# Chapter 5: Modules and Interfaces

## Module Basics

Modules are the fundamental building blocks of SystemVerilog designs. They encapsulate functionality and provide a way to create hierarchical designs through instantiation.

### Basic Module Structure

```
module module_name #(
    // Parameters (optional)
    parameter int WIDTH = 8
) (
    // Port declarations
    input  logic clk,
    input  logic reset_n,
    input  logic [WIDTH-1:0] data_in,
    output logic [WIDTH-1:0] data_out
);

    // Module body - internal logic
    always_ff @(posedge clk or negedge reset_n) begin
        if (!reset_n)
            data_out <= '0;
        else
            data_out <= data_in;
    end

endmodule
```

### Module Instantiation

```
// Named port connections (recommended)
module_name #(.WIDTH(16)) inst_name (
    .clk(system_clk),
    .reset_n(sys_reset),
    .data_in(input_data),
    .data_out(output_data)
);

// Positional port connections (not recommended for complex modules)
module_name #(16) inst_name (system_clk, sys_reset, input_data, output_data);
```

### Key Module Concepts

**Scope and Hierarchy**: Each module creates its own scope. Internal signals and variables are not accessible from outside the module unless explicitly connected through ports.

**Instance vs Module**: A module is the template/definition, while an instance is a specific instantiation of that module in your design.

### Example 1: Simple Counter

simple_counter - Basic module structure with parameters and ANSI-style ports

```
// simple_counter.sv
module simple_counter #(
    // Parameters with default values
    parameter int WIDTH = 8,                    // Counter width in bits
    parameter int MAX_COUNT = 255,              // Maximum count value
    parameter bit WRAP_AROUND = 1'b1,           // Enable wrap-around behavior
    parameter int RESET_VALUE = 0               // Reset value for counter
) (
    // ANSI-style port declarations
    input  logic                    clk,        // Clock input
    input  logic                    reset_n,    // Active-low asynchronous reset
    input  logic                    enable,     // Counter enable
    input  logic                    load,       // Load enable
    input  logic [WIDTH-1:0]        load_value, // Value to load
    input  logic                    count_up,   // Count direction (1=up, 0=down)
    output logic [WIDTH-1:0]        count,      // Current counter value
    output logic                    overflow,   // Overflow flag
    output logic                    underflow,  // Underflow flag
    output logic                    max_reached // Maximum count reached flag
);

    // Internal register to hold counter value
    logic [WIDTH-1:0] count_reg;

    // Derived parameters using localparam
    localparam logic [WIDTH-1:0] MIN_COUNT = '0;
    localparam logic [WIDTH-1:0] MAX_VAL = MAX_COUNT[WIDTH-1:0];
    localparam logic [WIDTH-1:0] RESET_VAL = RESET_VALUE[WIDTH-1:0];

    // Main counter logic
    always_ff @(posedge clk or negedge reset_n) begin
        if (!reset_n) begin
            // Asynchronous reset
            count_reg <= RESET_VAL;
            overflow <= 1'b0;
            underflow <= 1'b0;
        end else begin
            // Clear flags by default
            overflow <= 1'b0;
            underflow <= 1'b0;

            if (load) begin
                // Load operation has highest priority
                count_reg <= load_value;
```

```
            end else if (enable) begin
                if (count_up) begin
                    // Count up logic
                    if (count_reg >= MAX_VAL) begin
                        overflow <= 1'b1;
                        if (WRAP_AROUND) begin
                            count_reg <= MIN_COUNT;
                        end else begin
                            count_reg <= MAX_VAL; // Saturate at maximum
                        end
                    end else begin
                        count_reg <= count_reg + 1'b1;
                    end
                end else begin
                    // Count down logic
                    if (count_reg <= MIN_COUNT) begin
                        underflow <= 1'b1;
                        if (WRAP_AROUND) begin
                            count_reg <= MAX_VAL;
                        end else begin
                            count_reg <= MIN_COUNT; // Saturate at minimum
                        end
                    end else begin
                        count_reg <= count_reg - 1'b1;
                    end
                end
            end
            // If enable is false, counter holds its current value
        end
    end

    // Continuous assignments for outputs
    assign count = count_reg;
    assign max_reached = (count_reg == MAX_VAL);

    // Assertions for parameter validation (synthesis will ignore these)
    initial begin
        assert (WIDTH > 0)
            else $error("WIDTH parameter must be greater than 0");
        assert (MAX_COUNT >= 0)
            else $error("MAX_COUNT parameter must be non-negative");
        assert (MAX_COUNT < (2**WIDTH))
            else $error("MAX_COUNT exceeds maximum value for given WIDTH");
        assert (RESET_VALUE >= 0)
            else $error("RESET_VALUE parameter must be non-negative");
        assert (RESET_VALUE < (2**WIDTH))
            else $error("RESET_VALUE exceeds maximum value for given WIDTH");

        $display();
        $display("Simple Counter Module Initialized:");
        $display("  WIDTH = %0d bits", WIDTH);
        $display("  MAX_COUNT = %0d", MAX_COUNT);
        $display("  WRAP_AROUND = %b", WRAP_AROUND);
        $display("  RESET_VALUE = %0d", RESET_VALUE);
        $display("  Counter range: %0d to %0d", MIN_COUNT, MAX_VAL);
```

```
    end

endmodule
```

```
// simple_counter_testbench.sv
module simple_counter_testbench;

    // Testbench parameters
    localparam int CLK_PERIOD = 10;  // 100MHz clock
    localparam int TEST_WIDTH = 4;   // 4-bit counter for easy testing
    localparam int TEST_MAX = 12;    // Max count less than 2^4-1 for testing

    // Testbench signals
    logic                    clk;
    logic                    reset_n;
    logic                    enable;
    logic                    load;
    logic [TEST_WIDTH-1:0]   load_value;
    logic                    count_up;
    logic [TEST_WIDTH-1:0]   count;
    logic                    overflow;
    logic                    underflow;
    logic                    max_reached;

    // Clock generation
    initial begin
        clk = 0;
        forever #(CLK_PERIOD/2) clk = ~clk;
    end

    // DUT instantiation with custom parameters
    simple_counter #(
        .WIDTH(TEST_WIDTH),
        .MAX_COUNT(TEST_MAX),
        .WRAP_AROUND(1'b1),
        .RESET_VALUE(0)
    ) dut (
        .clk(clk),
        .reset_n(reset_n),
        .enable(enable),
        .load(load),
        .load_value(load_value),
        .count_up(count_up),
        .count(count),
        .overflow(overflow),
        .underflow(underflow),
        .max_reached(max_reached)
    );

    // Second instance with different parameters for comparison
    logic [7:0] count2;
    logic overflow2, underflow2, max_reached2;

    simple_counter #(
        .WIDTH(8),
```

```
        .MAX_COUNT(255),
        .WRAP_AROUND(1'b0),     // No wrap-around (saturation mode)
        .RESET_VALUE(128)       // Different reset value
    ) dut2 (
        .clk(clk),
        .reset_n(reset_n),
        .enable(enable),
        .load(1'b0),            // Disable load for this instance
        .load_value(8'h00),
        .count_up(count_up),
        .count(count2),
        .overflow(overflow2),
        .underflow(underflow2),
        .max_reached(max_reached2)
    );

    // Test stimulus
    initial begin
        // Initialize VCD dump
        $dumpfile("simple_counter_testbench.vcd");
        $dumpvars(0, simple_counter_testbench);

        $display();
        $display("=== Simple Counter Testbench Started ===");
        $display();

        // Initialize signals
        reset_n = 0;
        enable = 0;
        load = 0;
        load_value = 0;
        count_up = 1;

        // Reset phase
        $display("Phase 1: Reset Test");
        #(CLK_PERIOD * 2);
        reset_n = 1;
        #(CLK_PERIOD);
        $display("After reset - DUT1 count: %0d", count);
        $display("After reset - DUT2 count: %0d", count2);
        $display();

        // Test counting up
        $display("Phase 2: Count Up Test");
        enable = 1;
        count_up = 1;

        repeat (TEST_MAX + 3) begin
            #(CLK_PERIOD);
            $display(
                "Count up - DUT1: %0d (overflow=%b, max_reached=%b)",
                count, overflow, max_reached);
            $display("Count up - DUT2: %0d (overflow=%b)", count2, overflow2);
        end
        $display();
```

```
        // Test counting down
        $display("Phase 3: Count Down Test");
        count_up = 0;

        repeat (TEST_MAX + 3) begin
            #(CLK_PERIOD);
            $display(
                "Count down - DUT1: %0d (underflow=%b)", count, underflow);
            $display(
                "Count down - DUT2: %0d (underflow=%b)", count2, underflow2);
        end
        $display();

        // Test load operation
        $display("Phase 4: Load Operation Test");
        count_up = 1;
        load_value = TEST_WIDTH'(TEST_MAX / 2);  // Explicit width casting
        load = 1;
        #(CLK_PERIOD);
        $display("After load %0d - DUT1 count: %0d", load_value, count);
        load = 0;
        #(CLK_PERIOD);
        $display("After load release - DUT1 count: %0d", count);
        $display();

        // Test enable control
        $display("Phase 5: Enable Control Test");
        enable = 0;
        repeat (3) begin
            #(CLK_PERIOD);
            $display("Enable=0 - DUT1 count: %0d (should not change)", count);
        end

        enable = 1;
        repeat (3) begin
            #(CLK_PERIOD);
            $display("Enable=1 - DUT1 count: %0d (should increment)", count);
        end
        $display();

        // Test different parameter behavior
        $display("Phase 6: Parameter Comparison");
        $display(
            "DUT1 (4-bit, wrap-around): count=%0d, max_count=%0d",
            count, TEST_MAX);
        $display(
            "DUT2 (8-bit, saturation): count=%0d, max_count=255",
            count2);
        $display();

        // Final phase - reset test
        $display("Phase 7: Final Reset Test");
        reset_n = 0;
        #(CLK_PERIOD);
        reset_n = 1;
```

```
        #(CLK_PERIOD);
        $display("After final reset - DUT1: %0d, DUT2: %0d", count, count2);
        $display();

        $display("=== Simple Counter Testbench Completed ===");
        $display("Total simulation time: %0t", $time);
        $display();
        $finish;
    end

    // Monitor for detecting important events
    always @(posedge clk) begin
        if (reset_n) begin
            if (overflow)
                $display(
                    "*** OVERFLOW detected at time %0t, count=%0d ***",
                    $time, count);
            if (underflow)
                $display(
                    "*** UNDERFLOW detected at time %0t, count=%0d ***",
                    $time, count);
            if (max_reached && enable)
                $display(
                    "*** MAX_REACHED at time %0t, count=%0d ***",
                    $time, count);
        end
    end

endmodule
```

```
Verilator Simulation Output:
================================================================================
- Verilator: Built from 0.046 MB sources in 3 modules, into 0.084 MB in 10 C++
files needing 0.000 MB
- Verilator: Walltime 29.737 s (elab=0.012, cvt=0.062, bld=29.462); cpu 0.063 s
on 1 threads; alloced 20.176 MB

Simple Counter Module Initialized:
  WIDTH = 4 bits
  MAX_COUNT = 12
  WRAP_AROUND = 1
  RESET_VALUE = 0
  Counter range: 0 to 12

Simple Counter Module Initialized:
  WIDTH = 8 bits
  MAX_COUNT = 255
  WRAP_AROUND = 0
  RESET_VALUE = 128
  Counter range: 0 to 255

=== Simple Counter Testbench Started ===

Phase 1: Reset Test
After reset - DUT1 count: 0
After reset - DUT2 count: 128
```

```
Phase 2: Count Up Test
Count up - DUT1: 1 (overflow=0, max_reached=0)
Count up - DUT2: 129 (overflow=0)
Count up - DUT1: 2 (overflow=0, max_reached=0)
Count up - DUT2: 130 (overflow=0)
Count up - DUT1: 3 (overflow=0, max_reached=0)
Count up - DUT2: 131 (overflow=0)
Count up - DUT1: 4 (overflow=0, max_reached=0)
Count up - DUT2: 132 (overflow=0)
Count up - DUT1: 5 (overflow=0, max_reached=0)
Count up - DUT2: 133 (overflow=0)
Count up - DUT1: 6 (overflow=0, max_reached=0)
Count up - DUT2: 134 (overflow=0)
Count up - DUT1: 7 (overflow=0, max_reached=0)
Count up - DUT2: 135 (overflow=0)
Count up - DUT1: 8 (overflow=0, max_reached=0)
Count up - DUT2: 136 (overflow=0)
Count up - DUT1: 9 (overflow=0, max_reached=0)
Count up - DUT2: 137 (overflow=0)
Count up - DUT1: 10 (overflow=0, max_reached=0)
Count up - DUT2: 138 (overflow=0)
Count up - DUT1: 11 (overflow=0, max_reached=0)
Count up - DUT2: 139 (overflow=0)
Count up - DUT1: 12 (overflow=0, max_reached=1)
Count up - DUT2: 140 (overflow=0)
*** MAX_REACHED at time 155, count=12 ***
Count up - DUT1: 0 (overflow=1, max_reached=0)
Count up - DUT2: 141 (overflow=0)
*** OVERFLOW detected at time 165, count=0 ***
Count up - DUT1: 1 (overflow=0, max_reached=0)
Count up - DUT2: 142 (overflow=0)
Count up - DUT1: 2 (overflow=0, max_reached=0)
Count up - DUT2: 143 (overflow=0)

Phase 3: Count Down Test
Count down - DUT1: 1 (underflow=0)
Count down - DUT2: 142 (underflow=0)
Count down - DUT1: 0 (underflow=0)
Count down - DUT2: 141 (underflow=0)
Count down - DUT1: 12 (underflow=1)
Count down - DUT2: 140 (underflow=0)
*** UNDERFLOW detected at time 215, count=12 ***
*** MAX_REACHED at time 215, count=12 ***
Count down - DUT1: 11 (underflow=0)
Count down - DUT2: 139 (underflow=0)
Count down - DUT1: 10 (underflow=0)
Count down - DUT2: 138 (underflow=0)
Count down - DUT1: 9 (underflow=0)
Count down - DUT2: 137 (underflow=0)
Count down - DUT1: 8 (underflow=0)
Count down - DUT2: 136 (underflow=0)
Count down - DUT1: 7 (underflow=0)
Count down - DUT2: 135 (underflow=0)
Count down - DUT1: 6 (underflow=0)
Count down - DUT2: 134 (underflow=0)
```

```
Count down - DUT1: 5 (underflow=0)
Count down - DUT2: 133 (underflow=0)
Count down - DUT1: 4 (underflow=0)
Count down - DUT2: 132 (underflow=0)
Count down - DUT1: 3 (underflow=0)
Count down - DUT2: 131 (underflow=0)
Count down - DUT1: 2 (underflow=0)
Count down - DUT2: 130 (underflow=0)
Count down - DUT1: 1 (underflow=0)
Count down - DUT2: 129 (underflow=0)
Count down - DUT1: 0 (underflow=0)
Count down - DUT2: 128 (underflow=0)

Phase 4: Load Operation Test
After load 6 - DUT1 count: 6
After load release - DUT1 count: 7

Phase 5: Enable Control Test
Enable=0 - DUT1 count: 7 (should not change)
Enable=0 - DUT1 count: 7 (should not change)
Enable=0 - DUT1 count: 7 (should not change)
Enable=1 - DUT1 count: 8 (should increment)
Enable=1 - DUT1 count: 9 (should increment)
Enable=1 - DUT1 count: 10 (should increment)

Phase 6: Parameter Comparison
DUT1 (4-bit, wrap-around): count=10, max_count=12
DUT2 (8-bit, saturation): count=133, max_count=255

Phase 7: Final Reset Test
After final reset - DUT1: 1, DUT2: 129

=== Simple Counter Testbench Completed ===
Total simulation time: 430

================================================================================
Process finished with return code: 0
Removing Chapter_5_examples/example_1__simple_counter/obj_dir directory...
Chapter_5_examples/example_1__simple_counter/obj_dir removed successfully.
```

