Verilog By Example
Table of Contents:

```
////////////////////////////////////////////////
//
// Header information
// Bus Breakout
//
////////////////////////////////////////////////

module bus_breakout
            (
               // Inputs
               in_1,
               in_2,
               // Outputs
               out_1
            );

    // Port definitions
    input  [3:0]   in_1;
    input  [3:0]   in_2;
    output [5:0]   out_1;

    // ------------- Design implementation --------

    assign out_1 = { in_2[3:2],
                     (in_1[3] & in_2[1]),
                     (in_1[2] & in_2[0]),
                     in_1[1:0]
                   };

endmodule
```

```
////////////////////////////////////////////////
//
// Header information
// Bus Signals
//
////////////////////////////////////////////////

module bus_sigs
            (
               // Inputs
               in_1,
               in_2,
               in_3,
               // Outputs
               out_1
            );

    // Port definitions

    input  [3:0]   in_1;
    input  [3:0]   in_2;
    input          in_3;

    output [3:0]   out_1;

    wire   [3:0]   in_3_bus;

    // ------------- Design implementation --------

    assign in_3_bus = {4{in_3}};
    assign out_1 = (~in_3_bus & in_1) | (in_3_bus & in_2);

endmodule
```

```
////////////////////////////////////////
// Clock Buffer

module clock_buffer
         ( reset,
           clk_in,
           dat_in,
           dat_out
         );

    input          reset;
    input          clk_;
    input          dat_in;
    output         dat_out;

    wire        clk;
    reg         dat_out;

    // ------ Design implementation -----

    // clock buffer instantiation
    BUFG clock_buffer_inst
         (
           .I   ( clk_in ),
           .O   ( clk    )
         );

    // register
    always @( posedge clk or posedge reset )
      begin
        if ( reset )
          dat_out <= 1'b0;
        else
          dat_out <= dat_in;
      end

endmodule
```

```
////////////////////////////////////////////
// D-flop with enable and clear
////////////////////////////////////////////

module dflop_en_clr  (  clk,
                        reset,
                        in_1,
                        enable,
                        clear
                        out_1
                      );

    input           clk;
    input           reset;
    input           in_1;
    input           enable;
    input           clear;
    output          out_1;

    reg             out_1;

    // ----- Design implementation -------

    always @( posedge clk or posedge reset )
      begin
        if ( reset )
          out_1 <= 1'b0;
        else if ( clear == 1'b0 )
          out_1 <= 1'b0;
        else if ( enable )
          out_1 <= in_1;
      end

endmodule
```

```
////////////////////////////////////////////////
// D-flop with reset
////////////////////////////////////////////////

module dflop_n_reset  (  clk,
                         reset,
                         in_1,
                         out_1
                       );

    input          clk;
    input          reset;
    input          in_1;
    output         out_1;

    reg            out_1;

    // ------------- Design implementation --------

    always @( posedge clk or posedge reset )
      begin
        if ( reset )
          out_1 <= 1'b0;
        else
          out_1 <= in_1;
      end

endmodule
```

```
////////////////////////////////////////
// Full Dual Port Memory

module full_dp_mem
         ( reset,
           // port a
           clk_a,
           dat_in_a,
           address_a,
           dat_out_a,
           wr_a,
           //port b
           clk_b,
           dat_in_b,
           address_b,
           dat_out_b,
           wr_b
         );

    input          reset;
    input          clk_a;
    input  [15:0]  dat_in_a;
    input  [9:0]   address_a;
    output [15:0]  dat_out_a;
    input          wr_a;
    input          clk_b;
    input  [15:0]  dat_in_b;
    input  [9:0]   address_b;
    output [15:0]  dat_out_b;
    input          wr_b;

    reg [15:0]     memory[0:1023];
    reg [15:0]     dat_out_a;
    reg [15:0]     dat_out_b;

    // ------ Design implementation -----

    // Port a
    //
    always @( posedge clk_a )
      begin
        dat_out_a <= memory[address_a];
        if (wr_a)
          begin
            dat_out_a         <= dat_in_a;
            memory[address_a] <= dat_in_a;
          end
      end

    // Port b
    //
    always @( posedge clk_b )
      begin
        dat_out_b <= memory[address_b];
```

```
        if (wr_b)
          begin
            dat_out_b         <= dat_in_b;
            memory[address_b] <= dat_in_b;
          end
      end

endmodule
```

```
////////////////////////////////////////////////
//
// Header information
// Intermediate Wire Signals
//
////////////////////////////////////////////////

module intermed_wire
            (
               // Inputs
               in_1,
               in_2,
               in_3,
               // Outputs
               out_1,
               out_2
            );

