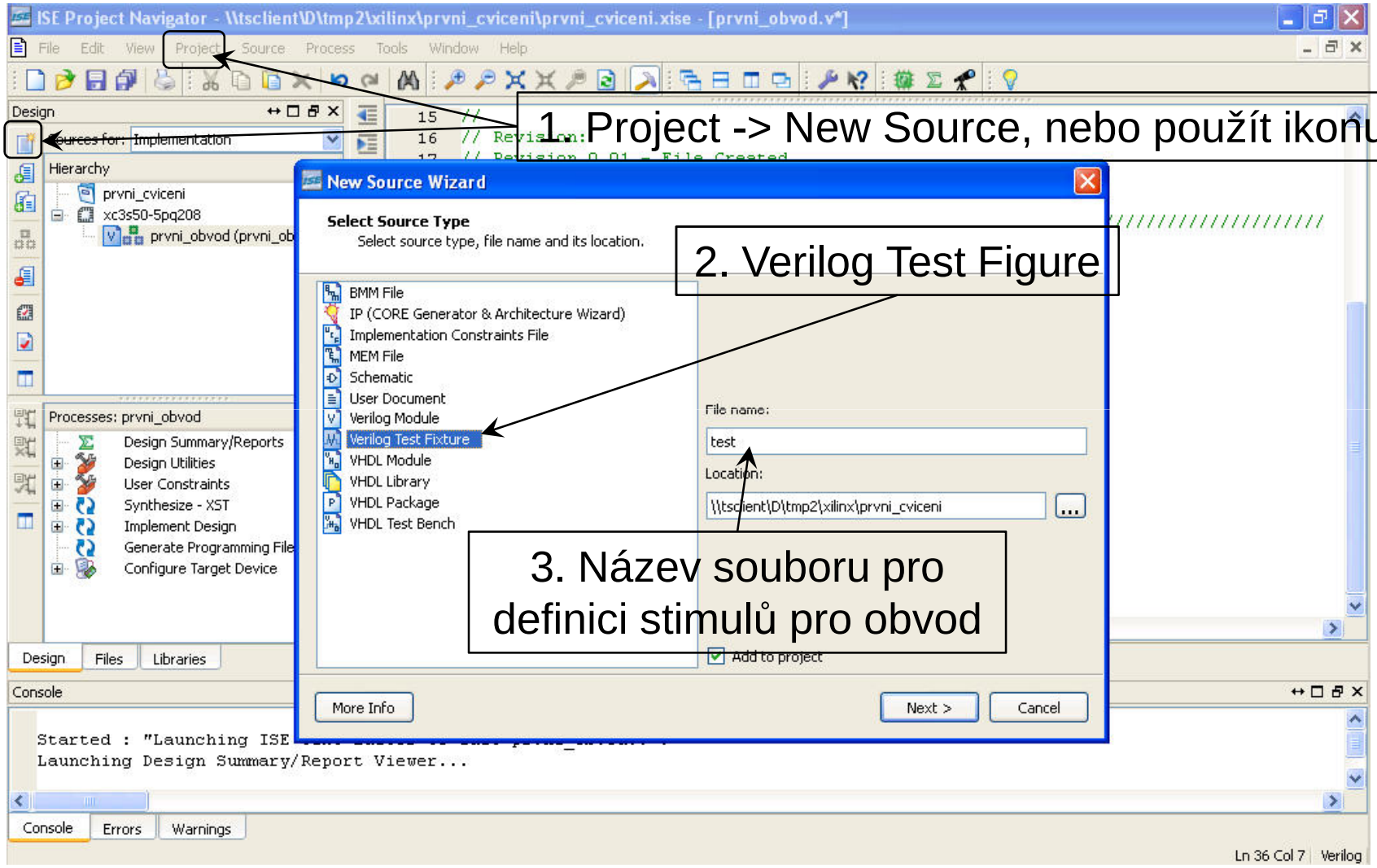
Signály s tečkou před jménem jsou názvy signálů uvnitř simulovaného obvodu, zatímco v závorkách jsou názvy vnitřních proměnných

inicializace vstupů

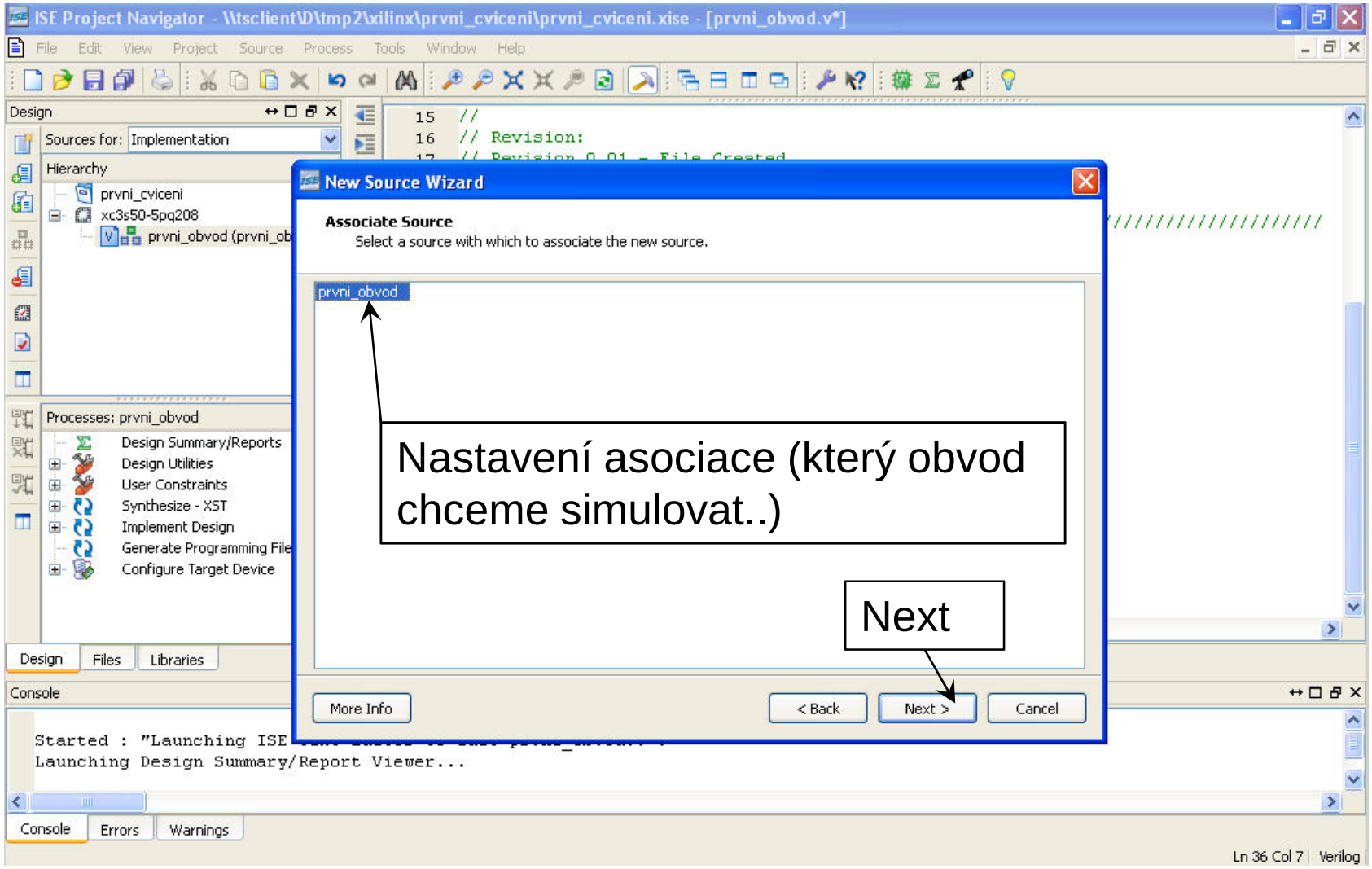
Všechny always bloky (vícero příkazů pro jeden always se ohraničuje mezi begin a end) jsou vykonávány současne…

