Laboratorijske vježbe 2: Uvod u Verilog i Questa TestBench

Predmet: Projektovanje sistema na cipu

1 Uvod u verilog

1.1 Hardware Description Language (HDL)?

Hardware Description Language (HDL) je specijalizovani jezik za opisivanje strukture i ponašanja elektronskih kola, posebno digitalnih sistema. Za razliku od tradicionalnih programskih jezika koji opisuju sekvencu instrukcija koje procesor izvršava jednu po jednu, HDL opisuje strukturu i ponašanje hardverskih komponenti koje se izvršavaju paralelno.

Ključna razlika

Softverski jezici (C, Python, Java): Sekvencijalno izvršavanje instrukcija
Hardverski jezici (Verilog, VHDL): Paralelni opis hardverskih komponenti

1.2 Verilog — primjena

Koristi se u industriji za:

- **Dizajn integrisanih kola** — ASIC i FPGA
- **Verifikaciju** — Provjera funkcionalnosti prije implementacije
- **Simulaciju** — Testiranje ponašanja kola
- **Sintezu** — Automatska konverzija Verilog koda u fizičke logičke kapije

1.3 Nivoi apstrakcije u Verilogu

Verilog omogućava opis sistema na različitim nivoima:

1. **Behavioralni nivo** — Funkcionalni opis bez detalja implementacije
2. **RTL nivo (Register Transfer Level)** — Opis transfera podataka između registara
3. **Gate nivo** — Direktno instanciranje logičkih kapija
4. **Switch nivo** — Najniži nivo, opis tranzistora

Mi ćemo se fokusirati na **RTL nivo**, koji je najčešće korišten u praksi.

1.4 Konkurentno vs sekvencijalno izvršavanje

U Verilogu:

- Svi `assign` izrazi izvršavaju se paralelno
- Svi `always` blokovi izvršavaju se paralelno
- Unutar `always` bloka, izrazi se izvršavaju sekvencijalno

```
// Ova tri assign-a izvrsavaju se ISTOVREMENO
assign out1 = a & b;
assign out2 = c | d;
assign out3 = e ^ f;
```

2 Osnovni elementi Verilog jezika

2.1 Moduli — Osnovna gradivna jedinica

Modul je fundamentalna struktura u Verilogu. Svaki Verilog dizajn se sastoji od jednog ili više modula. Modul je analogan funkciji u programskim jezicima, ali sa ključnom razlikom — on predstavlja fizički hardver.
Naziv modula mora biti isti kao naziv fajla.

2.1.1 Struktura modula

```
module ime_modula (
    // Lista portova - ulazi i izlazi
    input port1,
    input port2,
    output port3
);

    // Interni signali (wire ili reg)
    wire interni_signal;

    // Logika modula
    assign port3 = port1 & port2;

endmodule
```

2.1.2 Portovi modula

Portovi su interfejs modula prema van. Mogu biti:

- **input** — Ulazni signal (čitanje)
- **output** — Izlazni signal (pisanje)
- **inout** — Dvosmjerni signal (bidirekcioni)

```
module and_gate(
    input wire a,       // Prvi ulaz
    input wire b,       // Drugi ulaz
    output wire y       // Izlaz
);

    assign y = a & b;

endmodule
```

2.1.3 Višebitni portovi (vektori)

```
module adder_4bit(
    input wire [3:0] a,       // 4-bitni ulaz A
    input wire [3:0] b,       // 4-bitni ulaz B
    input wire cin,           // Carry in
    output wire [3:0] sum,    // 4-bitni zbir
    output wire cout          // Carry out
);

    assign {cout, sum} = a + b + cin;

endmodule
```

*

Notacija `[3:0]` označava 4-bitni signal gdje je bit 3 MSB (Most Significant Bit), a bit 0 LSB (Least Significant Bit). Moguće je koristiti i `[0:3]`, ali je konvencija `[N:0]`.