    // Port definitions

    input          in_1;
    input          in_2;
    input          in_3;

    output         out_1;
    output         out_2;

    wire           intermediate_sig;

    // ------------- Design implementation --------

    assign intermediate_sig = in_1 & in_2;

    assign out_1 = intermediate_sig & in_3;
    assign out_2 = intermediate_sig | in_3;

endmodule
```

```
////////////////////////////////////////
// Modular Design #1

module modular_1 (  clk,
                    reset,
                    go_1,
                    kill_1,
                    go_2,
                    kill_2,
                    go_3,
                    kill_3,
                    kill_clr,
                    done_1,
                    done_2,
                    done_3,
                    kill_ltchd
                 );

    input           clk;
    input           reset;
    input           go_1;
    input           kill_1;
    input           go_2;
    input           kill_2;
    input           go_3;
    input           kill_3;
    input           kill_clr;

    output          done_1;
    output          done_2;
    output          done_3;
    output          kill_ltchd;

    reg             kill_ltchd;

    // ------ Design implementation -----

    // first module instantiation
    state_machine_1 go_delay_1
           (
             .reset   ( reset  ),
             .clk     ( clk    ),
             .go      ( go_1   ),
             .kill    ( kill_1 ),
             .done    ( done_1 )
           );

    // second module instantiation
    state_machine_1 go_delay_2
           (
             .reset   ( reset  ),
             .clk     ( clk    ),
```

```
          .go      ( go_2   ),
          .kill    ( kill_2 ),
          .done    ( done_2 )
        );

    // third module instantiation
    state_machine_1 go_delay_3
        (
          .reset   ( reset  ),
          .clk     ( clk    ),
          .go      ( go_3   ),
          .kill    ( kill_3 ),
          .done    ( done_3 )
        );


    // Kill Latch
    always @( posedge clk or posedge reset )
      begin
        if ( reset )
          kill_ltchd <= 1'b0;
        else if (   kill_1
                  || kill_2
                  || kill_3
                )
          kill_ltchd <= 1'b1;
        else if ( kill_clr )
          kill_ltchd <= 1'b0;
      end

endmodule
```

```
////////////////////////////////////////
// Modular Design #2

module modular_2 (  clk,
                    reset,
                    go_1,
                    kill_1,
                    go_2,
                    kill_2,
                    go_3,
                    kill_3,
                    kill_clr,
                    done_out,
                    kill_ltchd
                 );

    input           clk;
    input           reset;
    input           go_1;
    input           kill_1;
    input           go_2;
    input           kill_2;
    input           go_3;
    input           kill_3;
    input           kill_clr;

    output          done_out;
    output          kill_ltchd;

    reg             kill_ltchd;
    wire            done_1;
    wire            done_2;

    // ------ Design implementation -----

    // first module instantiation
    state_machine_1 go_delay_1
           (
             .reset   ( reset  ),
             .clk     ( clk    ),
             .go      ( go_1   ),
             .kill    ( kill_1 ),
             .done    ( done_1 )
           );

    // second module instantiation
    state_machine_1 go_delay_2
           (
             .reset   ( reset  ),
             .clk     ( clk    ),
             .go      ( done_1 | go_2 ),
             .kill    ( kill_2 ),
```

```
          .done    ( done_2 )
        );

    // third module instantiation
    state_machine_1 go_delay_3
        (
          .reset   ( reset    ),
          .clk     ( clk      ),
          .go      (   done_1
                     | done_2
                     | go_3
                   ),
          .kill    ( kill_3   ),
          .done    ( done_out )
        );


    // Kill Latch
    always @( posedge clk or posedge reset )
      begin
        if ( reset )
          kill_ltchd <= 1'b0;
        else if (   kill_1
                  || kill_2
                  || kill_3
                )
          kill_ltchd <= 1'b1;
        else if ( kill_clr )
          kill_ltchd <= 1'b0;
      end

endmodule
```

```
////////////////////////////////////////
// Sample Design for Simulation

module sim_sample (  clk,
                     reset,
                     dat_in,
                     enable,
                     comp_cnt
                   );

    input          clk;
    input          reset;
    input  [7:0]   dat_in;
    input          enable;
    output [9:0]   comp_cnt;

    reg [7:0]   dat_in_d1;
    reg [9:0]   comp_cnt;

    // ------ Design implementation -----

    always @( posedge clk or posedge reset )
      begin
        if ( reset )
          begin
            dat_in_d1 <= 8'h00;
            comp_cnt  <= 10'd0;
          end
        else if ( enable )
          begin
            dat_in_d1 <= dat_in;