invertuj b každých 80 ns

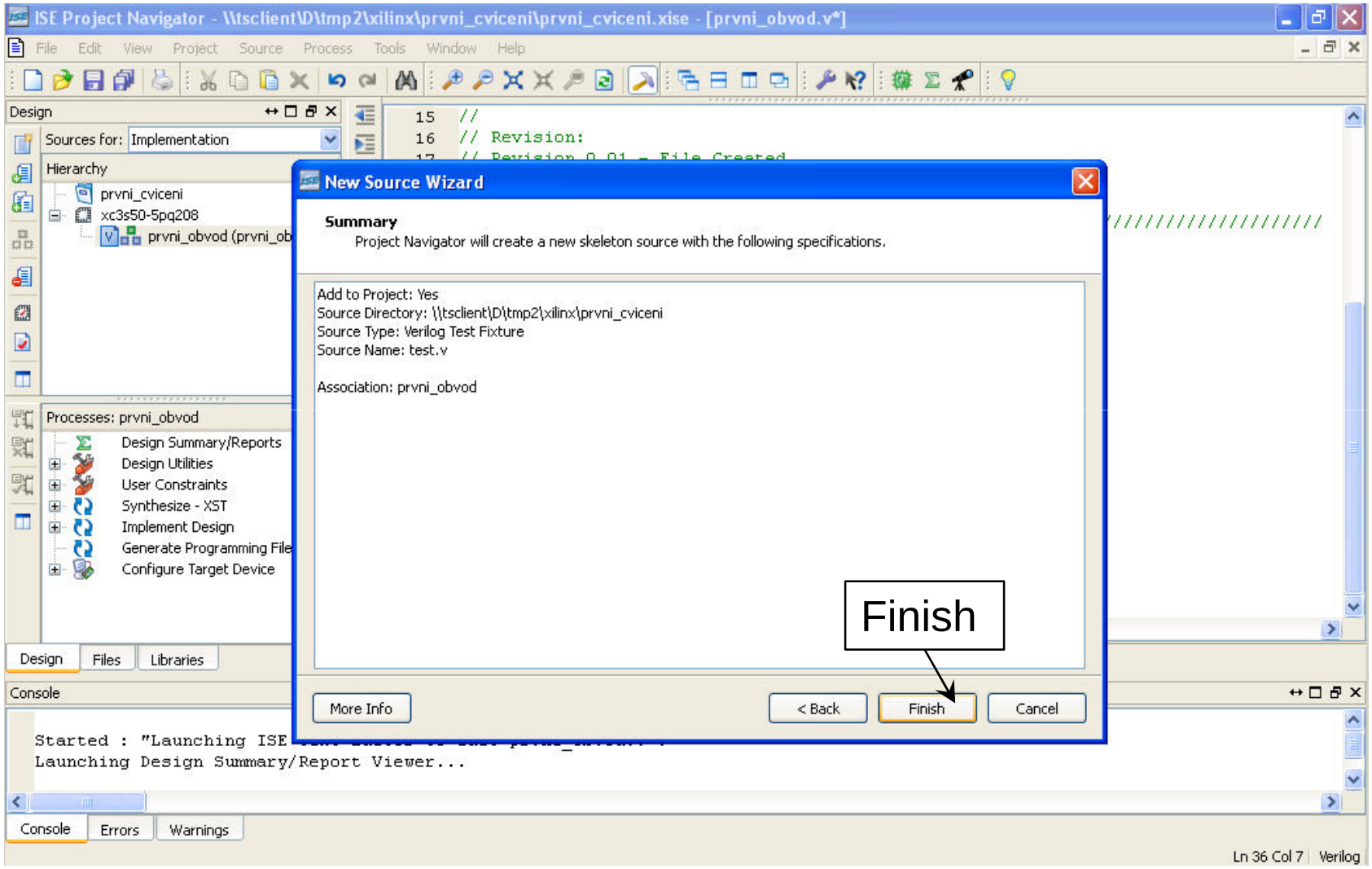
Simulace tohoto obvodu v prostředí Xilinx IDE



