Complete SystemVerilog Reference Guide

Comprehensive Coverage with Examples

Author: Shahrear Hossain Shawon
Algo Science Lab

1 Data Types

1.1 Logic and Bit Types

SystemVerilog provides 4-state (0,1,X,Z) and 2-state (0,1) data types for hardware modeling.

```
// 4-state types
logic [7:0] data;       // 8-bit
logic signed [15:0] s; // signed
reg [3:0] old_style;    // legacy

// 2-state types
bit [7:0] fast_data;    // faster simulation
byte signed_byte;       // 8-bit signed
shortint si;            // 16-bit signed
int i;                  // 32-bit signed
longint li;             // 64-bit signed
```

1.2 Enumerated Types

Enumerations provide named values for better readability and type safety.

```
typedef enum logic [1:0] {
    IDLE = 2'b00,
    READ = 2'b01,
    WRITE = 2'b10,
    ERROR = 2'b11
} state_t;

state_t current_state, next_state;
```

1.3 Structures and Unions

Structures group related data, unions save space by overlapping members.

```
// Structure
typedef struct packed {
    logic [7:0] opcode;
    logic [15:0] addr;
    logic [31:0] data;
} instruction_t;

instruction_t instr;

// Union
typedef union packed {
    logic [31:0] word;
    logic [15:0] half[2];
    logic [7:0] byte[4];
} data_u;
```

1.4 Arrays

SystemVerilog supports packed, unpacked, dynamic, associative, and queue arrays.

```
// Packed array
logic [7:0][3:0] packed_arr;

// Unpacked array
logic [7:0] mem [0:255];

// Dynamic array
int dyn_arr[];
initial dyn_arr = new[10];

// Associative array
int assoc_arr[string];

// Queue
int queue[$];
queue.push_back(5);
```

2 Modules and Interfaces

2.1 Module with Parameters

Modules are the basic building blocks with optional parameters for configurability.

```
module adder #(
    parameter WIDTH = 8
)(
    input  logic [WIDTH-1:0] a, b,
    input  logic cin,
    output logic [WIDTH-1:0] sum,
    output logic cout
);
    assign {cout, sum} = a + b + cin;
endmodule
```

2.2 Interface

Interfaces encapsulate communication signals between modules.

```
interface axi_if #(
    parameter ADDR_WIDTH = 32,
    parameter DATA_WIDTH = 32
)(
    input logic clk, rst_n
);
    logic [ADDR_WIDTH-1:0] awaddr;
    logic awvalid, awready;
    logic [DATA_WIDTH-1:0] wdata;
    logic wvalid, wready;

    modport master (
        output awaddr, awvalid, wdata, wvalid,
        input  awready, wready
    );

    modport slave (
        input  awaddr, awvalid, wdata, wvalid,
        output awready, wready
    );
endinterface
```

2.3 Module with Interface

```
module axi_master(
    axi_if.master bus
);
    always_ff @(posedge bus.clk) begin
        if (!bus.rst_n)
            bus.awvalid <= 0;
        else if (bus.awready)
            bus.awvalid <= 1;
    end
endmodule
```

3 Sequential Logic

3.1 Always FF Blocks

Use `always_ff` for synthesizable sequential logic.

```
module dff(
    input  logic clk, rst_n,
    input  logic d,
    output logic q
);
    always_ff @(posedge clk or negedge rst_n)
    begin
        if (!rst_n)
            q <= 1'b0;
        else
            q <= d;
    end
endmodule
```

3.2 Counter Example

```
module counter #(
    parameter WIDTH = 8
)(
    input  logic clk, rst_n, en,
    output logic [WIDTH-1:0] count
);
    always_ff @(posedge clk or negedge rst_n)
    begin
        if (!rst_n)
            count <= '0;
        else if (en)
            count <= count + 1;
    end
endmodule
```