            if ( dat_in_d1 == dat_in )
              comp_cnt <= comp_cnt + 1;
          end
      end

endmodule
```

```
////////////////////////////////////////////////
// Simple D-flop
////////////////////////////////////////////////

module simple_dflop  (  clk,
                        in_1,
                        out_1
                      );

    input           clk;
    input           in_1;
    output          out_1;

    reg             out_1;

    // ------------- Design implementation --------

    always @( posedge clk )
      begin
        out_1 <= in_1;
      end

endmodule
```

```
////////////////////////////////////////
// Simple Dual Port Memory

module simple_dp_mem
         ( clk,
           reset,
           dat_in,
           wr_adr,
           wr_en,
           dat_out,
           rd_adr
         );

    input          clk;
    input          reset;
    input  [15:0]  dat_in;
    input  [9:0]   wr_adr;
    input          wr_en;
    output [15:0]  dat_out;
    input  [9:0]   rd_adr;

    reg [15:0]     memory[0:1023];
    reg [15:0]     dat_out;

    // ------ Design implementation -----

    // Memory
    //
    always @( posedge clk )
      begin
        if (wr_en)
          memory[wr_adr] <= dat_in;
        dat_out <= memory[rd_adr];
      end

endmodule
```

```
////////////////////////////////////////////////
//
// Header information -- details about the context,
// constraints, etc..
//
////////////////////////////////////////////////

module simple_in_n_out
            (
               // Inputs
               in_1,
               in_2,
               in_3,
               // Outputs
               out_1,
               out_2
            );

    // Port definitions

    input          in_1;
    input          in_2;
    input          in_3;

    output         out_1;
    output         out_2;

    // ------------- Design implementation --------

    assign out_1 = in_1 & in_2 & in_3;
    assign out_2 = in_1 | in_2 | in_3;


endmodule
```

```
////////////////////////////////////////
// Single-port Memory

module single_port_mem
         ( clk,
           reset,
           data_io,
           address,
           wr_en,
           rd
         );

    input          clk;
    input          reset;
    inout  [15:0]  data_io;  //new I/O type
    input  [9:0]   address;
    input          wr_en;
    input  [9:0]   rd;

    reg [15:0]     memory[0:1023];
    reg [15:0]     dat_out;
    reg            rd_d1;

    // ------ Design implementation -----

    // Memory
    //
    always @( posedge clk )
      begin
        if (wr_en)
          memory[address] <= data_io;
        dat_out <= memory[address];
        rd_d1   <= rd;
      end

    assign data_io = rd_d1 ? dat_out : 16'bz;

endmodule
```

```
////////////////////////////////////////
// SR flop and counter
////////////////////////////////////////

module srflop_n_cntr (  clk,
                        reset,
                        start,
                        stop,
                        count
                      );

    input          clk;
    input          reset;
    input          start;
    input          stop;
    output [3:0]   count;

    reg         cnt_en;
    reg [3:0]   count;
    reg         stop_d1;
    reg         stop_d2;

    // ------ Design implementation -----

    // SR flop
    always @( posedge clk or posedge reset )
      begin
        if ( reset )
          cnt_en <= 1'b0;
        else if ( start )
          cnt_en <= 1'b1;
        else if ( stop )
          cnt_en <= 1'b0;
      end

    // Counter
    always @( posedge clk or posedge reset )
      begin
        if ( reset )
          count <= 4'h0;
        else if (   cnt_en
                  && count == 4'd13
                )
          count <= 4'h0;
        else if ( cnt_en )
          count <= count + 1;
      end

     // delay
    always @( posedge clk or posedge reset )
      begin
        if ( reset )
          begin
            stop_d1 <= 1'b0;
            stop_d2 <= 1'b0;
          end
        else
```

```
            begin
              stop_d1 <= stop;
              stop_d2 <= stop_d1;
            end
        end

endmodule
```

```
//////////////////////////////////////////
// SR flop and counter, one always block
//////////////////////////////////////////

module srflop_n_cntr_1 (  clk,
                          reset,
                          start,
                          stop,
                          count
                       );

    input          clk;
    input          reset;
    input          start;
    input          stop;
    output [3:0]   count;

    reg         cnt_en;
    reg [3:0]   count;
    reg         stop_d1;
    reg         stop_d2;

    // ------ Design implementation -----

    always @( posedge clk or posedge reset )
      begin
        if ( reset )
          begin
            cnt_en <= 1'b0;
            count <= 4'h0;
            stop_d1 <= 1'b0;
            stop_d2 <= 1'b0;
          end
        else
          begin
            if ( start )
              cnt_en <= 1'b1;
            else if ( stop )
              cnt_en <= 1'b0;

            if (   cnt_en