2.2 Tipovi podataka: wire i reg

2.2.1 Wire — Kombinacijski signal

wire predstavlja fizičku vezu (žicu). Vrijednost wire signala je uvijek određena izrazom koji ga definiše. Ne može pamtiti prethodnu vrijednost.

```
module wire_example(
    input wire a,
    input wire b,
    output wire y
);

    wire temp;
    assign temp = a & b;
    assign y = ~temp;

endmodule
```

Ključne karakteristike wire:

- Koristi se za kombinacijsku logiku
- Mora biti povezan sa `assign` ili izlazom drugog modula

- Ne može se dodijeliti unutar `always` bloka
- Promjena ulaza odmah se reflektuje na izlazu

2.2.2 Reg — Promjenjiva vrijednost

reg može pamtiti vrijednost. Iako ime sugeriše registar, to ne mora uvijek biti flip-flop — zavisi od konteksta u kojem se koristi.

```
module reg_example(
    input wire clk,
    input wire d,
    output reg q
);
    always @(posedge clk) begin
        q <= d;
    end
endmodule
```

Ključne karakteristike reg:

- Koristi se unutar `always` blokova
- Može predstavljati flip-flop, latch, ili kombinacijsku logiku
- Zadržava vrijednost dok se eksplicitno ne promijeni
- Ne može se koristiti sa `assign`

2.3 Assign — Kontinuirana dodjela

`assign` definiše stalnu vezu između signala i koristi se samo izvan `always` blokova.

```
module assign_examples(
    input wire [3:0] a,
    input wire [3:0] b,
    input wire sel,
    output wire [3:0] out1,
    output wire [3:0] out2,
    output wire [3:0] out3,
    output wire [3:0] mux_out
);

    assign out1 = a & b;
    assign out2 = a | b;
    assign out3 = a ^ b;
    assign mux_out = sel ? a : b;

endmodule
```

2.3.1 Operatori u Verilogu

```
module bit_operations(
    input wire [7:0] data_in,
    output wire [7:0] data_out,
    output wire [15:0] extended
```

Tip	Operator	Opis
Logički	`&&`, `||`, `!`	AND, OR, NOT (1-bitni rezultat)
Bitwise	`&`, `|`, `^`, `~`	Bit-po-bit operacije
Aritmetički	`+`, `-`, `*`, `/`	Sabiranje, oduzimanje, množenje, dijeljenje
Relacioni	`==`, `!=`, `<`, `>`	Poređenje
Shift	`«`, `»`	Pomjeranje bitova
Uslovni	`?  :`	Ternarni operator
Konkatenacija	`{}`	Spajanje bitova

Table 1: Pregled operatora u Verilogu

```
);

    wire msb = data_in[7];
    wire lsb = data_in[0];
    wire [3:0] upper_nibble = data_in[7:4];
    wire [3:0] lower_nibble = data_in[3:0];
    assign data_out = {lower_nibble, upper_nibble};
    assign extended = {8'b0, data_in};

endmodule
```

2.4 Always blokovi

`always` blok je proceduralni blok koji se izvršava kad se promijeni signal u sensitivity listi.

2.4.1 Kombinatorna logika sa always @(*)

```
module simple_alu(
    input wire [3:0] a,
    input wire [3:0] b,
    input wire [1:0] op,
    output reg [3:0] result
);
    always @(*) begin
        case(op)
            2'b00: result = a + b;
            2'b01: result = a - b;
            2'b10: result = a & b;
            2'b11: result = a | b;
            default: result = 4'b0000;
        endcase
    end
endmodule
```

2.4.2 Sekvencijalna logika sa always @(posedge clk)

```
module dff_async_reset(
    input wire clk,
    input wire rst_n,
    input wire d,
    output reg q
```

```
);
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n)
            q <= 1'b0;
        else
            q <= d;
    end
endmodule
```

```
module counter_4bit(
    input wire clk,
    input wire rst,
    input wire enable,
    output reg [3:0] count
);
    always @(posedge clk) begin
        if (rst)
            count <= 4'b0000;
        else if (enable)
            count <= count + 1'b1;
    end
endmodule
```

2.4.3 Blocking vs. Non-blocking dodjele

*

- **Blocking dodjela** (=) — Koristi se u kombinacijskoj logici
- **Non-blocking dodjela** (<=) — Koristi se u sekvencijalnoj logici

```
always @(*) begin
    temp = a & b;
    out = temp | c;
end
```

```
always @(posedge clk) begin
    q1 <= d;
    q2 <= q1;
end
```

```
always @(posedge clk) begin
    q1 = d;
    q2 <= q1;
end
```