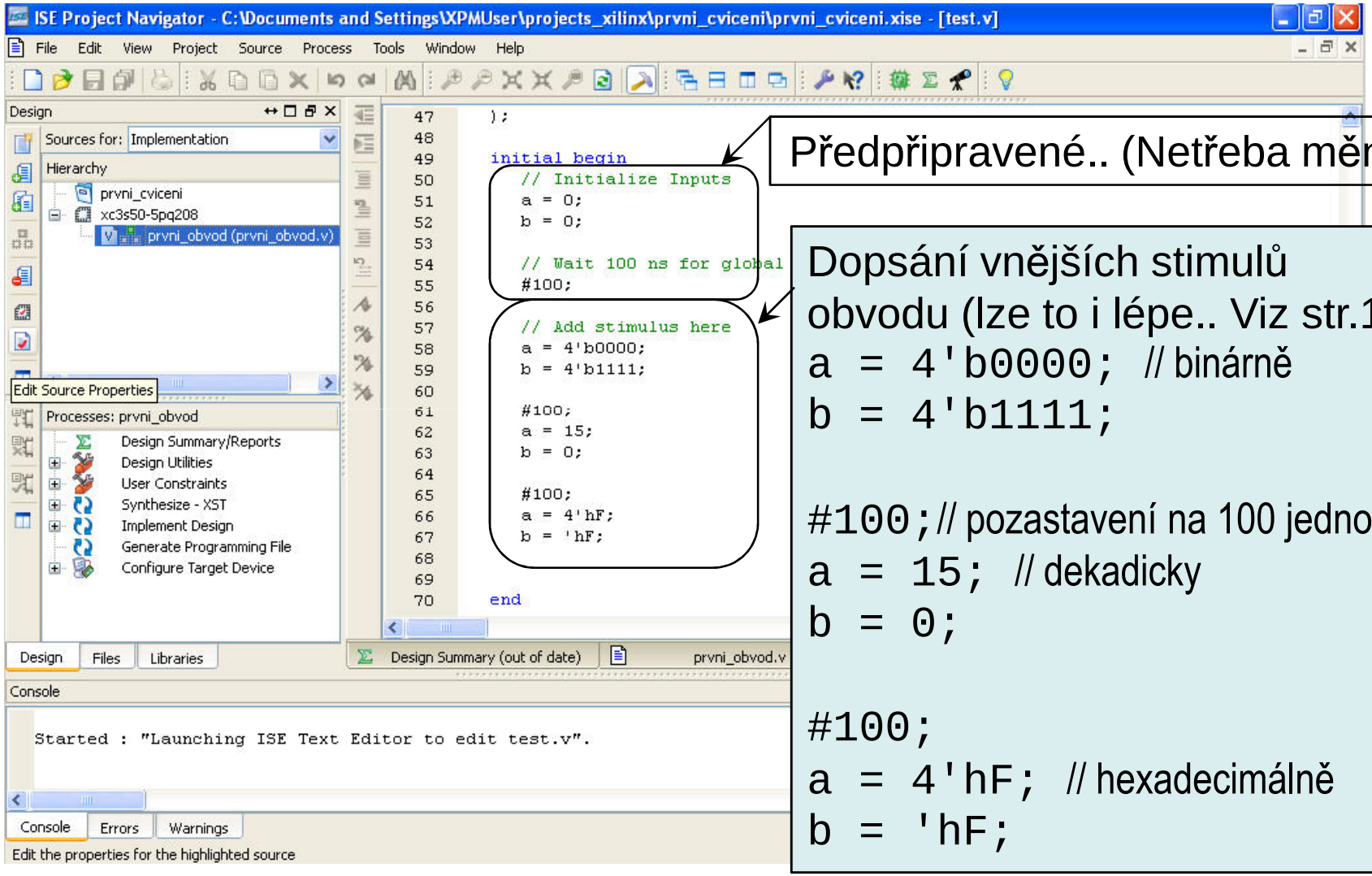
Simulace tohoto obvodu v prostředí Xilinx IDE