                && count == 4'd13
               )
              count <= 4'h0;
            else if ( cnt_en )
              count <= count + 1;

            stop_d1 <= stop;
            stop_d2 <= stop_d1;
          end
      end

endmodule
```

```
////////////////////////////////////////////////
//
// Header information
// Standard Mux
//
////////////////////////////////////////////////

module standard_mux
            (
               // Inputs
               in_1,
               in_2,
               in_3,
               // Outputs
               out_1,
            );

    // Port definitions

    input  [3:0]   in_1;
    input  [3:0]   in_2;
    input          in_3;

    output [3:0]   out_1;

    // ------------- Design implementation --------

    assign out_1 = in_3_bus ? in_2: in_1;

endmodule
```

```
////////////////////////////////////////
// State Machine

module state_machine_1 (  clk,
                          reset,
                          go,
                          kill,
                          done
                        );

    input          clk;
    input          reset;
    input          go;
    input          kill;
    output         done;

    reg [6:0]   count;
    reg         done;
    reg [1:0]   state_reg;

    // state machine parameters
    parameter   idle     = 2'b00;
    parameter   active   = 2'b01;
    parameter   finish   = 2'b10;
    parameter   abort    = 2'b11;

    // ------ Design implementation -----

    // State Machine
    //
    always @( posedge clk or posedge reset )
      begin
        if ( reset )
          state_reg <= idle;
        else
          case ( state_reg )
            idle :
              if ( go )                    state_reg <= active;

            active :
              if ( kill )                  state_reg <= abort;
              else if ( count == 7'd100 )  state_reg <= finish;

            finish :                       state_reg <= idle;

            abort :
              if ( !kill )                 state_reg <= idle;

            default :                      state_reg <= idle;
          endcase
      end

    // Counter
    always @( posedge clk or posedge reset )
      begin
        if ( reset )
          count <= 7'h00;
```

```
        else if (   state_reg == finish
                 || state_reg == abort
                )
          count <= 7'h00;
        else if ( state_reg == active )
          count <= count + 1;
      end

    // done register
    always @( posedge clk or posedge reset )
      begin
        if ( reset )
          done <= 1'b0;
        else if ( state_reg == finish )
          done <= 1'b1;
        else
          done <= 1'b0;
      end

endmodule
```

```
////////////////////////////////////////
// State Machine

module state_machine_2 ( clk,
                         reset,
                         go,
                         kill,
                         done
                       );

    input          clk;
    input          reset;
    input          go;
    input          kill;
    output         done;

    reg [6:0]   count;
    reg         done;
    reg [1:0]   state_reg;

    // state machine parameters
    parameter   idle     = 2'b00;
    parameter   active   = 2'b01;
    parameter   finish   = 2'b10;
    parameter   abort    = 2'b11;

    // ------ Design implementation -----

    // State Machine
    //
    always @( posedge clk or posedge reset )
      begin
        if ( reset )
          begin
            state_reg <=  idle;
            count     <= 7'h00;
            done      <=  1'b0;
          end
        else
          case ( state_reg )

            idle :
              begin
                count <= 7'h00;
                done  <= 1'b0;
                if ( go )
                  state_reg <= active;
              end

            active :
              begin
                count <= count + 1;
                done  <= 1'b0;
                if ( kill )
                  state_reg <= abort;
                else if ( count == 7'd100 )
                  state_reg <= finish;
```

```
            end

          finish :
            begin
              count     <= 7'h00;
              done      <= 1'b1;
              state_reg <= idle;
            end

          abort :
            begin
              count <= 7'h00;
              done  <= 1'b0;
              if ( !kill )
                state_reg <= idle;
            end

          default :
            begin
              count     <= 7'h00;
              done      <= 1'b0;
              state_reg <= idle;
            end
        endcase
    end

endmodule
```

```
////////////////////////////////////////////////
// Simple Testbench Using Embedded, Explicit Vectors

module tb_sim_sample_1
           (
            // no I/O for the testbench
           );

   // input signals to the test module.
   reg        reset;
   reg        sim_clk;
   reg [7:0]  dat_in;
   reg        enable;

   // output signals from the test module.
   wire [9:0]   comp_cnt;

   // testbench signals.
   integer i;
   integer j;

   // clock periods
   parameter CLK_PERIOD =  10; // 10 ns = 100 MHz.