2.5 Case i If-Else strukture

```
module seven_segment_decoder(
    input wire [3:0] digit,
    output reg [6:0] segments
);
    always @(*) begin
```

```
        case(digit)
            4'd0: segments = 7'b1111110;
            4'd1: segments = 7'b0110000;
            4'd2: segments = 7'b1101101;
            4'd3: segments = 7'b1111001;
            4'd4: segments = 7'b0110011;
            4'd5: segments = 7'b1011011;
            4'd6: segments = 7'b1011111;
            4'd7: segments = 7'b1110000;
            4'd8: segments = 7'b1111111;
            4'd9: segments = 7'b1111011;
            default: segments = 7'b0000000;
        endcase
    end
endmodule
```

```
module priority_encoder(
    input wire [7:0] req,
    output reg [2:0] grant,
    output reg valid
);
    always @(*) begin
        if (req[7]) begin
            grant = 3'd7; valid = 1'b1;
        end
        else if (req[6]) begin
            grant = 3'd6; valid = 1'b1;
        end
        else if (req[5]) begin
            grant = 3'd5; valid = 1'b1;
        end
        else if (req[0]) begin
            grant = 3'd0; valid = 1'b1;
        end
        else begin
            grant = 3'd0; valid = 1'b0;
        end
    end
endmodule
```

3 Primjeri

3.1 Half Adder

```
module half_adder(
    input wire a,
    input wire b,
    output wire sum,
    output wire carry
);
    assign sum = a ^ b;
    assign carry = a & b;
endmodule
```

3.2 2:1 Multiplexer

```
module mux2to1(
    input wire a,
    input wire b,
    input wire sel,
    output reg y
);
    always @(*) begin
        if (sel)
            y = a;
        else
            y = b;
    end
endmodule
```

3.3 Zadaci za vježbu

1. Napisati 4u1 multiplexer
2. Napišite brojač koji broji do 9 nakon cega se resetuje.

4 ModelSim i Testbenchovi

4.1 Uvod u ModelSim

ModelSim je program razvijen od strane **Mentor Graphics** i koristi se za simulaciju **VHDL** i **Verilog** dizajna. Može se instalirati neovisno o Quartus paketu i koristi se za testiranje HDL koda bez potrebe za fizičkim FPGA čipom.

Simulacija je ključni korak u dizajnu digitalnih sistema jer omogućava provjeru ispravnosti koda prije implementacije. Testiranjem kroz ModelSim možemo provjeriti reakciju dizajna na različite ulazne signale i uočiti greške koje bi u stvarnom hardveru bile teže za otkriti.

4.2 Šta je testbench

Testbench je poseban Verilog fajl koji se koristi za provjeru funkcionalnosti dizajna. On ne predstavlja fizički dio kola — ne sadrži ulazne ni izlazne portove — već instancira glavni modul (tzv. *Device Under Test* ili *DUT*) i stimuliše ga različitim vrijednostima signala.

Prednosti simulacije pomoću testbencha:

- Nije potreban stvarni hardver.
- Može se pregledati svaki signal unutar dizajna.
- Moguće je testirati sva stanja i scenarije.

4.3 Primjer jednostavnog Verilog modula

Listing 1: basic_and.v — jednostavno AND kolo

```
module basic_and (
    input wire a,       // prvi ulaz
    input wire b,       // drugi ulaz
    output wire y       // izlaz
);
    assign y = a & b;  // logi?ka AND operacija
endmodule
```

Ovaj modul implementira višebitno AND kolo koje spaja dva ulaza `a` i `b` i rezultat postavlja na `out`.