0

```
from gtkwave_runner import run_docker_compose

run_docker_compose("Chapter_5_examples/example_1__simple_counter/")
```

```
Docker Compose Output:
================================================================================
 Container notebooks-verilator-1  Creating
 Container notebooks-verilator-1  Created
 Container notebooks-verilator-1  Starting
 Container notebooks-verilator-1  Started
Could not initialize GTK!  Is DISPLAY env var/xhost set?

Usage: gtkwave [OPTION]... [DUMPFILE] [SAVEFILE] [RCFILE]

  -n, --nocli=DIRPATH        use file requester for dumpfile name
```

```
  -f, --dump=FILE           specify dumpfile name
  -F, --fastload            generate/use VCD recoder fastload files
  -o, --optimize            optimize VCD to FST
  -a, --save=FILE           specify savefile name
  -A, --autosavename        assume savefile is suffix modified dumpfile name
  -r, --rcfile=FILE         specify override .rcfile name
  -d, --defaultskip         if missing .rcfile, do not use useful defaults
  -D, --dualid=WHICH        specify multisession identifier
  -l, --logfile=FILE        specify simulation logfile name for time values
  -s, --start=TIME          specify start time for LXT2/VZT block skip
  -e, --end=TIME            specify end time for LXT2/VZT block skip
  -t, --stems=FILE          specify stems file for source code annotation
  -c, --cpu=NUMCPUS         specify number of CPUs for parallelizable ops
  -N, --nowm                disable window manager for most windows
  -M, --nomenus             do not render menubar (for making applets)
  -S, --script=FILE         specify Tcl command script file for execution
  -T, --tcl_init=FILE       specify Tcl command script file to be loaded on
startup
  -W, --wish                enable Tcl command line on stdio
  -R, --repscript=FILE      specify timer-driven Tcl command script file
  -P, --repperiod=VALUE     specify repscript period in msec (default: 500)
  -X, --xid=XID             specify XID of window for GtkPlug to connect to
  -1, --rpcid=RPCID         specify RPCID of GConf session
  -2, --chdir=DIR           specify new current working directory
  -3, --restore             restore previous session
  -4, --rcvar               specify single rc variable values individually
  -5, --sstexclude          specify sst exclusion filter filename
  -I, --interactive         interactive VCD mode (filename is shared mem ID)
  -C, --comphier            use compressed hierarchy names (slower)
  -g, --giga                use gigabyte mempacking when recoding (slower)
  -L, --legacy              use legacy VCD mode rather than the VCD recoder
  -v, --vcd                 use stdin as a VCD dumpfile
  -O, --output=FILE         specify filename for stdout/stderr redirect
  -z, --slider-zoom         enable horizontal slider stretch zoom
  -V, --version             display version banner then exit
  -h, --help                display this help then exit
  -x, --exit                exit after loading trace (for loader benchmarks)

VCD files and save files may be compressed with zip or gzip.
GHW files may be compressed with gzip or bzip2.
Other formats must remain uncompressed due to their non-linear access.
Note that DUMPFILE is optional if the --dump or --nocli options are specified.
SAVEFILE and RCFILE are always optional.

Report bugs to <bybell@rocketmail.com>.
================================================================================
Process finished with return code: 0
Removing Chapter_5_examples/example_1__simple_counter/obj_dir directory...
Chapter_5_examples/example_1__simple_counter/obj_dir removed successfully.

0
```