Simulace tohoto obvodu v prostředí Xilinx IDE



Simulace tohoto obvodu v prostředí Xilinx IDE

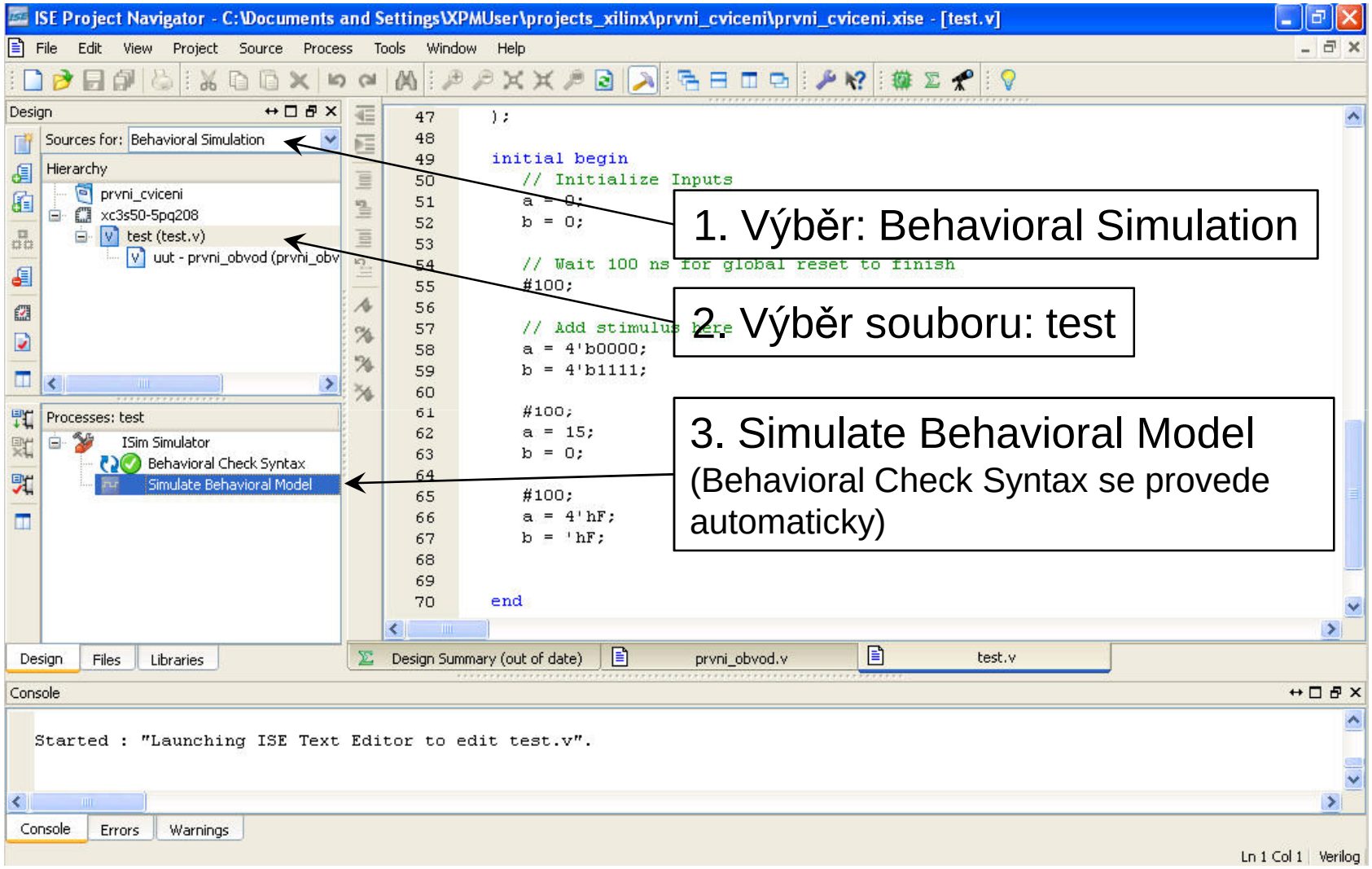


Předpřipravené.. (Netřeba měnit)

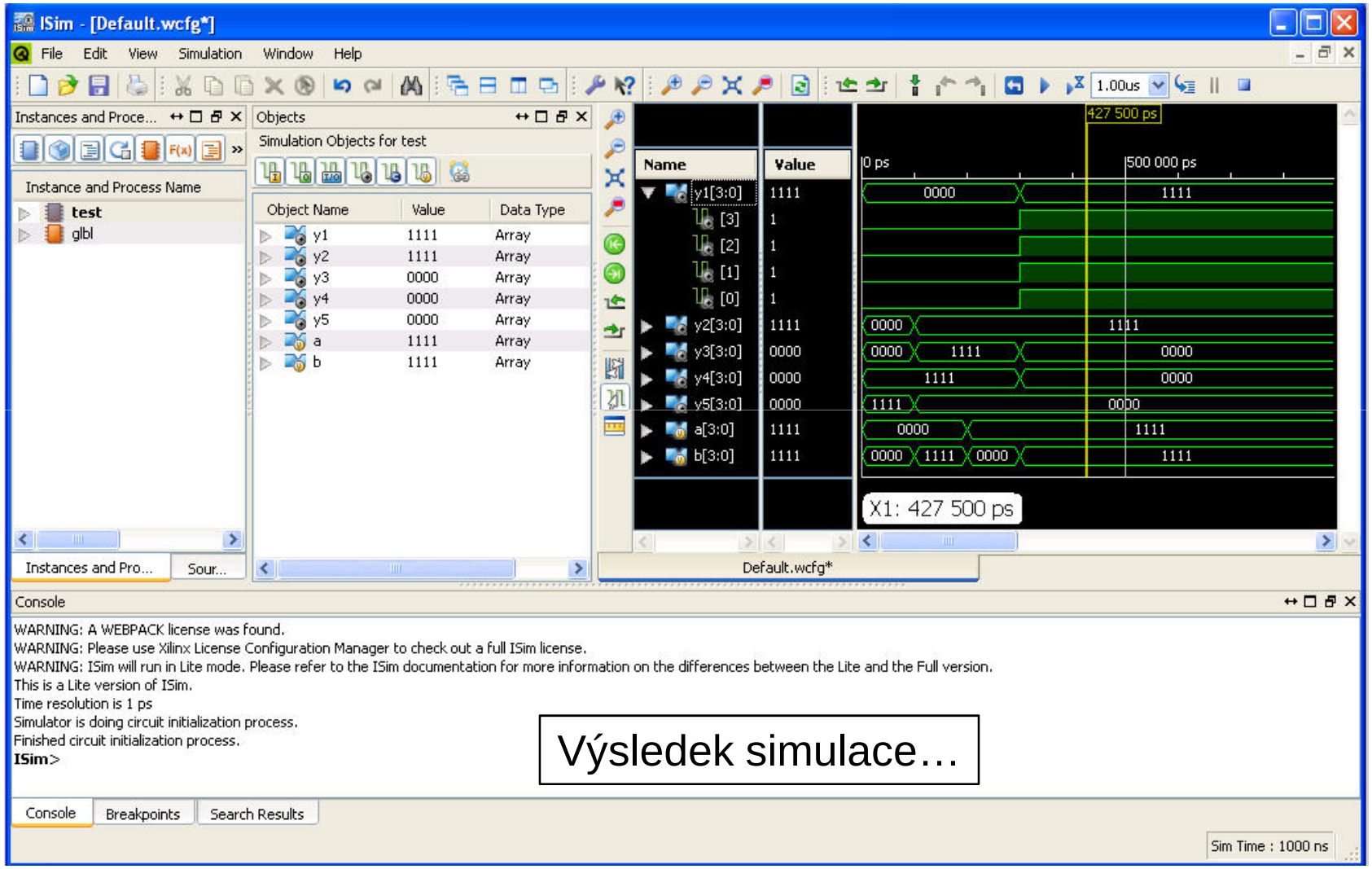
Dopsání vnějších stimulů obvodu (lze to i lépe.. Viz str.17):

```
a = 4'b0000;  // binárně
b = 4'b1111;

#100;// pozastavení na 100 jednotek
a = 15;  // dekadicky
b = 0;

#100;
a = 4'hF;  // hexadecimálně
b = 'hF;
```

Simulace tohoto obvodu v prostředí Xilinx IDE



Simulace tohoto obvodu v prostředí Xilinx IDE



Verilog – behaviorální popis – pokračování v úvodu do jazyka

Verilog operator precedence

	Op	Meaning
Highest	~	NOT
	*, /, %	MUL, DIV, MOD
	+, −	PLUS, MINUS
	<<, >>	Logical Left/Right Shift
	<<<, >>>	Arithmetic Left/Right Shift
	<, <=, >, >=	Relative Comparison
	==, !=	Equality Comparison
Lowest	&, ~&	AND, NAND
	^, ~^	XOR, XNOR
	|, ~|	OR, NOR
	?:	Conditional