4.4 Testbench za AND kolo

Listing 2: basic_and_tb.v — testbench za AND kolo

```
`timescale 1ns/1ps

// Testbench nema ulazne/izlazne portove
module basic_and_tb;
    reg a;        // definisanje promenljivih koje ?aljemo u DUT
    reg b;
    wire y;       // signal koji dolazi iz DUT (Device Under Test)
```

```
    // Instanciranje testiranog modula
    basic_and uut (
        .a(a),
        .b(b),
        .y(y)
    );

    // Glavni testni proces
    initial begin
        $display("Time\tA\tB\tY");
        $monitor("%0t\t%b\t%b\t%b", $time, a, b, y);

        // Inicijalne vrijednosti
        a = 0; b = 0; #10;    // ocekujemo y=0
        a = 0; b = 1; #10;    // ocekujemo y=0
        a = 1; b = 0; #10;    // oekujemo y=0
        a = 1; b = 1; #10;    // ocekujemo y=1

        $finish;  // zavrsetak simulacije
    end
endmodule
```

Ovdje vidimo upotrebu `initial` bloka i `#delay` izraza:

- `initial` se izvršava samo jednom — na početku simulacije.
- #20 označava vremenski pomak od 20 jedinica simulacionog vremena.

4.5 Pokretanje simulacije u ModelSim-u

1. Pokrenite ModelSim i kreirajte novi projekat:
   - `File → New → Project`
   - Izaberite ime i lokaciju bez specijalnih karaktera (č, ć, š, ž).
2. Dodajte fajlove `basic_and.v` i `basic_and_tb.v`.
3. U komandnoj liniji (Transcript) unesite:

```
vlog basic_and.v basic_and_tb.v
vsim work.basic_and_tb
add wave *
run 100ns
```

4. U Wave prozoru analizirajte signale `a`, `b` i `out`.

5 Sekvencijalni testbenchovi — primjer PWM modula

Za razliku od kombinacionih kola, sekvencijalna kola (poput PWM generatora, brojača ili registra) zahtijevaju testbench koji simulira **vremensku sekvencu signala**. Potrebno je generisati **clock** i **reset** signale te u odgovarajućim trenucima mijenjati ulazne vrijednosti.

5.1 PWM modul

U nastavku je prikazan jednostavan PWM (Pulse Width Modulation) modul, čiji se izlaz mijenja u zavisnosti od vrijednosti ulaza `compare`.

Listing 3: pwm.v — sekvencijalni PWM modul

```
module pwm #(parameter CTR_LEN = 8) (
    input clk,
    input rst,
    input [CTR_LEN - 1 : 0] compare,
    output pwm
);

  reg pwm_d, pwm_q;
  reg [CTR_LEN - 1: 0] ctr_d, ctr_q;

  assign pwm = pwm_q;

  always @(*) begin
    ctr_d = ctr_q + 1'b1;

    if (compare > ctr_q)
      pwm_d = 1'b1;
    else
      pwm_d = 1'b0;
  end

  always @(posedge clk) begin
    if (rst) begin
      ctr_q <= 1'b0;
    end else begin
      ctr_q <= ctr_d;
    end

    pwm_q <= pwm_d;
  end

endmodule
```

5.2 Testbench za PWM modul

Listing 4: pwm_tb.v — sekvencijalni testbench za PWM

```
module pwm_tb ();

  reg clk, rst;
  reg [7:0] compare;
  wire pwm;

  pwm #(.CTR_LEN(8)) DUT (
    .clk(clk),
    .rst(rst),
    .compare(compare),
    .pwm(pwm)
  );
```

```
  // Generisanje clock i reset signala
  initial begin
    clk = 1'b0;
    rst = 1'b1;
    repeat(4) #10 clk = ~clk;
    rst = 1'b0;
    forever #10 clk = ~clk; // generate a clock
  end

  // Glavni testni proces
  initial begin
    compare = 8'd0; // initial value
    @(negedge rst); // wait for reset
    compare = 8'd128;
    repeat(256) @(posedge clk);
    compare = 8'd30;
    repeat(256) @(posedge clk);
    $finish;
  end

endmodule
```

5.3 Objašnjenje testbencha

Primijetimo da testbench **nema ulaze ni izlaze**. On instancira DUT (Device Under Test) i sadrži dva `initial` bloka.

1. Blok — generisanje clock i reset signala

```
initial begin
  clk = 1'b0;
  rst = 1'b1;
  repeat(4) #10 clk = ~clk;
  rst = 1'b0;
  forever #10 clk = ~clk; // generate a clock
end
```

Ovaj blok kreira reset i clock signale. Ovakav `initial` blok se koristi za **svako sekvencijalno kolo**. Signali su inicijalizirani na 0 i 1, a naredba `repeat(4) #10 clk = clk;` mijenja clock četiri puta s 10 vremenskih jedinica između promjena.