**Example 2: Data Register**

data_register - Module instantiation examples (named vs positional connections)

```
// data_register.sv
module data_register (
    input  logic       clk,     // Clock signal
    input  logic       rst_n,   // Active-low reset
    input  logic       enable,  // Enable signal
    input  logic [7:0] data_in, // 8-bit input data
    output logic [7:0] data_out // 8-bit output data
);

    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            data_out <= 8'h00;  // Reset to zero
        end else if (enable) begin
            data_out <= data_in; // Load input data when enabled
        end
        // If enable is low, data_out retains its value
    end

    // Display messages for simulation
    initial begin
        $display();
        $display("Data Register module initialized");
        $display("  - 8-bit register with enable control");
        $display("  - Active-low reset");
        $display();
    end

endmodule
```

```
// data_register_testbench.sv
module data_register_testbench;

    // Testbench signals
    logic       tb_clk;
    logic       tb_rst_n;
    logic       tb_enable;
    logic [7:0] tb_data_in;
    logic [7:0] tb_data_out_named;
    logic [7:0] tb_data_out_positional;

    // Clock generation
    initial begin
        tb_clk = 0;
        forever #5 tb_clk = ~tb_clk; // 10ns period clock
    end

    // ========================================================================
    // EXAMPLE 1: NAMED PORT CONNECTIONS (Recommended method)
    // ========================================================================
    data_register DUT_NAMED_CONNECTIONS (
        .clk(tb_clk),
        .rst_n(tb_rst_n),
        .enable(tb_enable),
        .data_in(tb_data_in),
```

```
        .data_out(tb_data_out_named)
    );

    // ========================================================================
    // EXAMPLE 2: POSITIONAL PORT CONNECTIONS (Order matters!)
    // ========================================================================
    data_register DUT_POSITIONAL_CONNECTIONS (
        tb_clk,                     // clk (1st port)
        tb_rst_n,                   // rst_n (2nd port)
        tb_enable,                  // enable (3rd port)
        tb_data_in,                 // data_in (4th port)
        tb_data_out_positional      // data_out (5th port)
    );

    // Test stimulus
    initial begin
        // Dump waves
        $dumpfile("data_register_testbench.vcd");
        $dumpvars(0, data_register_testbench);

        $display("=== Data Register Testbench Started ===");
        $display("Testing both named and positional instantiation methods");
        $display();

        // Initialize signals
        tb_rst_n = 0;
        tb_enable = 0;
        tb_data_in = 8'h00;

        // Reset sequence
        $display("Time %0t: Applying reset", $time);
        #20;
        tb_rst_n = 1;
        $display("Time %0t: Releasing reset", $time);
        #10;

        // Test 1: Load data with enable
        tb_enable = 1;
        tb_data_in = 8'hAA;
        $display("Time %0t: Loading data 0x%02h with enable=1", $time, tb_data_in);
        #20;
        $display("Time %0t: Named output = 0x%02h, Positional output = 0x%02h",
                 $time, tb_data_out_named, tb_data_out_positional);

        // Test 2: Change input with enable disabled
        tb_enable = 0;
        tb_data_in = 8'h55;
        $display("Time %0t: Changing input to 0x%02h with enable=0",
                 $time, tb_data_in);
        #20;
        $display("Time %0t: Named output = 0x%02h, Positional output = 0x%02h",
                 $time, tb_data_out_named, tb_data_out_positional);
        $display("  -> Data should remain unchanged (0xAA)");

        // Test 3: Enable again to load new data
```

```
        tb_enable = 1;
        $display("Time %0t: Re-enabling to load new data 0x%02h",
                 $time, tb_data_in);
        #20;
        $display("Time %0t: Named output = 0x%02h, Positional output = 0x%02h",
                 $time, tb_data_out_named, tb_data_out_positional);

        $display();
        $display("=== Both instantiation methods produce identical results ===");
        $display("Named connections: More readable and less error-prone");
        $display("Positional connections: Shorter but order-dependent");
        $display();

        #10;
        $finish;
    end

endmodule
```

```
Verilator Simulation Output:
================================================================================

Data Register module initialized
  - 8-bit register with enable control
  - Active-low reset


Data Register module initialized
  - 8-bit register with enable control
  - Active-low reset

=== Data Register Testbench Started ===
Testing both named and positional instantiation methods

Time 0: Applying reset
Time 20: Releasing reset
Time 30: Loading data 0xaa with enable=1
Time 50: Named output = 0xaa, Positional output = 0xaa
Time 50: Changing input to 0x55 with enable=0
Time 70: Named output = 0xaa, Positional output = 0xaa
  -> Data should remain unchanged (0xAA)
Time 70: Re-enabling to load new data 0x55
Time 90: Named output = 0x55, Positional output = 0x55

=== Both instantiation methods produce identical results ===
Named connections: More readable and less error-prone
Positional connections: Shorter but order-dependent

================================================================================
Process finished with return code: 0
Removing Chapter_5_examples/example_2__data_register/obj_dir directory...
Chapter_5_examples/example_2__data_register/obj_dir removed successfully.

0
```

## Port Declarations and Directions

SystemVerilog provides several ways to declare module ports, offering more flexibility than traditional Verilog.

### Port Directions

```
module port_example (
    input  logic        clk,          // Input port
    output logic        valid,        // Output port
    inout  wire         bidir_signal, // Bidirectional port
    ref    int          shared_var    // Reference port (SystemVerilog)
);
```

### ANSI-Style Port Declarations (Recommended)

```
module counter #(
    parameter int WIDTH = 8
) (
    input  logic             clk,
    input  logic             reset_n,
    input  logic             enable,
    input  logic             load,
    input  logic [WIDTH-1:0] load_value,
    output logic [WIDTH-1:0] count,
    output logic             overflow
);

    logic [WIDTH-1:0] count_reg;

    always_ff @(posedge clk or negedge reset_n) begin
        if (!reset_n) begin
            count_reg <= '0;
            overflow <= 1'b0;
        end else if (load) begin
            count_reg <= load_value;
            overflow <= 1'b0;
        end else if (enable) begin
            {overflow, count_reg} <= count_reg + 1'b1;
        end
    end

    assign count = count_reg;

endmodule
```

### Non-ANSI Style (Legacy)

```
module counter (clk, reset_n, enable, count);
    parameter WIDTH = 8;

    input             clk;
    input             reset_n;
```

```
    input             enable;
    output [WIDTH-1:0] count;

    // Port declarations separate from module header
endmodule
```

**Advanced Port Features**

**Interface Ports**:

```
module processor (
    input logic clk,
    input logic reset_n,
    memory_if.master mem_bus,  // Interface port
    axi4_if.slave    axi_port
);
```

**Unpacked Array Ports**:

```
module multi_port (
    input  logic [7:0] data_in [0:3],   // Array of inputs
    output logic [7:0] data_out [0:3]   // Array of outputs
);
```

**Example 3: Port Direction**

port_direction - Different port types (input, output, inout, ref)

```
// port_direction.sv
module port_direction (
    // INPUT ports - data flows INTO the module
    input  logic       clk,        // Clock signal
    input  logic       reset_n,    // Active-low reset
    input  logic [3:0] data_in,    // 4-bit input data

    // OUTPUT ports - data flows OUT of the module
    output logic [3:0] data_out,   // 4-bit output data
    output logic       valid_out,  // Output valid flag

    // INOUT ports - bidirectional data flow
    inout  wire  [7:0] bus_data,   // 8-bit bidirectional bus

    // REF ports - pass by reference (SystemVerilog only)
    ref    logic [1:0] ref_counter // Reference to external counter
);

    // Internal signals
    logic       bus_enable;    // Controls bus direction
    logic [7:0] internal_bus;  // Internal bus data

    // Simple counter for demonstration
    always_ff @(posedge clk or negedge reset_n) begin
        if (!reset_n) begin
```

```
            data_out <= 4'h0;
            valid_out <= 1'b0;
            internal_bus <= 8'h00;
            bus_enable <= 1'b0;
        end else begin
            // Process input data (double it)
            data_out <= data_in * 2;
            valid_out <= |data_in;  // Valid if any input bit is set

            // Bus control logic
            bus_enable <= data_in[0];  // Use LSB to control bus
            if (data_in[0]) begin
                internal_bus <= {4'h0, data_in};  // Drive bus
            end

            // Modify ref_counter directly (demonstrates ref port)
            if (valid_out) begin
                ref_counter <= ref_counter + 1;
            end
        end
    end

    // Bidirectional bus driver
    assign bus_data = bus_enable ? internal_bus : 8'hZZ;

    // Display messages for simulation
    initial begin
        $display();
        $display("Port Direction Demo module initialized");
        $display("  - INPUT: clk, reset_n, data_in");
        $display("  - OUTPUT: data_out, valid_out");
        $display("  - INOUT: bus_data (bidirectional)");
        $display("  - REF: ref_counter (pass by reference)");
        $display();
    end

endmodule
```

```
// port_direction_testbench.sv
module port_direction_testbench;

    // Testbench signals for different port types
    logic       tb_clk;
    logic       tb_reset_n;
    logic [3:0] tb_data_in;        // INPUT port connection
    logic [3:0] tb_data_out;       // OUTPUT port connection
    logic       tb_valid_out;      // OUTPUT port connection
    wire  [7:0] tb_bus_data;       // INOUT port connection (wire)
    logic [1:0] tb_ref_counter;    // REF port connection

    // Additional signals for testing inout port
    logic       tb_bus_drive;
    logic [7:0] tb_bus_value;
```

```
    // Clock generation
    initial begin
        tb_clk = 0;
        forever #5 tb_clk = ~tb_clk; // 10ns period clock
    end

    // ================================================================
    // MODULE INSTANTIATION - Demonstrating all port types
    // ================================================================
    port_direction DUT (
        // INPUT ports - we drive these from testbench
        .clk(tb_clk),
        .reset_n(tb_reset_n),
        .data_in(tb_data_in),

        // OUTPUT ports - module drives these to testbench
        .data_out(tb_data_out),
        .valid_out(tb_valid_out),

        // INOUT port - bidirectional connection
        .bus_data(tb_bus_data),

        // REF port - direct reference to testbench variable
        .ref_counter(tb_ref_counter)
    );

    // Bidirectional bus driver for testing
    assign tb_bus_data = tb_bus_drive ? tb_bus_value : 8'hZZ;

    // Test stimulus
    initial begin
        // Dump waves
        $dumpfile("port_direction_testbench.vcd");
        $dumpvars(0, port_direction_testbench);

        $display("=== Port Direction Demo Testbench Started ===");
        $display();

        // Initialize signals
        tb_reset_n = 0;
        tb_data_in = 4'h0;
        tb_ref_counter = 2'b00;
        tb_bus_drive = 0;
        tb_bus_value = 8'h00;

        // Reset sequence
        $display("Time %0t: Applying reset", $time);
        #20;
        tb_reset_n = 1;
        $display("Time %0t: Releasing reset", $time);
        #10;

        // Test 1: INPUT and OUTPUT ports
        $display("--- Testing INPUT and OUTPUT ports ---");
        tb_data_in = 4'h3;
```

```
        #20;
        $display("Time %0t: INPUT data_in = %d, OUTPUT data_out = %d, valid = %b",
                 $time, tb_data_in, tb_data_out, tb_valid_out);
        $display("  -> Module doubles input: %d * 2 = %d",
                 tb_data_in, tb_data_out);

        // Test 2: REF port modification
        $display("--- Testing REF port ---");
        $display("Time %0t: REF counter before = %d", $time, tb_ref_counter);
        tb_data_in = 4'h5;
        #20;
        $display("Time %0t: REF counter after = %d", $time, tb_ref_counter);
        $display("  -> Module modified ref_counter directly!");

        // Test 3: INOUT port - Module driving bus
        $display("--- Testing INOUT port - Module driving ---");
        tb_data_in = 4'h7;  // LSB = 1, so module will drive bus
        #20;
        $display("Time %0t: Module driving bus_data = 0x%02h",
                 $time, tb_bus_data);
        $display("  -> Module drives bus when data_in[0] = 1");

        // Test 4: INOUT port - Testbench driving bus
        $display("--- Testing INOUT port - Testbench driving ---");
        tb_data_in = 4'h6;  // LSB = 0, so module tri-states bus
        tb_bus_drive = 1;
        tb_bus_value = 8'hFF;
        #20;
        $display("Time %0t: Testbench driving bus_data = 0x%02h",
                 $time, tb_bus_data);
        $display("  -> Testbench drives bus when module tri-states");

        // Test 5: Show final state
        tb_bus_drive = 0;
        tb_data_in = 4'h0;
        #20;
        $display();
        $display("=== Final Results ===");
        $display("INPUT ports: Driven by testbench to module");
        $display("OUTPUT ports: Driven by module to testbench");
        $display("INOUT ports: Bidirectional, either side can drive");
        $display("REF ports: Direct reference, module can modify testbench vars");
        $display("Final ref_counter value: %d", tb_ref_counter);
        $display();

        #10;
        $finish;
    end

endmodule
```

```
Verilator Simulation Output:
================================================================================

Port Direction Demo module initialized
  - INPUT: clk, reset_n, data_in
```

```
  - OUTPUT: data_out, valid_out
  - INOUT: bus_data (bidirectional)
  - REF: ref_counter (pass by reference)

=== Port Direction Demo Testbench Started ===

Time 0: Applying reset
Time 20: Releasing reset
--- Testing INPUT and OUTPUT ports ---
Time 50: INPUT data_in =  3, OUTPUT data_out =  6, valid = 1
  -> Module doubles input:  3 * 2 =  6
--- Testing REF port ---
Time 50: REF counter before = 1
Time 70: REF counter after = 3
  -> Module modified ref_counter directly!
--- Testing INOUT port - Module driving ---
Time 90: Module driving bus_data = 0x07
  -> Module drives bus when data_in[0] = 1
--- Testing INOUT port - Testbench driving ---
Time 110: Testbench driving bus_data = 0xff
  -> Testbench drives bus when module tri-states

=== Final Results ===
INPUT ports: Driven by testbench to module
OUTPUT ports: Driven by module to testbench
INOUT ports: Bidirectional, either side can drive
REF ports: Direct reference, module can modify testbench vars
Final ref_counter value: 0

================================================================================
Process finished with return code: 0
Removing Chapter_5_examples/example_3__port_direction/obj_dir directory...
Chapter_5_examples/example_3__port_direction/obj_dir removed successfully.

0
```