Verilog – behaviorální popis

- ternární operátor „?“

```
module mux2 (input [3:0] d0, d1,
             input s,
             output [3:0] y);
      assign y = s ? d1 : d0;
endmodule
```

- redukce

```
module and5(input [4:0] a, output y);
    assign y = &a;
```

místo

```
    assign y = a[4] & a[3] & a[2] & a[1] & a[0];
```

podobně pro operátory: |, ^, ~&, a ~| (or, xor, nand, a nor)

Verilog – behaviorální popis

- čísla

```
N'ZakladHodnota  // N: počet bitů; Zaklad: b,o,d,h
b – binární, o – octal, d – decimal, h - hexa
3'b101 // 101
'b11  // při přiřazení do 6-bitové proměnné 000011
3'd5  // 101
```

- plovoucí hodnota „z“

```
module tristateBuffer (input [3:0] a,
                       input en,
                       output [3:0] y);
    assign y = en ? a : 4'bz;
endmodule
```

Verilog – behaviorální popis

- neplatná hodnota „x“

  „x“ vzniká když se snažíme přiřadit 0 a 1 najednou, při neinicializovaném výstupu, nebo pokud hradlo nedokáže určit svůj výstup (na vstupu „z“; 1 and „z“ = „x“ ale 0 and „z“ = 0)

- konkatenace (spájení) { }

  y = $c_2c_1d_0d_0d_0c_0101$ se získá pomocí:

```
assign y = {c[2:1],{3{d[0]}},c[0],3'b101};
```

  na = ~a; nb = ~b; nc = ~c; se získá pomocí:

```
{na, nb, nb} = ~{a, b, c};
```

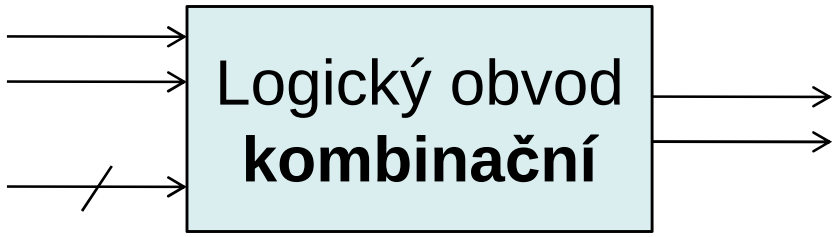
Verilog – behaviorální popis

- zpoždění

```
‘timescale 1ns/1ps              // jednotka/přesnost

module example (input a, b,
                output y);
    assign #3 y = a & b;        // 3 jednotky = 3 ns
endmodule
```

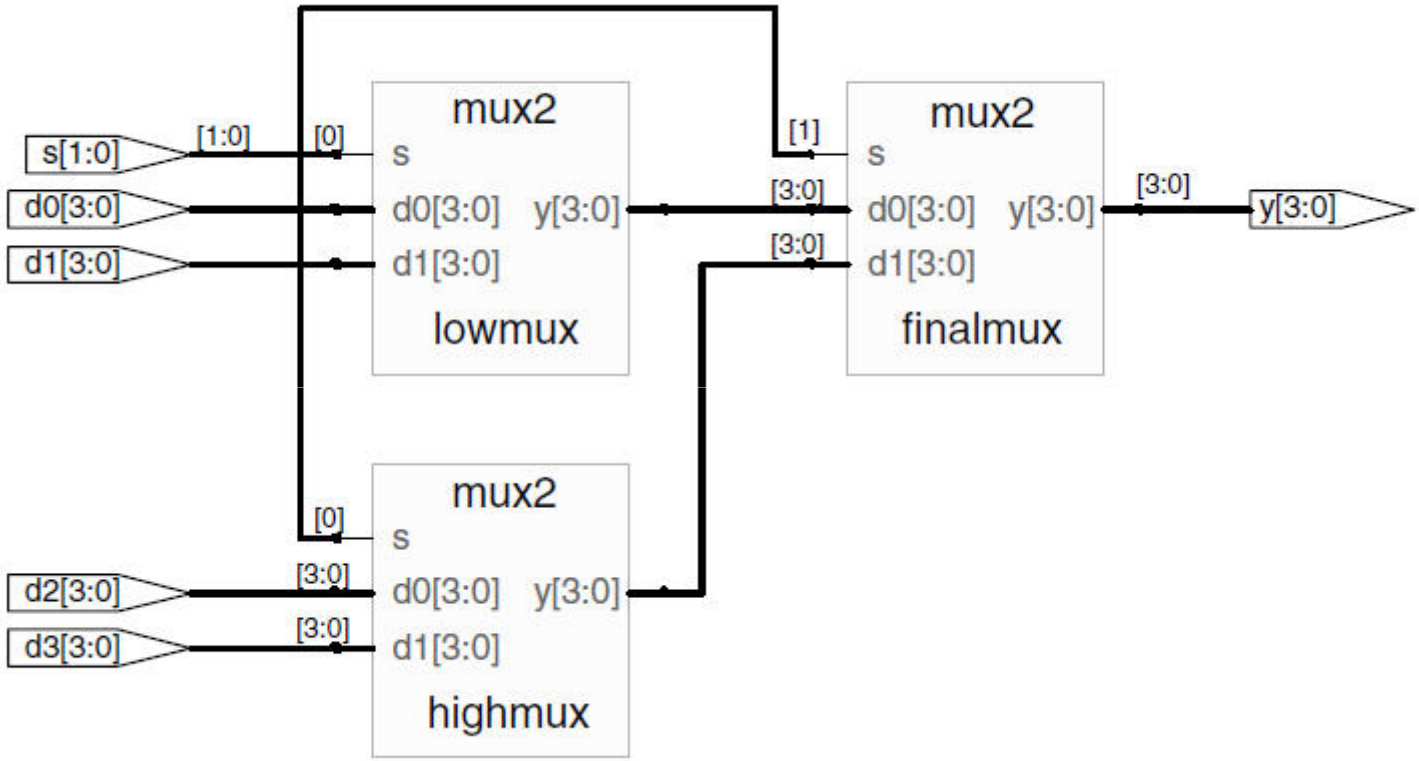
Zpoždění se používá pro účely simulace, při syntéze se ignoruje.