4 Combinational Logic

4.1 Always Comb Blocks

Use `always_comb` for combinational logic.

```
module mux4to1(
    input  logic [1:0] sel,
    input  logic [3:0] in,
    output logic out
);
    always_comb begin
        case(sel)
            2'b00: out = in[0];
            2'b01: out = in[1];
            2'b10: out = in[2];
            2'b11: out = in[3];
        endcase
    end
endmodule
```

4.2 Priority and Unique

`priority` and `unique` provide synthesis and simulation checks.

```
module decoder(
    input  logic [2:0] in,
    output logic [7:0] out
);
    always_comb begin
        unique case(in)
            3'd0: out = 8'b00000001;
            3'd1: out = 8'b00000010;
            3'd2: out = 8'b00000100;
            3'd3: out = 8'b00001000;
            3'd4: out = 8'b00010000;
            3'd5: out = 8'b00100000;
            3'd6: out = 8'b01000000;
            3'd7: out = 8'b10000000;
        endcase
    end
endmodule
```

5 State Machines

5.1 FSM with Enum

Finite state machines are fundamental in digital design.

```
module fsm(
    input  logic clk, rst_n,
    input  logic start, done,
    output logic busy
);
    typedef enum logic [1:0] {
        IDLE, ACTIVE, WAIT, COMPLETE
    } state_t;

    state_t state, next;

    // State register
    always_ff @(posedge clk or negedge rst_n)
    begin
        if (!rst_n)
            state <= IDLE;
        else
            state <= next;
    end

    // Next state logic
    always_comb begin
        next = state;
        case(state)
            IDLE: if (start) next = ACTIVE;
            ACTIVE: next = WAIT;
            WAIT: if (done) next = COMPLETE;
            COMPLETE: next = IDLE;
        endcase
    end

    // Output logic
    assign busy = (state != IDLE);
endmodule
```

6 Functions and Tasks

6.1 Functions

Functions return a single value and execute in zero simulation time.

```
function automatic int parity(
    input logic [7:0] data
);
    parity = ^data; // XOR reduction
endfunction

function automatic logic [7:0] reverse(
    input logic [7:0] in
);
    for (int i=0; i<8; i++)
        reverse[i] = in[7-i];
endfunction
```

6.2 Tasks

Tasks can have multiple outputs and consume simulation time.

```
task automatic read_mem(
    input  logic [7:0] addr,
    output logic [31:0] data,
    ref    logic [31:0] mem[]
);
    #10 data = mem[addr];
endtask

```

```
task automatic write_mem(
    input logic [7:0] addr,
    input logic [31:0] data,
    ref   logic [31:0] mem[]
);
    #10 mem[addr] = data;
endtask
```

7 Assertions

7.1 Immediate Assertions

Checked immediately like a procedural statement.

```
module assert_example(
    input logic clk, rst_n,
    input logic req, gnt
);
    // Immediate assertion
    always_comb begin
        assert (!(req && !gnt) || gnt)
        else $error("Grant without request");
    end
endmodule
```

7.2 Concurrent Assertions

Checked based on clock edges, using sequences and properties.

```
module sva_example(
    input logic clk, rst_n,
    input logic req, gnt, done
);
    // Property: grant follows request
    property p_req_gnt;
        @(posedge clk) disable iff (!rst_n)
        req |-> ##[1:3] gnt;
    endproperty

    assert property (p_req_gnt)
    else $error("Grant not received");

    // Sequence: done after grant
    sequence s_gnt_done;
        gnt ##[1:5] done;
    endsequence

    assert property (
        @(posedge clk) disable iff (!rst_n)
        gnt |-> s_gnt_done
    );
endmodule
```

8 Constrained Random

8.1 Classes with Constraints

Object-oriented features for verification.

```
class packet;
    rand bit [7:0] addr;
    rand bit [15:0] data;
    rand bit [1:0] ptype;

    constraint c_addr {
        addr inside {[8'h00:8'h7F]};
    }

    constraint c_type {
        ptype dist {
            2'b00 := 40,
            2'b01 := 30,
            2'b10 := 20,
            2'b11 := 10
        };
    }

    function void display();
        $display("Addr=%h Data=%h Type=%b",
                 addr, data, ptype);
    endfunction
endclass

module test;
    initial begin
        packet pkt = new();
        repeat(5) begin
            assert(pkt.randomize());
            pkt.display();
        end
    end
endmodule
```