**Example 4: Array Port Module**

array_port_module - Unpacked array ports demonstration

```
// array_port_module.sv - Fixed version with proper bit widths
module array_port_module #(
    parameter int DATA_WIDTH = 8,               // Width of each data element
    parameter int NUM_CHANNELS = 4              // Number of parallel channels
) (
    // Clock and reset
    input  logic                                clk,
    input  logic                                reset_n,
    input  logic                                enable,

    // Unpacked array input ports - multiple data channels
    input  logic [DATA_WIDTH-1:0]          data_in [NUM_CHANNELS],     // Array of input data
    input  logic                                valid_in [NUM_CHANNELS],    // Array of valid sig

    // Unpacked array output ports
    output logic [DATA_WIDTH-1:0]          data_out [NUM_CHANNELS],    // Array of output dat
```

```
    output logic                            valid_out [NUM_CHANNELS],   // Array of valid out

    // Status arrays
    output logic [7:0]                      channel_count [NUM_CHANNELS],   // Per-channel co
    output logic [DATA_WIDTH-1:0]           channel_max [NUM_CHANNELS],     // Maximum value p
    output logic [DATA_WIDTH-1:0]           channel_min [NUM_CHANNELS]      // Minimum value p
);

    // Generate block for per-channel processing
    generate
        genvar ch;
        for (ch = 0; ch < NUM_CHANNELS; ch++) begin : channel_proc

            // Per-channel registers
            logic [DATA_WIDTH-1:0] data_reg;
            logic                  valid_reg;
            logic [7:0]            counter;
            logic [DATA_WIDTH-1:0] max_val;
            logic [DATA_WIDTH-1:0] min_val;

            // Per-channel processing logic
            always_ff @(posedge clk or negedge reset_n) begin
                if (!reset_n) begin
                    data_reg <= '0;
                    valid_reg <= 1'b0;
                    counter <= '0;
                    max_val <= '0;
                    min_val <= '1; // All 1s for minimum initialization
                end else if (enable) begin
                    // Process input data
                    data_reg <= data_in[ch];
                    valid_reg <= valid_in[ch];

                    // Update statistics when valid data arrives
                    if (valid_in[ch]) begin
                        counter <= counter + 1'b1;

                        // Track max/min values
                        if (data_in[ch] > max_val) begin
                            max_val <= data_in[ch];
                        end
                        if (data_in[ch] < min_val) begin
                            min_val <= data_in[ch];
                        end
                    end
                end
            end

            // Connect internal registers to output arrays
            assign data_out[ch] = data_reg;
            assign valid_out[ch] = valid_reg;
            assign channel_count[ch] = counter;
            assign channel_max[ch] = max_val;
            assign channel_min[ch] = min_val;
        end
```

```
    endgenerate

    // Cross-channel operations using array manipulation
    logic [DATA_WIDTH-1:0] sum_all_channels;
    logic [$clog2(NUM_CHANNELS+1)-1:0] active_channels;

    // Calculate sum and count of active channels
    always_comb begin
        sum_all_channels = '0;
        active_channels = '0;

        for (int i = 0; i < NUM_CHANNELS; i++) begin
            if (valid_out[i]) begin
                sum_all_channels += data_out[i];
                active_channels++;
            end
        end
    end

    // Simple array comparison example - fixed width truncation
    logic [DATA_WIDTH-1:0] highest_value;
    logic [$clog2(NUM_CHANNELS)-1:0] highest_channel;

    always_comb begin
        highest_value = '0;
        highest_channel = '0;

        for (int i = 0; i < NUM_CHANNELS; i++) begin
            if (valid_out[i] && (data_out[i] > highest_value)) begin
                highest_value = data_out[i];
                highest_channel = i[$clog2(NUM_CHANNELS)-1:0]; // Fixed: proper width convers
            end
        end
    end

    // Parameter validation and info display
    initial begin
        assert (NUM_CHANNELS > 0 && NUM_CHANNELS <= 16)
            else $error("NUM_CHANNELS must be between 1 and 16");
        assert (DATA_WIDTH > 0 && DATA_WIDTH <= 16)
            else $error("DATA_WIDTH must be between 1 and 16");

        $display("Simple Array Port Module Initialized:");
        $display("  DATA_WIDTH = %0d bits", DATA_WIDTH);
        $display("  NUM_CHANNELS = %0d", NUM_CHANNELS);
    end

endmodule
```

```
// array_port_module_testbench.sv - Fixed testbench with proper array syntax
module array_port_module_testbench;

    // Testbench parameters
    localparam int CLK_PERIOD = 10;
```

```
    localparam int TB_DATA_WIDTH = 8;
    localparam int TB_NUM_CHANNELS = 4;

    // Testbench signals - using arrays to match DUT
    logic                                clk;
    logic                                reset_n;
    logic                                enable;

    // Array signals
    logic [TB_DATA_WIDTH-1:0]          data_in [TB_NUM_CHANNELS];
    logic                               valid_in [TB_NUM_CHANNELS];
    logic [TB_DATA_WIDTH-1:0]          data_out [TB_NUM_CHANNELS];
    logic                               valid_out [TB_NUM_CHANNELS];
    logic [7:0]                        channel_count [TB_NUM_CHANNELS];
    logic [TB_DATA_WIDTH-1:0]          channel_max [TB_NUM_CHANNELS];
    logic [TB_DATA_WIDTH-1:0]          channel_min [TB_NUM_CHANNELS];

    // Clock generation
    initial begin
        clk = 0;
        forever #(CLK_PERIOD/2) clk = ~clk;
    end

    // DUT instantiation
    array_port_module #(
        .DATA_WIDTH(TB_DATA_WIDTH),
        .NUM_CHANNELS(TB_NUM_CHANNELS)
    ) dut (
        .clk(clk),
        .reset_n(reset_n),
        .enable(enable),
        .data_in(data_in),
        .valid_in(valid_in),
        .data_out(data_out),
        .valid_out(valid_out),
        .channel_count(channel_count),
        .channel_max(channel_max),
        .channel_min(channel_min)
    );

    // Task to send data to a specific channel
    task automatic send_data(input int channel, input logic [TB_DATA_WIDTH-1:0] data, input l
        if (channel < TB_NUM_CHANNELS) begin
            data_in[channel] = data;
            valid_in[channel] = valid;
        end
    endtask

    // Task to display all channel status
    task automatic display_status();
        $display("\n=== Channel Status at time %0t ===", $time);
        for (int i = 0; i < TB_NUM_CHANNELS; i++) begin
            $display("Channel %0d: out=%3d, valid=%b, count=%3d, max=%3d, min=%3d",
                    i, data_out[i], valid_out[i], channel_count[i],
                    channel_max[i], channel_min[i]);
```

```
        end
        $display("====================================\n");
    endtask

    // Test stimulus
    initial begin
        // Initialize VCD dump
        $dumpfile("array_port_module_testbench.vcd");
        $dumpvars(0, array_port_module_testbench);

        $display("=== Simple Array Port Module Testbench Started ===");
        $display("Testing with %0d channels, %0d-bit data width", TB_NUM_CHANNELS, TB_DATA_WI

        // Initialize all signals
        reset_n = 0;
        enable = 0;

        // Initialize input arrays
        for (int i = 0; i < TB_NUM_CHANNELS; i++) begin
            data_in[i] = 0;
            valid_in[i] = 0;
        end

        // Reset sequence
        $display("\nPhase 1: Reset Test");
        #(CLK_PERIOD * 2);
        reset_n = 1;
        enable = 1;
        #(CLK_PERIOD);
        display_status();

        // Phase 2: Send data to all channels simultaneously
        $display("Phase 2: Send Data to All Channels");
        for (int cycle = 0; cycle < 5; cycle++) begin
            $display("Cycle %0d:", cycle);
            for (int ch = 0; ch < TB_NUM_CHANNELS; ch++) begin
                send_data(ch, 8'(cycle * 20 + ch * 5), 1'b1);
            end
            #(CLK_PERIOD);
            display_status();
        end

        // Phase 3: Send data to specific channels only
        $display("Phase 3: Selective Channel Operation");

        // Clear all valid signals first
        for (int i = 0; i < TB_NUM_CHANNELS; i++) begin
            valid_in[i] = 0;
        end
        #(CLK_PERIOD);

        // Send to channel 0 and 2 only
        send_data(0, 8'd150, 1'b1);
        send_data(2, 8'd200, 1'b1);
        #(CLK_PERIOD);
```

```
        display_status();

        // Send to channel 1 and 3 only
        for (int i = 0; i < TB_NUM_CHANNELS; i++) begin
            valid_in[i] = 0;
        end
        send_data(1, 8'd75, 1'b1);
        send_data(3, 8'd250, 1'b1);
        #(CLK_PERIOD);
        display_status();

        // Phase 4: Test min/max tracking
        $display("Phase 4: Min/Max Value Tracking");

        // Use the pre-declared test_val variable
        for (int cycle = 0; cycle < 5; cycle++) begin
            case (cycle)
                0: test_val = 50;
                1: test_val = 200;
                2: test_val = 10;
                3: test_val = 240;
                4: test_val = 30;
                default: test_val = 0;
            endcase

            $display("Sending test value %0d to all channels", test_val);
            for (int ch = 0; ch < TB_NUM_CHANNELS; ch++) begin
                send_data(ch, test_val, 1'b1);
            end
            #(CLK_PERIOD);
            display_status();
        end

        // Phase 5: Test enable control
        $display("Phase 5: Enable Control Test");

        // Disable the module
        enable = 0;
        for (int ch = 0; ch < TB_NUM_CHANNELS; ch++) begin
            send_data(ch, 8'd100, 1'b1);
        end
        #(CLK_PERIOD * 2);
        $display("With enable=0 (should not process new data):");
        display_status();

        // Re-enable
        enable = 1;
        #(CLK_PERIOD);
        $display("With enable=1 (should process new data):");