   // ------ Design implementation -----

   // module under test
   sim_sample  mut
         (
          .clk      ( sim_clk  ),
          .reset    ( reset    ),
          .dat_in   ( dat_in   ),
          .enable   ( enable   ),
          .comp_cnt ( comp_cnt )
         );

   // generate clock and reset

   initial sim_clk = 1'b0;

   always #( CLK_PERIOD/2.0 )
     sim_clk = ~sim_clk;

   initial reset   = 1'b1;
   initial i       = 0;

   // reset goes inactive after 20 clocks
   always @(posedge sim_clk)
     begin
       i = i+1;
       if (i == 20)
         #1 reset <= 1'b0;
     end
```

```
    // feed stimulus vectors to module under test
    initial
    begin
      dat_in  = 8'b0;
      enable  = 0'b0;
      //
      wait (  reset );
      wait ( ~reset );
      @(posedge sim_clk);
      for ( j = 0; j < 20; j = j + 1 )
        begin
          @(posedge sim_clk);
        end
      enable  = 1'b1;
      dat_in  = 8'h00;
      //
      @(posedge sim_clk);
      dat_in  = 8'h01;
      @(posedge sim_clk);
      dat_in  = 8'h20;
      @(posedge sim_clk);
      dat_in  = 8'h21;
      @(posedge sim_clk);
      dat_in  = 8'h21;
      @(posedge sim_clk);
      dat_in  = 8'h33;
      @(posedge sim_clk);
      dat_in  = 8'h56;
      @(posedge sim_clk);
      dat_in  = 8'h56;
      @(posedge sim_clk);
      dat_in  = 8'h33;
      //
      @(posedge sim_clk);
      enable  = 1'b0;
      //
      forever
        begin
          @(posedge sim_clk)
          enable  = 1'b0;
        end
    end

endmodule
```

```
////////////////////////////////////////////////////
// Simple Testbench Using Embedded, Automatic Vectors

module tb_sim_sample_2
           (
            // no I/O for the testbench
           );

   parameter QUANT_VECTORS =  32; //quantity of vector samples

   // input signals to the test module.
   reg        reset;
   reg        sim_clk;
   reg [7:0]  dat_in;
   reg        enable;
   reg [31:0] random_num;

   // output signals from the test module.
   wire [9:0]   comp_cnt;

   // testbench signals.
   integer i;
   integer j;

   // clock periods
   parameter CLK_PERIOD =  10; // 10 ns = 100 MHz.

   // ------ Design implementation -----

   // module under test
   sim_sample  mut
         (
          .clk      ( sim_clk  ),
          .reset    ( reset    ),
          .dat_in   ( dat_in   ),
          .enable   ( enable   ),
          .comp_cnt ( comp_cnt )
         );

   // generate clock and reset

   initial sim_clk = 1'b0;

   always #( CLK_PERIOD/2.0 )
     sim_clk = ~sim_clk;

   initial reset   = 1'b1;
   initial i       = 0;

   // reset goes inactive after 20 clocks
   always @(posedge sim_clk)
     begin
       i = i+1;
```

```
      if (i == 20)
        #1 reset <= 1'b0;
    end

  // feed stimulus vectors to module under test
  initial
  begin
    dat_in  = 8'b0;
    enable  = 0'b0;
    random_num = $random(1);
    //
    wait (  reset );
    wait ( ~reset );
    @(posedge sim_clk);
    for ( j = 0; j < 20; j = j + 1 )
      begin
        @(posedge sim_clk);
      end
    enable  = 1'b1;
    dat_in  = 8'h00;
    //
    for ( j = 0; j < (QUANT_VECTORS); j = j + 1 )
      begin
        @(posedge sim_clk);
        random_num = $random;
        if ( random_num[2:0] != 3'h0 )
          dat_in = dat_in + 1;
      end
    //
    @(posedge sim_clk);
    enable  = 1'b0;
    //
    forever
      begin
        @(posedge sim_clk)
        enable  = 1'b0;
      end
  end

endmodule
```