9 Coverage

9.1 Covergroups

Functional coverage to track test completeness.

```
module coverage_example(
    input logic clk,
    input logic [2:0] opcode,
    input logic [7:0] data
);
    covergroup cg @(posedge clk);
        cp_opcode: coverpoint opcode {
            bins add = {3'b000};
            bins sub = {3'b001};
            bins mul = {3'b010};
            bins div = {3'b011};
            bins others = default;
        }

        cp_data: coverpoint data {
            bins low = {[0:63]};
            bins mid = {[64:191]};
            bins high = {[192:255]};
        }

        cross_op_data: cross cp_opcode, cp_data;
    endgroup

    cg cg_inst = new();
endmodule
```

10 Clocking Blocks

10.1 Synchronous Testbench

Clocking blocks provide synchronous drive and sample timing.

```
interface bus_if(input logic clk);
    logic [7:0] data;
    logic valid, ready;

    clocking cb @(posedge clk);
        default input #1step output #2;
        output data, valid;
        input ready;
    endclocking

    modport TB (clocking cb);
endinterface

program test(bus_if.TB bus);
    initial begin
        bus.cb.data <= 8'hAA;
        bus.cb.valid <= 1;
        @(bus.cb);
        wait(bus.cb.ready == 1);
    end
```

```
endprogram
```

11 Virtual Interfaces

11.1 Polymorphism in Verification

Virtual interfaces enable dynamic interface binding.

```
class driver;
    virtual bus_if vif;

    function new(virtual bus_if vif);
        this.vif = vif;
    endfunction

    task run();
        forever begin
            @(vif.cb);
            vif.cb.data <= $random;
            vif.cb.valid <= 1;
            @(vif.cb);
            vif.cb.valid <= 0;
        end
    endtask
endclass

module top;
    logic clk;
    bus_if bus(clk);

    initial begin
        driver drv = new(bus);
        fork
            drv.run();
        join_none
    end
endmodule
```

12 Parameterized Classes

12.1 Generic Components

```
class queue #(type T = int);
    T items[$];

    function void push(T item);
        items.push_back(item);
    endfunction

    function T pop();
        return items.pop_front();
    endfunction

    function int size();
        return items.size();
    endfunction
endclass

// Usage
queue#(int) int_q = new();
queue#(byte) byte_q = new();
```

13 Interprocess Communication

13.1 Mailbox

```
module ipc_example;
    mailbox #(int) mbx = new();

    initial begin // Producer
        for (int i=0; i<10; i++) begin
            mbx.put(i);
            $display("Sent: %0d", i);
            #10;
        end
    end

    initial begin // Consumer
        int data;
        repeat(10) begin
            mbx.get(data);
            $display("Received: %0d", data);
        end
    end
endmodule
```

13.2 Semaphore

```
module semaphore_example;
    semaphore sem = new(1);

    task access_resource(int id);
        sem.get();
        $display("Task %0d accessing", id);
        #20;
        $display("Task %0d done", id);
        sem.put();
    endtask

    initial fork
        access_resource(1);
        access_resource(2);
    join
endmodule
```

14 DPI (Direct Programming Interface)

14.1 Importing C Functions

```
// C function
import "DPI-C" function int c_add(
    input int a, input int b
);

module dpi_test;
    initial begin
        int result;
        result = c_add(5, 3);
        $display("Result = %0d", result);
    end
endmodule
```

15 Advanced Memory Modeling

15.1 Associative Array with Methods

```
module mem_model;
    int mem[bit[15:0]];

    initial begin
        mem[16'h0000] = 32'hDEADBEEF;
        mem[16'h0100] = 32'hCAFEBABE;

        // Iterate
        foreach(mem[addr]) begin
            $display("Addr=%h Data=%h",
                     addr, mem[addr]);
        end

        // Check existence
        if (mem.exists(16'h0000))
            $display("Address exists");

        // Delete
        mem.delete(16'h0000);
        $display("Size: %0d", mem.size());
    end
endmodule
```