        display_status();

        // Phase 6: Counter test
        $display("Phase 6: Counter Test - Send Multiple Values");
        for (int burst = 0; burst < 3; burst++) begin
            for (int ch = 0; ch < TB_NUM_CHANNELS; ch++) begin
```

```
                send_data(ch, 8'(burst * 50 + ch * 10), 1'b1);
            end
            #(CLK_PERIOD);
        end
        display_status();

        // Phase 7: Final test with mixed valid signals
        $display("Phase 7: Mixed Valid Signals Test");
        for (int cycle = 0; cycle < 4; cycle++) begin
            for (int ch = 0; ch < TB_NUM_CHANNELS; ch++) begin
                // Alternate valid signals in a pattern - fixed width truncation
                logic valid = ((cycle + ch) % 2) == 1;
                send_data(ch, 8'(cycle * 30 + ch), valid);
            end
            #(CLK_PERIOD);
            if (cycle % 2 == 1) display_status();
        end

        // Final status
        $display("Final Status:");
        // Clear all inputs
        for (int i = 0; i < TB_NUM_CHANNELS; i++) begin
            valid_in[i] = 0;
        end
        #(CLK_PERIOD);
        display_status();

        $display("=== Array Port Module Testbench Completed ===");
        $display("Total simulation time: %0t", $time);
        $finish;
    end

    // Monitor for interesting events
    always @(posedge clk) begin
        if (reset_n && enable) begin
            // Count how many channels are active
            int active_count = 0;
            for (int i = 0; i < TB_NUM_CHANNELS; i++) begin
                if (valid_in[i]) active_count++;
            end

            if (active_count == TB_NUM_CHANNELS) begin
                $display("*** ALL %0d CHANNELS ACTIVE at time %0t ***", TB_NUM_CHANNELS, $tim
            end
        end
    end

    // Example of array manipulation in testbench
    logic [TB_DATA_WIDTH-1:0] sum_outputs;
    logic [TB_DATA_WIDTH-1:0] max_output;
    int valid_output_count;

    // Test value variable for Phase 4
    logic [TB_DATA_WIDTH-1:0] test_val;
```

```
    // Calculate statistics from output arrays
    always_comb begin
        sum_outputs = '0;
        max_output = '0;
        valid_output_count = 0;

        for (int i = 0; i < TB_NUM_CHANNELS; i++) begin
            if (valid_out[i]) begin
                sum_outputs += data_out[i];
                valid_output_count++;
                if (data_out[i] > max_output) begin
                    max_output = data_out[i];
                end
            end
        end
    end

endmodule
```

```
Verilator Simulation Output:
================================================================================
Simple Array Port Module Initialized:
  DATA_WIDTH = 8 bits
  NUM_CHANNELS = 4
=== Simple Array Port Module Testbench Started ===
Testing with 4 channels, 8-bit data width

Phase 1: Reset Test

=== Channel Status at time 30 ===
Channel 0: out=  0, valid=0, count=  0, max=  0, min=255
Channel 1: out=  0, valid=0, count=  0, max=  0, min=255
Channel 2: out=  0, valid=0, count=  0, max=  0, min=255
Channel 3: out=  0, valid=0, count=  0, max=  0, min=255
====================================

Phase 2: Send Data to All Channels
Cycle 0:
*** ALL 4 CHANNELS ACTIVE at time 35 ***

=== Channel Status at time 40 ===
Channel 0: out=  0, valid=1, count=  1, max=  0, min=  0
Channel 1: out=  5, valid=1, count=  1, max=  5, min=  5
Channel 2: out= 10, valid=1, count=  1, max= 10, min= 10
Channel 3: out= 15, valid=1, count=  1, max= 15, min= 15
====================================

Cycle 1:
*** ALL 4 CHANNELS ACTIVE at time 45 ***

=== Channel Status at time 50 ===
Channel 0: out= 20, valid=1, count=  2, max= 20, min=  0
Channel 1: out= 25, valid=1, count=  2, max= 25, min=  5
Channel 2: out= 30, valid=1, count=  2, max= 30, min= 10
Channel 3: out= 35, valid=1, count=  2, max= 35, min= 15
====================================
```

```
Cycle 2:
*** ALL 4 CHANNELS ACTIVE at time 55 ***

=== Channel Status at time 60 ===
Channel 0: out= 40, valid=1, count=  3, max= 40, min=  0
Channel 1: out= 45, valid=1, count=  3, max= 45, min=  5
Channel 2: out= 50, valid=1, count=  3, max= 50, min= 10
Channel 3: out= 55, valid=1, count=  3, max= 55, min= 15
====================================

Cycle 3:
*** ALL 4 CHANNELS ACTIVE at time 65 ***

=== Channel Status at time 70 ===
Channel 0: out= 60, valid=1, count=  4, max= 60, min=  0
Channel 1: out= 65, valid=1, count=  4, max= 65, min=  5
Channel 2: out= 70, valid=1, count=  4, max= 70, min= 10
Channel 3: out= 75, valid=1, count=  4, max= 75, min= 15
====================================

Cycle 4:
*** ALL 4 CHANNELS ACTIVE at time 75 ***

=== Channel Status at time 80 ===
Channel 0: out= 80, valid=1, count=  5, max= 80, min=  0
Channel 1: out= 85, valid=1, count=  5, max= 85, min=  5
Channel 2: out= 90, valid=1, count=  5, max= 90, min= 10
Channel 3: out= 95, valid=1, count=  5, max= 95, min= 15
====================================

Phase 3: Selective Channel Operation

=== Channel Status at time 100 ===
Channel 0: out=150, valid=1, count=  6, max=150, min=  0
Channel 1: out= 85, valid=0, count=  5, max= 85, min=  5
Channel 2: out=200, valid=1, count=  6, max=200, min= 10
Channel 3: out= 95, valid=0, count=  5, max= 95, min= 15
====================================


=== Channel Status at time 110 ===
Channel 0: out=150, valid=0, count=  6, max=150, min=  0
Channel 1: out= 75, valid=1, count=  6, max= 85, min=  5
Channel 2: out=200, valid=0, count=  6, max=200, min= 10
Channel 3: out=250, valid=1, count=  6, max=250, min= 15
====================================

Phase 4: Min/Max Value Tracking
Sending test value 50 to all channels
*** ALL 4 CHANNELS ACTIVE at time 115 ***

=== Channel Status at time 120 ===
Channel 0: out= 50, valid=1, count=  7, max=150, min=  0
Channel 1: out= 50, valid=1, count=  7, max= 85, min=  5
Channel 2: out= 50, valid=1, count=  7, max=200, min= 10
```

```
Channel 3: out= 50, valid=1, count=  7, max=250, min= 15
====================================

Sending test value 200 to all channels
*** ALL 4 CHANNELS ACTIVE at time 125 ***

=== Channel Status at time 130 ===
Channel 0: out=200, valid=1, count=  8, max=200, min=  0
Channel 1: out=200, valid=1, count=  8, max=200, min=  5
Channel 2: out=200, valid=1, count=  8, max=200, min= 10
Channel 3: out=200, valid=1, count=  8, max=250, min= 15
====================================

Sending test value 10 to all channels
*** ALL 4 CHANNELS ACTIVE at time 135 ***

=== Channel Status at time 140 ===
Channel 0: out= 10, valid=1, count=  9, max=200, min=  0
Channel 1: out= 10, valid=1, count=  9, max=200, min=  5
Channel 2: out= 10, valid=1, count=  9, max=200, min= 10
Channel 3: out= 10, valid=1, count=  9, max=250, min= 10
====================================

Sending test value 240 to all channels
*** ALL 4 CHANNELS ACTIVE at time 145 ***

=== Channel Status at time 150 ===
Channel 0: out=240, valid=1, count= 10, max=240, min=  0
Channel 1: out=240, valid=1, count= 10, max=240, min=  5
Channel 2: out=240, valid=1, count= 10, max=240, min= 10
Channel 3: out=240, valid=1, count= 10, max=250, min= 10
====================================

Sending test value 30 to all channels
*** ALL 4 CHANNELS ACTIVE at time 155 ***

=== Channel Status at time 160 ===
Channel 0: out= 30, valid=1, count= 11, max=240, min=  0
Channel 1: out= 30, valid=1, count= 11, max=240, min=  5
Channel 2: out= 30, valid=1, count= 11, max=240, min= 10
Channel 3: out= 30, valid=1, count= 11, max=250, min= 10
====================================

Phase 5: Enable Control Test
With enable=0 (should not process new data):

=== Channel Status at time 180 ===
Channel 0: out= 30, valid=1, count= 11, max=240, min=  0
Channel 1: out= 30, valid=1, count= 11, max=240, min=  5
Channel 2: out= 30, valid=1, count= 11, max=240, min= 10
Channel 3: out= 30, valid=1, count= 11, max=250, min= 10
====================================

*** ALL 4 CHANNELS ACTIVE at time 185 ***
With enable=1 (should process new data):
```

```
=== Channel Status at time 190 ===
Channel 0: out=100, valid=1, count= 12, max=240, min=  0
Channel 1: out=100, valid=1, count= 12, max=240, min=  5
Channel 2: out=100, valid=1, count= 12, max=240, min= 10
Channel 3: out=100, valid=1, count= 12, max=250, min= 10
====================================

Phase 6: Counter Test - Send Multiple Values
*** ALL 4 CHANNELS ACTIVE at time 195 ***
*** ALL 4 CHANNELS ACTIVE at time 205 ***
*** ALL 4 CHANNELS ACTIVE at time 215 ***

=== Channel Status at time 220 ===
Channel 0: out=100, valid=1, count= 15, max=240, min=  0
Channel 1: out=110, valid=1, count= 15, max=240, min=  5
Channel 2: out=120, valid=1, count= 15, max=240, min= 10
Channel 3: out=130, valid=1, count= 15, max=250, min= 10
====================================

Phase 7: Mixed Valid Signals Test

=== Channel Status at time 240 ===
Channel 0: out= 30, valid=1, count= 16, max=240, min=  0
Channel 1: out= 31, valid=0, count= 16, max=240, min=  1
Channel 2: out= 32, valid=1, count= 16, max=240, min= 10
Channel 3: out= 33, valid=0, count= 16, max=250, min=  3
====================================


=== Channel Status at time 260 ===
Channel 0: out= 90, valid=1, count= 17, max=240, min=  0
Channel 1: out= 91, valid=0, count= 17, max=240, min=  1
Channel 2: out= 92, valid=1, count= 17, max=240, min= 10
Channel 3: out= 93, valid=0, count= 17, max=250, min=  3
====================================

Final Status:

=== Channel Status at time 270 ===
Channel 0: out= 90, valid=0, count= 17, max=240, min=  0
Channel 1: out= 91, valid=0, count= 17, max=240, min=  1
Channel 2: out= 92, valid=0, count= 17, max=240, min= 10
Channel 3: out= 93, valid=0, count= 17, max=250, min=  3
====================================

=== Array Port Module Testbench Completed ===
Total simulation time: 270
================================================================================
Process finished with return code: 0
Removing Chapter_5_examples/example_4__array_port_module/obj_dir directory...
Chapter_5_examples/example_4__array_port_module/obj_dir removed successfully.

0
```