Nakon što se nekoliko puta toggla clock, reset se postavlja na 0 kako bi se kolo pokrenulo. Reset je neophodan jer flip-flopovi nemaju početne vrijednosti i bez resetovanja daju `x` u simulaciji.

Posljednja linija — `forever #10 clk = clk;` — stvara beskonačnu petlju koja proizvodi clock signal do kraja simulacije.

2. Blok — testiranje funkcionalnosti

```
initial begin
  compare = 8'd0; // initial value
  @(negedge rst); // wait for reset
  compare = 8'd128;
  repeat(256) @(posedge clk);
  compare = 8'd30;
  repeat(256) @(posedge clk);
  $finish;
end
```

Ovdje koristimo `@(negedge rst);` da čekamo negativnu ivicu reseta — simulacija se ne nastavlja dok reset ne bude deaktiviran. Signal `compare` se postavlja prvo na 128, čime dobijamo **50% duty cycle**. Budući da PWM brojač ima širinu od 8 bita, jedan puni ciklus PWM-a traje 256 ciklusa clocka, pa zato koristimo `repeat(256) @(posedge clk);`.

Zatim se testira nova vrijednost (`compare = 30`) i ponovo čeka jedan PWM ciklus prije završetka simulacije.

Dobar testbench bi trebao testirati i dodatne vrijednosti, naročito granične slučajeve (`compare = 0` i `compare = 255`).

5.4 Rezultat simulacije

U ModelSim-u, nakon kompajliranja:

```
vlog pwm.v pwm_tb.v
vsim work.pwm_tb
add wave *
run 5000ns
```

Vidjet ćemo signal `pwm` koji mijenja svoj duty cycle proporcionalno vrijednosti `compare` — veća vrijednost daje duži HIGH impuls, a manja kraći.

6 Primjeri digitalnih kola u Verilogu

6.1 4-u-1 multiplekser

4-u-1 multiplekser ima četiri ulaza i jedan izlaz, a dva selekcijska bita određuju koji se ulaz prosljeđuje na izlaz.

Listing 5: Kod 4-u-1 multipleksera

```
module mux4to1 (
    input wire [3:0] d,
    input wire [1:0] sel,
    output wire y
);
    assign y = (sel == 2'b00) ? d[0] :
               (sel == 2'b01) ? d[1] :
               (sel == 2'b10) ? d[2] :
                                d[3];
endmodule
```

Testbench:

Listing 6: Testbench za 4-u-1 multiplekser

```
`timescale 1ns/1ps

module mux4to1_tb;
    reg [3:0] d;
    reg [1:0] sel;
    wire y;

    mux4to1 uut (
        .d(d),
        .sel(sel),
        .y(y)
    );

    initial begin
        $display("Time\tSel\tD\tY");
        $monitor("%0t\t%b\t%b\t%b", $time, sel, d, y);

        d = 4'b1010;
        sel = 2'b00; #10;
        sel = 2'b01; #10;
        sel = 2'b10; #10;
        sel = 2'b11; #10;
        $finish;
    end
endmodule
```

6.2 Jednobitni sabirač (Full Adder)

Jednobitni sabirač ima tri ulaza: A, B i ulazni prenos C_{in}, te dva izlaza: zbir S i izlazni prenos C_{out}.

Listing 7: Kod jednobitnog sabirača

```
module full_adder (
```

```
    input wire a, b, cin,
    output wire sum, cout
);
    assign sum  = a ^ b ^ cin;
    assign cout = (a & b) | (b & cin) | (a & cin);
endmodule
```

Testbench:

Listing 8: Testbench za full adder

```
`timescale 1ns/1ps

module full_adder_tb;
    reg a, b, cin;
    wire sum, cout;

    full_adder uut (
        .a(a),
        .b(b),
        .cin(cin),
        .sum(sum),
        .cout(cout)
    );

    integer i;

    initial begin
        $display("Time\tA\tB\tCin\tSum\tCout");
        $monitor("%0t\t%b\t%b\t%b\t%b\t%b", $time, a, b, cin, sum, cout)
            ;

        for (i = 0; i < 8; i = i + 1) begin
            {a, b, cin} = i[2:0];
            #10;
        end

        $finish;
    end
endmodule
```