Verilog – strukturální popis




```
module název (input seznam_vstupů,
              output seznam_výstupů);

  wire deklarace_vnitřních_proměnných;

  název_modulu pojmenování(mapování_portů);
  název_modulu pojmenování(mapování_portů);
endmodule
```

Verilog - příklad



Verilog – strukturální popis

```
module mux4 (input [3:0] d0, d1, d2, d3,
             input [1:0] s,
             output [3:0] y);

  wire [3:0] low, high;

  mux2 lowmux (d0, d1, s[0], low);
  mux2 highmux (d2, d3, s[0], high);
  mux2 finalmux (low, high, s[1], y);

endmodule
```

Verilog – strukturální popis

```
module mux2 (input [3:0] d0, d1,
             input s,
             output [3:0] y);

   tristateBuffer t0 (d0, ~s, y);   // výraz ~s povolen
   tristateBuffer t1 (d1, s, y);

endmodule



module tristateBuffer (input [3:0] a,
                       input en,
                       output [3:0] y);
    assign y = en ? a : 4'bz;
endmodule
```

Verilog

- Přiřazení výstupu pomocí `assign` je vyhodnoceno vždy když se změní pravá strana výrazu
- Přiřazení výstupu pomocí `always @()` je vyhodnoceno jen za určitých podmínek (levá strana výrazu si může uchovat svou hodnotu i když se pravá strana mění)
- Blokující přiřazení „=“ jsou postupně vyhodnocovány v pořadí jak jsou zapsány
- Neblokující přiřazení „<=“ jsou vyhodnoceny současne; levá strana všech výrazů je aktualizována až po vyhodnocení pravých stran

Verilog – always

- Doporučení:
  Používat blokující přiřazení pro kombinační obvody, neblokující pro sekvenční.

- Přiřazení výstupu pomocí `always:`

```
always @ (sensitivity_list)
   statement;
```

  - `statement` je vyhodnocen jenom v případě když nastane událost uvedena v `sensitivity_list`
  - Pokud je uvedeno `always @(*)`, pak se reaguje na jakoukoliv změnu
  - Všechny signály na levé straně přiřazení (=, <=) musejí být deklarovány jako `reg`

Verilog – blokující vs. neblokující přiřazení

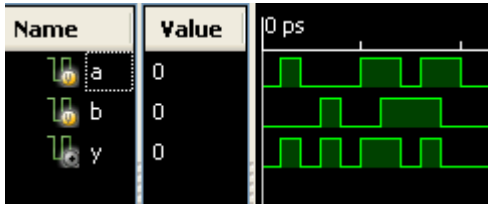
```
module obvod (input a, b,
              output reg y);
reg x;
always @(*)
  begin
    x = a ^ b; //blokující
    y = x & a; //blokující
  end
endmodule
```

```
module obvod (input a, b,
              output reg y);
reg x;
always @(*)
  begin
    x <= a ^ b; //neblokující
    y <= x & a; //neblokující
  end
endmodule
```

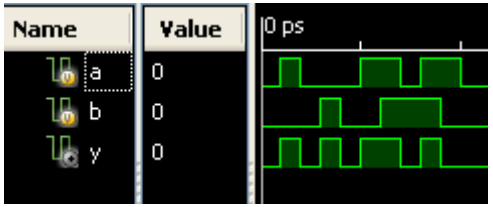
Uvažujme, že a=b=0. Pak y=0. V nějakém čase se změní a na 1 (a=1), čímž se spustí always blok. Blokující příkazy se vykonají v pořadí, v jakém jsou zapsány, čili x nadobude novou hodnotu před tím, než je vyhodnoceno y. V případě užití neblokujících přirazení, se všechny výrazy vyhodnotí najednou, až pak se aktualizují levé strany. To má za důsledek, že y zůstane rovno 0. Ovšem změna x z 0 na 1 má za důsledek opětovné spuštění always bloku, a následně správné vyhodnocení y. Kdybychom však psali výčet: always @(a,b) místo always @(*) jednalo by se o dva rozdílné obvody...

Verilog

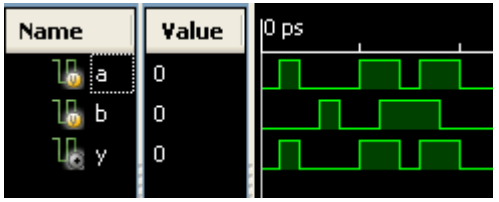
```
module abc(
    input a,
    input b,
    output reg y
    );

  always @(*)
    y = a;

  always @(*)
    y = b;

endmodule
```




```
module abc(
    input a,
    input b,
    output reg y
    );

  always @(*)
    y <= a;

  always @(*)
    y <= b;

endmodule
```




```
module abc(
    input a,
    input b,
    output reg y
    );

  always @(*)
    assign y = a;

  always @(*)
    y = b;

endmodule
```



Verilog

Deklarace proměnné jako:

- `reg  x;` znamená, že se jedná o 1-bitovou proměnnou (signál), která bude použita na levé straně přiřazení uvnitř `always`
- `reg [7:0] x;` znamená, že x je 8-bitová proměnná…
- `reg x[7:0];` znamená, že x je kolekce osmi jednobitových proměnných, ke kterým lze přistupovat přes index (x[0], x[1],…)
- `reg [7:0] x[15:0];` znamená…

Verilog – always

Klopný obvod D spouštěný náběžnou hranou clk (positive edge):

```
module D_flip_flop (input clk,
                    input [3:0] d,
                    output reg [3:0] q);
      always @ (posedge clk)
            q <= d;
endmodule
```

Asynchronně resetovatelný klopný obvod D:

```
module D_flip_flop_r (input clk, reset,
                      input [3:0] d,
                      output reg [3:0] q);
      always @ (posedge clk, posedge reset)
            if (reset) q <= 4'b0;
            else q <= d;
endmodule
```