## Parameters and Localparams

Parameters provide a way to create configurable, reusable modules. They allow customization at instantiation time.

### Parameter Types

```
module parameterized_module #(
    // Type parameters
    parameter type DATA_TYPE = logic [31:0],
    parameter type ADDR_TYPE = logic [15:0],

    // Value parameters
    parameter int DATA_WIDTH = 32,
    parameter int ADDR_WIDTH = 16,
    parameter int DEPTH = 1024,

    // String parameters
    parameter string MODE = "NORMAL",

    // Real parameters
    parameter real FREQUENCY = 100.0
) (
    input  logic      clk,
    input  DATA_TYPE  data_in,
    input  ADDR_TYPE  address,
    output DATA_TYPE  data_out
);
```

### Localparam Usage

Localparams are parameters that cannot be overridden during instantiation. They're typically used for derived values.

```
module memory #(
    parameter int DATA_WIDTH = 32,
    parameter int ADDR_WIDTH = 10
) (
    input  logic                    clk,
    input  logic                    we,
    input  logic [ADDR_WIDTH-1:0]   addr,
    input  logic [DATA_WIDTH-1:0]   wdata,
    output logic [DATA_WIDTH-1:0]   rdata
);

    // Localparams derived from parameters
    localparam int DEPTH = 2**ADDR_WIDTH;
    localparam int BYTES_PER_WORD = DATA_WIDTH / 8;

    logic [DATA_WIDTH-1:0] mem_array [0:DEPTH-1];

    always_ff @(posedge clk) begin
        if (we)
            mem_array[addr] <= wdata;
```

```
        rdata <= mem_array[addr];
    end

endmodule
```

**Parameter Override Examples**

```
// Override during instantiation
memory #(
    .DATA_WIDTH(64),
    .ADDR_WIDTH(12)
) ram_inst (
    .clk(clk),
    .we(write_enable),
    .addr(address),
    .wdata(write_data),
    .rdata(read_data)
);

// Using defparam (not recommended)
defparam ram_inst.DATA_WIDTH = 64;
defparam ram_inst.ADDR_WIDTH = 12;
```