16 Practical Examples

16.1 FIFO Design

```
module fifo #(
    parameter DEPTH = 16,
    parameter WIDTH = 8
)(
    input  logic clk, rst_n,
    input  logic wr_en, rd_en,
    input  logic [WIDTH-1:0] wr_data,
    output logic [WIDTH-1:0] rd_data,
    output logic full, empty
);
    logic [WIDTH-1:0] mem[DEPTH];
    logic [$clog2(DEPTH):0] wr_ptr, rd_ptr;

    assign full = (wr_ptr[$clog2(DEPTH)] !=
                   rd_ptr[$clog2(DEPTH)]) &&
                  (wr_ptr[$clog2(DEPTH)-1:0]
    ==
                   rd_ptr[$clog2(DEPTH)-1:0]);
    assign empty = (wr_ptr == rd_ptr);

    always_ff @(posedge clk or negedge rst_n)
    begin
        if (!rst_n) begin
            wr_ptr <= '0;
            rd_ptr <= '0;
        end else begin
            if (wr_en && !full)
                wr_ptr <= wr_ptr + 1;
            if (rd_en && !empty)
                rd_ptr <= rd_ptr + 1;
        end
    end

    always_ff @(posedge clk) begin
        if (wr_en && !full)
            mem[wr_ptr[$clog2(DEPTH)-1:0]] <=
                wr_data;
    end

    assign rd_data =
        mem[rd_ptr[$clog2(DEPTH)-1:0]];
endmodule
```

16.2 UART Transmitter

```
module uart_tx #(
    parameter CLK_FREQ = 50_000_000,
    parameter BAUD_RATE = 115200
)(
    input  logic clk, rst_n,
    input  logic [7:0] data,
    input  logic start,
    output logic tx,
    output logic busy
);
    localparam CLKS_PER_BIT =
        CLK_FREQ / BAUD_RATE;

    typedef enum logic [2:0] {
        IDLE, START, DATA, STOP
    } state_t;

    state_t state;
    logic [7:0] tx_data;
    logic [$clog2(CLKS_PER_BIT)-1:0] clk_cnt;
    logic [2:0] bit_cnt;

    assign busy = (state != IDLE);

    always_ff @(posedge clk or negedge rst_n)
    begin
        if (!rst_n) begin
            state <= IDLE;
            tx <= 1;
            clk_cnt <= 0;
            bit_cnt <= 0;
        end else begin
            case(state)
                IDLE: begin
                    tx <= 1;
                    if (start) begin
                        tx_data <= data;
                        state <= START;
                        clk_cnt <= 0;
                    end
                end

                START: begin
                    tx <= 0;
                    if (clk_cnt ==
    CLKS_PER_BIT-1) begin
                        state <= DATA;
                        clk_cnt <= 0;
                        bit_cnt <= 0;
                    end else
                        clk_cnt <= clk_cnt +
    1;
                end

                DATA: begin
                    tx <= tx_data[bit_cnt];
                    if (clk_cnt ==
    CLKS_PER_BIT-1) begin
                        clk_cnt <= 0;
                        if (bit_cnt == 7)
                            state <= STOP;
                        else
                            bit_cnt <= bit_cnt
     + 1;
                    end else
                        clk_cnt <= clk_cnt +
    1;
                end

                STOP: begin
                    tx <= 1;
                    if (clk_cnt ==
    CLKS_PER_BIT-1)
                        state <= IDLE;
                    else
                        clk_cnt <= clk_cnt +
    1;
                end
            endcase
        end
    end
endmodule
```

Conclusion

This reference covers the essential SystemVerilog constructs for both RTL design and verification. SystemVerilog combines the best of Verilog with modern programming concepts, making it the industry standard for digital design and verification.

Algo Science Lab - SystemVerilog Reference v1.0
Author: Shahrear Hossain Shawon