Verilog – always

V případě, že `sensitivity_list` reaguje na změny na všech vstupech, a výstupní hodnota je předepsána pro všechny možné kombinace vstupů, pak `always @ ( )` může předepisovat chování kombinačního obvodu. Příklad:

```
module comb (input a, b
             output reg y);
   always @ (a,b)
      y  = ~a & b;
endmodule
```

lepší je však psát:

```
module comb (input a, b
             output reg y);
   always @ (*)                    // <--
      y  = ~a & b;
endmodule
```

Verilog – if-else

Konstrukce `if, if-else, case, casez` musejí být psány uvnitř `always`. Příklady:

```
module D_flip_flop_en_r (input clk, reset, enable,
                         input [3:0] d,
                         output reg [3:0] q);
      always @ (posedge clk, posedge reset)
            if (reset) q <= 4'b0;
            else if (en) q <= d;
endmodule
```

Verilog – case

```
module seven_segment_display_decoder (input [3:0] data,
                                      output reg [6:0] segments);
   always @ (*)
        case (data)
          0: segments = 7'b111_1110;
          1: segments = 7'b011_0000;
          2: segments = 7'b110_1101;
          3: segments = 7'b111_1001;
          4: segments = 7'b011_0011;
          5: segments = 7'b101_1011;
          6: segments = 7'b101_1111;
          7: segments = 7'b111_0000;
          8: segments = 7'b111_1111;
          9: segments = 7'b111_1011;
          default: segments  7'b000_0000;
        endcase
endmodule
```

Verilog – casez

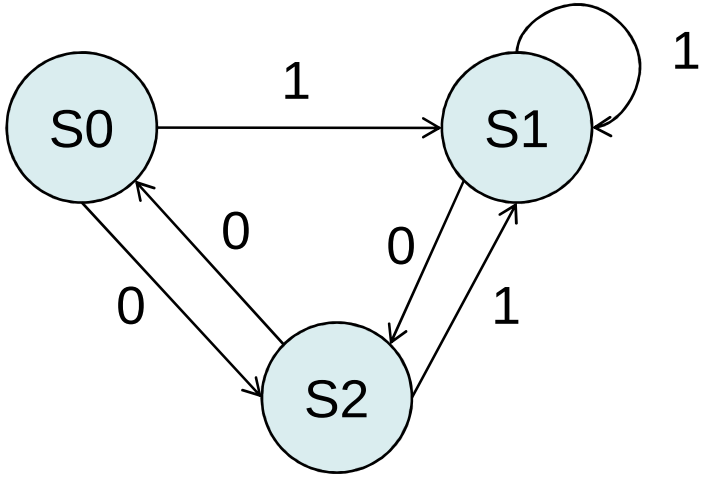
casez: V místě ? může být 0 nebo 1, tj. na hodnotě nezáleží.

```
module priority_circuit(input [3:0] a,
                        output reg [3:0] y);
    always @ (*)
        casez (a)
          4'b1???: y = 4'b1000;
          4'b01??: y = 4'b0100;
          4'b001?: y = 4'b0010;
          4'b0001: y = 4'b0001;
          default: y = 4'b0000;
        endcase
endmodule
```

Verilog – konečný automat Mealy/Moore

```
module FSM (input x, clk, reset,
            output y);
   reg [1:0] state, nextstate;
   parameter S0 = 2'b00;     // parameter – definice konstanty
   parameter S1 = 2'b01;
   parameter S2 = 2'b10;

   always @ (posedge clk, posedge reset)
        if (reset) state <= S0;
        else state <= nextstate;

   always @ (*)
        case (state)                    // next state logic
               S0: if (x) nextstate = S1;
                   else   nextstate = S0;
               S1: nextstate = S2;
               S2: nextstate = S0;
               default: nextstate = S0;
        endcase
   assign y = (x & state == S0 ) ? 1'b1 : 1'b0;   // output logic
endmodule
```

Verilog – **jiný** konečný automat Moore



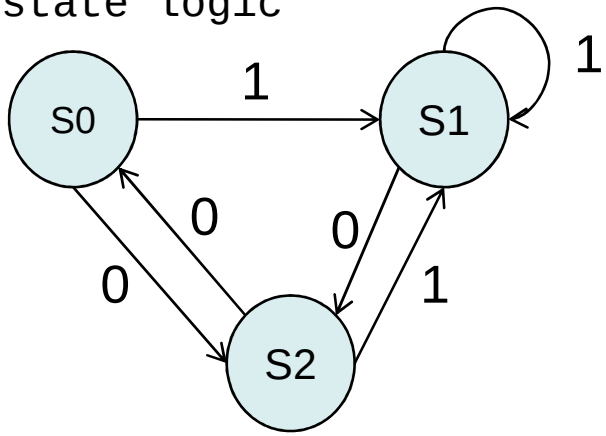
Tabulka výstupů

Stav	Výstup y
S0	0010
S1	0110
S2	1010

```
module FSM2 (input x, clk, reset,
             output reg [3:0] y);

   reg [1:0] state, nextstate;
   parameter S0 = 2'b00;
   parameter S1 = 2'b01;
   parameter S2 = 2'b10;

   always @(posedge clk, posedge reset)   // state register
        if (reset) state <= S0;
        else state <= nextstate;
```

Verilog – konečný automat Moore

```
    always @(*)
        case (state)                        // next state logic
                S0: nextstate = x ? S1 : S2;
                S1: nextstate = x ? S1 : S2;
                S2: nextstate = x ? S1 : S0;
                default: nextstate = S0; //!!
        endcase
```




```
    always @(*)
        case (state)                         // output logic
                S0: y = 4'b0010;
                S1: y = 4'b0110;
                S2: y = 4'b1010;
                default: y = 4'b0010;  //!!
        endcase

endmodule
```