**Example 5: Configurable Memory**

configurable_memory - Parameter types (type, value, string, real parameters)

```
// configurable_memory.sv - Corrected Version
module configurable_memory #(
    // Integer parameter - memory depth
    parameter int MEMORY_DEPTH = 1024,

    // Type parameter - data width
    parameter type DATA_TYPE = bit [7:0],

    // String parameter - memory identification
    parameter string MEMORY_NAME = "DEFAULT_MEM",

    // Real parameter - access delay in ns
    parameter real ACCESS_DELAY_NS = 2.5
)(
    input  logic clk,
    input  logic reset_n,
    input  logic write_enable,
    input  logic [$clog2(MEMORY_DEPTH)-1:0] address,
    input  DATA_TYPE write_data,
    output DATA_TYPE read_data
);

    // Memory array using parameterized type and depth
    DATA_TYPE memory_array [MEMORY_DEPTH];

    initial begin
```

```
        $display();
        $display("=== Memory Configuration ===");
        $display("Memory Name: %s", MEMORY_NAME);
        $display("Memory Depth: %0d words", MEMORY_DEPTH);
        $display("Data Width: %0d bits", $bits(DATA_TYPE));
        $display("Access Delay: %0.1f ns", ACCESS_DELAY_NS);
        $display("Address Width: %0d bits", $clog2(MEMORY_DEPTH));
        $display("============================");
    end

    // Memory write operation
    always_ff @(posedge clk or negedge reset_n) begin
        if (!reset_n) begin
            // Initialize memory to zero on reset
            for (int i = 0; i < MEMORY_DEPTH; i++) begin
                memory_array[i] <= '0;
            end
        end else if (write_enable) begin
            memory_array[address] <= write_data;
            $display("[%s] Write: Addr=0x%h, Data=0x%h",
                    MEMORY_NAME, address, write_data);
        end
    end

    // Memory read operation - combinational read
    always_comb begin
        read_data = memory_array[address];
    end

endmodule
```

```
// configurable_memory_testbench.sv
module configurable_memory_testbench;

    // Testbench signals
    logic clk;
    logic reset_n;

    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk;  // 10ns period = 100MHz
    end

    // === Instance 1: Small 8-bit memory ===
    logic small_we;
    logic [3:0] small_addr;
    bit [7:0] small_wdata, small_rdata;

    configurable_memory #(
        .MEMORY_DEPTH(16),
        .DATA_TYPE(bit [7:0]),
        .MEMORY_NAME("SMALL_8BIT_MEM"),
        .ACCESS_DELAY_NS(1.0)
```

```
    ) small_memory (
        .clk(clk),
        .reset_n(reset_n),
        .write_enable(small_we),
        .address(small_addr),
        .write_data(small_wdata),
        .read_data(small_rdata)
    );

    // === Instance 2: Large 16-bit memory ===
    logic large_we;
    logic [9:0] large_addr;
    bit [15:0] large_wdata, large_rdata;

    configurable_memory #(
        .MEMORY_DEPTH(1024),
        .DATA_TYPE(bit [15:0]),
        .MEMORY_NAME("LARGE_16BIT_MEM"),
        .ACCESS_DELAY_NS(3.5)
    ) large_memory (
        .clk(clk),
        .reset_n(reset_n),
        .write_enable(large_we),
        .address(large_addr),
        .write_data(large_wdata),
        .read_data(large_rdata)
    );

    // === Instance 3: Custom 32-bit memory ===
    logic custom_we;
    logic [7:0] custom_addr;
    bit [31:0] custom_wdata, custom_rdata;

    configurable_memory #(
        .MEMORY_DEPTH(256),
        .DATA_TYPE(bit [31:0]),
        .MEMORY_NAME("CUSTOM_32BIT_CACHE"),
        .ACCESS_DELAY_NS(0.8)
    ) custom_memory (
        .clk(clk),
        .reset_n(reset_n),
        .write_enable(custom_we),
        .address(custom_addr),
        .write_data(custom_wdata),
        .read_data(custom_rdata)
    );

    initial begin
        // Dump waves
        $dumpfile("configurable_memory_testbench.vcd");
        $dumpvars(0, configurable_memory_testbench);

        $display("\n=== Starting Configurable Memory Test ===");

        // Initialize ALL signals
```

```
        reset_n = 0;
        small_we = 0; large_we = 0; custom_we = 0;
        small_addr = 0; small_wdata = 0;
        large_addr = 0; large_wdata = 0;
        custom_addr = 0; custom_wdata = 0;

        // Reset sequence
        #20 reset_n = 1;
        #10;

        // Test small memory - Write and Read Back
        $display("\n--- Testing Small 8-bit Memory ---");
        small_we = 1;
        small_addr = 4'h5;
        small_wdata = 8'hAB;
        #10;

        small_we = 0;
        #10;
        $display("Small Memory Read: Addr=0x%h, Data=0x%h",
                 small_addr, small_rdata);

        // Write multiple locations
        small_we = 1;
        small_addr = 4'h2; small_wdata = 8'h11; #10;
        small_addr = 4'h7; small_wdata = 8'h22; #10;
        small_addr = 4'hF; small_wdata = 8'hFF; #10;

        // Read back all locations
        small_we = 0;
        small_addr = 4'h2; #10;
        $display("Small Memory Read: Addr=0x%h, Data=0x%h",
                 small_addr, small_rdata);
        small_addr = 4'h5; #10;
        $display("Small Memory Read: Addr=0x%h, Data=0x%h",
                 small_addr, small_rdata);
        small_addr = 4'h7; #10;
        $display("Small Memory Read: Addr=0x%h, Data=0x%h",
                 small_addr, small_rdata);
        small_addr = 4'hF; #10;
        $display("Small Memory Read: Addr=0x%h, Data=0x%h",
                 small_addr, small_rdata);

        // Test large memory - Write and Read Back
        $display("\n--- Testing Large 16-bit Memory ---");
        large_we = 1;
        large_addr = 10'h123;
        large_wdata = 16'hDEAD;
        #10;

        large_we = 0;
        #10;
        $display("Large Memory Read: Addr=0x%h, Data=0x%h",
                 large_addr, large_rdata);
```

```
        // Write pattern to multiple locations
        large_we = 1;
        large_addr = 10'h000; large_wdata = 16'h1234; #10;
        large_addr = 10'h100; large_wdata = 16'h5678; #10;
        large_addr = 10'h200; large_wdata = 16'h9ABC; #10;
        large_addr = 10'h3FF; large_wdata = 16'hBEEF; #10;

        // Read back pattern
        large_we = 0;
        large_addr = 10'h000; #10;
        $display("Large Memory Read: Addr=0x%h, Data=0x%h",
                 large_addr, large_rdata);
        large_addr = 10'h100; #10;
        $display("Large Memory Read: Addr=0x%h, Data=0x%h",
                 large_addr, large_rdata);
        large_addr = 10'h123; #10;
        $display("Large Memory Read: Addr=0x%h, Data=0x%h",
                 large_addr, large_rdata);
        large_addr = 10'h200; #10;
        $display("Large Memory Read: Addr=0x%h, Data=0x%h",
                 large_addr, large_rdata);
        large_addr = 10'h3FF; #10;
        $display("Large Memory Read: Addr=0x%h, Data=0x%h",
                 large_addr, large_rdata);

        // Test custom memory - Write and Read Back
        $display("\n--- Testing Custom 32-bit Memory ---");
        custom_we = 1;
        custom_addr = 8'h42;
        custom_wdata = 32'hCAFEBABE;
        #10;

        custom_we = 0;
        #10;
        $display("Custom Memory Read: Addr=0x%h, Data=0x%h",
                 custom_addr, custom_rdata);

        // Write test pattern
        custom_we = 1;
        custom_addr = 8'h00; custom_wdata = 32'h12345678; #10;
        custom_addr = 8'h10; custom_wdata = 32'h87654321; #10;
        custom_addr = 8'h80; custom_wdata = 32'hA5A5A5A5; #10;
        custom_addr = 8'hFF; custom_wdata = 32'h5A5A5A5A; #10;

        // Read back test pattern
        custom_we = 0;
        custom_addr = 8'h00; #10;
        $display("Custom Memory Read: Addr=0x%h, Data=0x%h",
                 custom_addr, custom_rdata);
        custom_addr = 8'h10; #10;
        $display("Custom Memory Read: Addr=0x%h, Data=0x%h",
                 custom_addr, custom_rdata);
        custom_addr = 8'h42; #10;
        $display("Custom Memory Read: Addr=0x%h, Data=0x%h",
                 custom_addr, custom_rdata);
```

```
        custom_addr = 8'h80; #10;
        $display("Custom Memory Read: Addr=0x%h, Data=0x%h",
                 custom_addr, custom_rdata);
        custom_addr = 8'hFF; #10;
        $display("Custom Memory Read: Addr=0x%h, Data=0x%h",
                 custom_addr, custom_rdata);

        // Memory integrity test - verify data persistence
        $display("\n--- Memory Integrity Test ---");
        custom_addr = 8'h42; #10;
        if (custom_rdata == 32'hCAFEBABE)
            $display("Memory integrity PASSED - Original data preserved");
        else
            $display("Memory integrity FAILED - Expected 0xCAFEBABE, " +
                    "got 0x%h", custom_rdata);

        // Cross-memory integrity test
        $display("\n--- Cross-Memory Integrity Test ---");
        small_addr = 4'h5; #10;
        if (small_rdata == 8'hAB)
            $display("Small memory integrity PASSED");
        else
            $display("Small memory integrity FAILED - Expected 0xAB, " +
                    "got 0x%h", small_rdata);

        large_addr = 10'h123; #10;
        if (large_rdata == 16'hDEAD)
            $display("Large memory integrity PASSED");
        else
            $display("Large memory integrity FAILED - Expected 0xDEAD, " +
                    "got 0x%h", large_rdata);

        // Test parameter effects and memory utilization
        $display("\n--- Parameter Summary & Memory Utilization ---");
        $display("Small Memory: %0d x %0d bits (Total: %0d bits)",
                 16, 8, 16*8);
        $display("Large Memory: %0d x %0d bits (Total: %0d bits)",
                 1024, 16, 1024*16);
        $display("Custom Memory: %0d x %0d bits (Total: %0d bits)",
                 256, 32, 256*32);

        // Test write/read performance with different delays
        $display("\n--- Access Delay Comparison ---");
        $display("Small Memory Delay: 1.0 ns");
        $display("Large Memory Delay: 3.5 ns");
        $display("Custom Memory Delay: 0.8 ns");

        #50;
        $display();
        $display("=== Configurable Memory Test Complete ===");
        $display();
        $finish;
    end

endmodule
```

```
Verilator Simulation Output:
================================================================================

=== Memory Configuration ===
Memory Name: SMALL_8BIT_MEM
Memory Depth: 16 words
Data Width: 8 bits
Access Delay: 1.0 ns
Address Width: 4 bits
============================

=== Memory Configuration ===
Memory Name: LARGE_16BIT_MEM
Memory Depth: 1024 words
Data Width: 16 bits
Access Delay: 3.5 ns
Address Width: 10 bits
============================

=== Memory Configuration ===
Memory Name: CUSTOM_32BIT_CACHE
Memory Depth: 256 words
Data Width: 32 bits
Access Delay: 0.8 ns
Address Width: 8 bits
============================

=== Starting Configurable Memory Test ===

--- Testing Small 8-bit Memory ---
[SMALL_8BIT_MEM] Write: Addr=0x5, Data=0xab
Small Memory Read: Addr=0x5, Data=0xab
[SMALL_8BIT_MEM] Write: Addr=0x2, Data=0x11
[SMALL_8BIT_MEM] Write: Addr=0x7, Data=0x22
[SMALL_8BIT_MEM] Write: Addr=0xf, Data=0xff
Small Memory Read: Addr=0x2, Data=0x11
Small Memory Read: Addr=0x5, Data=0xab
Small Memory Read: Addr=0x7, Data=0x22
Small Memory Read: Addr=0xf, Data=0xff

--- Testing Large 16-bit Memory ---
[LARGE_16BIT_MEM] Write: Addr=0x123, Data=0xdead
Large Memory Read: Addr=0x123, Data=0xdead
[LARGE_16BIT_MEM] Write: Addr=0x000, Data=0x1234
[LARGE_16BIT_MEM] Write: Addr=0x100, Data=0x5678
[LARGE_16BIT_MEM] Write: Addr=0x200, Data=0x9abc
[LARGE_16BIT_MEM] Write: Addr=0x3ff, Data=0xbeef
Large Memory Read: Addr=0x000, Data=0x1234
Large Memory Read: Addr=0x100, Data=0x5678
Large Memory Read: Addr=0x123, Data=0xdead
Large Memory Read: Addr=0x200, Data=0x9abc
Large Memory Read: Addr=0x3ff, Data=0xbeef

--- Testing Custom 32-bit Memory ---
[CUSTOM_32BIT_CACHE] Write: Addr=0x42, Data=0xcafebabe
Custom Memory Read: Addr=0x42, Data=0xcafebabe
```

```
[CUSTOM_32BIT_CACHE] Write: Addr=0x00, Data=0x12345678
[CUSTOM_32BIT_CACHE] Write: Addr=0x10, Data=0x87654321
[CUSTOM_32BIT_CACHE] Write: Addr=0x80, Data=0xa5a5a5a5
[CUSTOM_32BIT_CACHE] Write: Addr=0xff, Data=0x5a5a5a5a
Custom Memory Read: Addr=0x00, Data=0x12345678
Custom Memory Read: Addr=0x10, Data=0x87654321
Custom Memory Read: Addr=0x42, Data=0xcafebabe
Custom Memory Read: Addr=0x80, Data=0xa5a5a5a5
Custom Memory Read: Addr=0xff, Data=0x5a5a5a5a

--- Memory Integrity Test ---
Memory integrity PASSED - Original data preserved

--- Cross-Memory Integrity Test ---
Small memory integrity PASSED
Large memory integrity PASSED

--- Parameter Summary & Memory Utilization ---
Small Memory: 16 x 8 bits (Total: 128 bits)
Large Memory: 1024 x 16 bits (Total: 16384 bits)
Custom Memory: 256 x 32 bits (Total: 8192 bits)

--- Access Delay Comparison ---
Small Memory Delay: 1.0 ns
Large Memory Delay: 3.5 ns
Custom Memory Delay: 0.8 ns

=== Configurable Memory Test Complete ===

================================================================================
Process finished with return code: 0
Removing Chapter_5_examples/example_5__configurable_memory/obj_dir directory...
Chapter_5_examples/example_5__configurable_memory/obj_dir removed successfully.
0
```