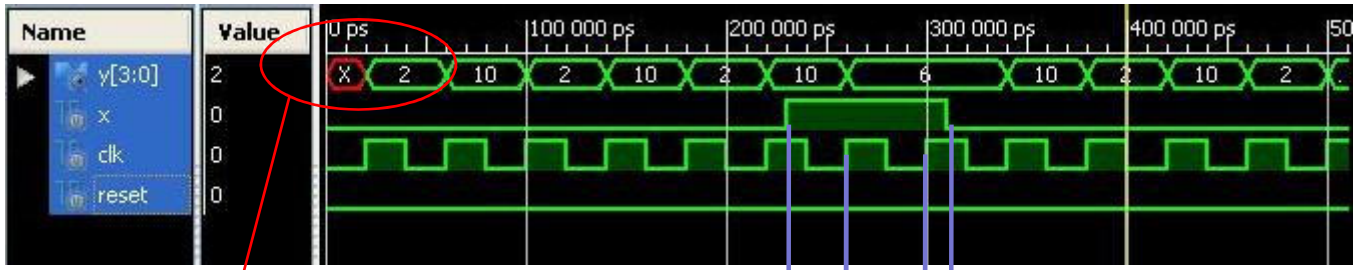
Stav	Výstup y
S0	0010
S1	0110
S2	1010

Verilog – konečný automat Moore - Simulace

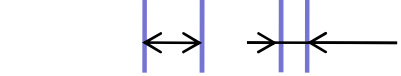
```
`timescale 1ns / 1ps

module test_moore;
   reg x;          // Inputs
   reg clk;
   reg reset;
   wire [3:0] y;  // Output

   FSM2 uut (
        .x(x),
        .clk(clk),
        .reset(reset),
        .y(y)
   );
```

```
initial begin
     x = 0;
     clk = 0;
     reset = 0;

     #230 x = 1;
     #80 x = 0;
     #100;
   end

   always #20 clk = ~clk;

endmodule
```

Verilog – konečný automat Moore

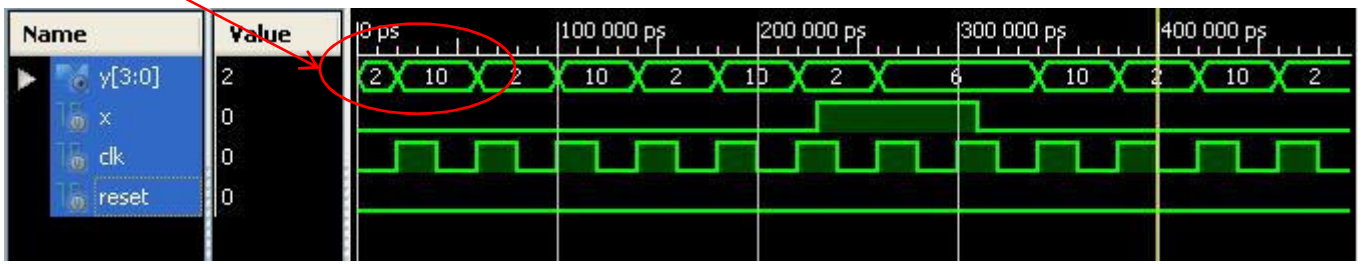



```
module FSM3 (input x, clk, reset,
             output reg [3:0] y);

   reg [1:0] state, nextstate;
   parameter S0 = 2'b00;
   parameter S1 = 2'b01;
   parameter S2 = 2'b10;

   initial  state <= S0;
```



předstih a přesah !!

Modul FSM3 se od FSM2 liší jenom dopsáním tohoto řádku...



Verilog – parametrizované moduly

(parameter …) hned po názvu modulu

```
module mux2
          # (parameter width = 8)
          (input [width:0] d0, d1,
           input s,
           output [width:0] y);
  assign y = s ? d1 : d0;
endmodule
```

Verilog – parametrizované moduly

Pak je možné psát:

```
module mux4_12(input [11:0] d0, d1, d2, d3,
               input [1:0] s,
               output [11:0] y);

  wire [7:0] low, hi;

  mux2 #(12) lowmux(d0,d1,s[0],low); //přepsání 8 na 12
  mux2 himux (d2,d3,s[1],hi); //zde zůstává width=8; chyba!
  mux2 #(12) outmux (low, hi, s[1], y);
endmodule
```

Lepší je však i nadále vytvářet parametrizovaný modul..

Pozor: #(…) znamená přepsání parametru; #... zase zpoždění.

Verilog – závěrečné poznámky

- Jazyk Verilog slouží na popis hardvéru. Při návrhu hardvéru se využívá zejména princip „shora dolů“, při kterém se jednotlivé subsystémy postupně zpřesňují, přičemž funkčnost systému jako celku může být simulována již v návrhové etapě bez konkrétní specifikace dílčích subsystémů na nejnižší úrovni. Stejný HDL kód popisující behaviorální chování obvodu může být různě implementován v závislosti na použitém syntetizačním nástroji, jeho nastavení a míře optimalizace. Příkladem budiž:

```
module adder (input [7:0] a, b,
              output [7:0] y);
    assign y = a + b;   // jak bude součet implementován?
endmodule
```

Kontrolní otázka č.1 – Co je špatně???

Klopný obvod D spouštěný náběžnou hranou clk (positive edge):

```
module D_flip_flop (input clk,
                    input d,
                    output q);
      always @ (posedge clk)
            q <= d;
  endmodule
```

Kontrolní otázka č.1 – Co je špatně???

Klopný obvod D spouštěný náběžnou hranou clk (positive edge):

```
module D_flip_flop (input clk,
                    input d,
                    output q);
      always @ (posedge clk)
            q <= d;
  endmodule
```

Všechny signály (použité uvnitř always) na levé straně přiřazení (=, <=) musejí být deklarovány jako `reg`

Kontrolní otázka č.2 – Co je špatně???

Multiplexor

```
module mux2 (input select, d0, d1,
             output reg y);
    always @ (select)
      if(select) y <= d1;
      else y <= d0;
endmodule
```

Kontrolní otázka č.2 – Co je špatně???

Multiplexor

```
module mux2 (input select, d0, d1,
             output reg y);
    always @ (select)
      if(select) y <= d1;
      else y <= d0;
endmodule
```

Hodnota y se nezmění při změně dat.. Řešení:
always @(*)
nebo ternární operátor: `assign y = select ? d1 : d0;`
a nepoužít konstrukci always

Literatura:

- Harris D.M. – Harris S.L.: Digital design and computer architecture, ed. Morgan–Kaufmann, 2007, chapter 4: Hardware description languages

Zdroje na internetu:

- http://www.aldec.com/Products/Product.aspx?productid=0380ca74-7c15-4a01-b727-2f7caab53730
- http://vol.verilog.com/
- http://www.verilogtutorial.info/
- http://courses.cit.cornell.edu/ece576/Verilog/LatticeTestbenchPrimer.pdf