**Example 6: Parameter Override**

parameter_override - Parameter override during instantiation

```
// parameter_override.sv
module parameter_override #(
  parameter WIDTH = 4,           // Default width parameter
  parameter DEPTH = 8            // Default depth parameter
)();

  initial begin
    $display();
    $display("Design Parameters:");
    $display("  WIDTH = %0d", WIDTH);
    $display("  DEPTH = %0d", DEPTH);
    $display("  Total capacity = %0d bits", WIDTH * DEPTH);
    $display();
  end

endmodule
```

```systemverilog
// parameter_override_testbench.sv
module parameter_override_testbench;

  // Instance 1: Use default parameters
  parameter_override default_params();

  // Instance 2: Override WIDTH parameter only
  parameter_override #(.WIDTH(16)) width_override();

  // Instance 3: Override both parameters
  parameter_override #(.WIDTH(32), .DEPTH(16)) both_override();

  // Instance 4: Override parameters in different order
  parameter_override #(.DEPTH(64), .WIDTH(8)) reordered_override();

  initial begin
    // Dump waves
    $dumpfile("parameter_override_testbench.vcd");
    $dumpvars(0, parameter_override_testbench);

    $display("=== Parameter Override Example ===");
    $display();

    #1; // Let all instances initialize

    $display("All instances created with different parameter values!");
    $display();

    #10;
    $finish;
  end

endmodule
```

```
Verilator Simulation Output:
================================================================================

Design Parameters:
  WIDTH = 4
  DEPTH = 8
  Total capacity = 32 bits


Design Parameters:
  WIDTH = 16
  DEPTH = 8
  Total capacity = 128 bits


Design Parameters:
  WIDTH = 32
  DEPTH = 16
  Total capacity = 512 bits


Design Parameters:
```

```
  WIDTH = 8
  DEPTH = 64
  Total capacity = 512 bits

=== Parameter Override Example ===

All instances created with different parameter values!

================================================================================
Process finished with return code: 0
Removing Chapter_5_examples/example_6__parameter_override/obj_dir directory...
Chapter_5_examples/example_6__parameter_override/obj_dir removed successfully.

0
```

**Example 7: Localparam Calculator**

localparam_calculator - Localparam usage for derived values

```
// localparam_calculator.sv
module localparam_calculator #(
  parameter DATA_WIDTH = 8,        // User-configurable parameter
  parameter NUM_ELEMENTS = 16      // User-configurable parameter
)(
  input  logic clk,
  input  logic reset
);

  // Localparams - calculated from parameters, cannot be overridden
  localparam ADDR_WIDTH = $clog2(NUM_ELEMENTS);           // Address width needed
  localparam TOTAL_BITS = DATA_WIDTH * NUM_ELEMENTS;      // Total memory bits
  localparam MAX_VALUE = (1 << DATA_WIDTH) - 1;          // Maximum data value
  localparam HALF_ELEMENTS = NUM_ELEMENTS / 2;           // Half the elements

  // Example memory array using derived values
  logic [DATA_WIDTH-1:0] memory [NUM_ELEMENTS-1:0];
  logic [ADDR_WIDTH-1:0] address;

  initial begin
    $display();
    $display("=== Localparam Calculator ===");
    $display("Input Parameters:");
    $display("  DATA_WIDTH = %0d", DATA_WIDTH);
    $display("  NUM_ELEMENTS = %0d", NUM_ELEMENTS);
    $display();
    $display("Calculated Localparams:");
    $display("  ADDR_WIDTH = %0d bits", ADDR_WIDTH);
    $display("  TOTAL_BITS = %0d bits", TOTAL_BITS);
    $display("  MAX_VALUE = %0d", MAX_VALUE);
    $display("  HALF_ELEMENTS = %0d", HALF_ELEMENTS);
    $display();
    $display("Memory array: [%0d:0] memory [%0d:0]", DATA_WIDTH-1, NUM_ELEMENTS-1);
    $display("Address signal: [%0d:0] address", ADDR_WIDTH-1);
    $display();
  end
```

```
  // Simple counter to demonstrate address usage
  always_ff @(posedge clk or posedge reset) begin
    if (reset)
      address <= '0;
    else
      address <= address + 1;  // Will automatically wrap at NUM_ELEMENTS
  end

endmodule
```

```
// localparam_calculator_testbench.sv
module localparam_calculator_testbench;

  logic clk, reset;

  // Instance 1: Default parameters (8-bit, 16 elements)
  localparam_calculator default_calc(
    .clk(clk),
    .reset(reset)
  );

  // Instance 2: 4-bit data, 32 elements
  localparam_calculator #(
    .DATA_WIDTH(4),
    .NUM_ELEMENTS(32)
  ) small_wide_calc(
    .clk(clk),
    .reset(reset)
  );

  // Instance 3: 16-bit data, 8 elements
  localparam_calculator #(
    .DATA_WIDTH(16),
    .NUM_ELEMENTS(8)
  ) wide_narrow_calc(
    .clk(clk),
    .reset(reset)
  );

  // Clock generation
  initial begin
    clk = 0;
    forever #5 clk = ~clk;
  end

  initial begin
    // Dump waves
    $dumpfile("localparam_calculator_testbench.vcd");
    $dumpvars(0, localparam_calculator_testbench);

    $display("=== Localparam Calculator Testbench ===");
    $display("Demonstrating how localparams are calculated from parameters");
    $display();
```

```
    // Reset sequence
    reset = 1;
    #10;
    reset = 0;

    // Run for a few clock cycles
    #100;

    $display("Notice how each instance has different localparam values");
    $display("based on their parameter overrides, but localparams cannot");
    $display("be overridden directly - they are always calculated!");
    $display();

    $finish;
  end

endmodule
```

```
Verilator Simulation Output:
================================================================================

=== Localparam Calculator ===
Input Parameters:
  DATA_WIDTH = 8
  NUM_ELEMENTS = 16

Calculated Localparams:
  ADDR_WIDTH = 4 bits
  TOTAL_BITS = 128 bits
  MAX_VALUE = 255
  HALF_ELEMENTS = 8

Memory array: [7:0] memory [15:0]
Address signal: [3:0] address


=== Localparam Calculator ===
Input Parameters:
  DATA_WIDTH = 4
  NUM_ELEMENTS = 32

Calculated Localparams:
  ADDR_WIDTH = 5 bits
  TOTAL_BITS = 128 bits
  MAX_VALUE = 15
  HALF_ELEMENTS = 16

Memory array: [3:0] memory [31:0]
Address signal: [4:0] address


=== Localparam Calculator ===
Input Parameters:
  DATA_WIDTH = 16
  NUM_ELEMENTS = 8
```

```
Calculated Localparams:
  ADDR_WIDTH = 3 bits
  TOTAL_BITS = 128 bits
  MAX_VALUE = 65535
  HALF_ELEMENTS = 4

Memory array: [15:0] memory [7:0]
Address signal: [2:0] address

=== Localparam Calculator Testbench ===
Demonstrating how localparams are calculated from parameters

Notice how each instance has different localparam values
based on their parameter overrides, but localparams cannot
be overridden directly - they are always calculated!

================================================================================
Process finished with return code: 0
Removing Chapter_5_examples/example_7__localparam_calculator/obj_dir directory...
Chapter_5_examples/example_7__localparam_calculator/obj_dir removed successfully.
0
```

## Generate Blocks

Generate blocks allow you to create repetitive hardware structures and conditional compilation based on parameters.

### Generate For Loops

```
module parallel_adder #(
    parameter int WIDTH = 32,
    parameter int STAGES = 4
) (
    input  logic [WIDTH-1:0] a,
    input  logic [WIDTH-1:0] b,
    input  logic             cin,
    output logic [WIDTH-1:0] sum,
    output logic             cout
);

    localparam int BITS_PER_STAGE = WIDTH / STAGES;

    logic [STAGES:0] carry;
    assign carry[0] = cin;
    assign cout = carry[STAGES];

    // Generate multiple adder stages
    generate
        for (genvar i = 0; i < STAGES; i++) begin : adder_stage
            logic [BITS_PER_STAGE-1:0] stage_sum;
            logic                      stage_cout;

            full_adder #(.WIDTH(BITS_PER_STAGE)) fa_inst (
                .a(a[i*BITS_PER_STAGE +: BITS_PER_STAGE]),
```

```
                .b(b[i*BITS_PER_STAGE +: BITS_PER_STAGE]),
                .cin(carry[i]),
                .sum(stage_sum),
                .cout(stage_cout)
            );

            assign sum[i*BITS_PER_STAGE +: BITS_PER_STAGE] = stage_sum;
            assign carry[i+1] = stage_cout;
        end
    endgenerate

endmodule
```

**Generate If-Else**

```
module configurable_memory #(
    parameter int    DATA_WIDTH = 32,
    parameter int    ADDR_WIDTH = 10,
    parameter string MEMORY_TYPE = "BLOCK"  // "BLOCK" or "DISTRIBUTED"
) (
    input  logic                    clk,
    input  logic                    we,
    input  logic [ADDR_WIDTH-1:0]   addr,
    input  logic [DATA_WIDTH-1:0]   wdata,
    output logic [DATA_WIDTH-1:0]   rdata
);

    localparam int DEPTH = 2**ADDR_WIDTH;

    generate
        if (MEMORY_TYPE == "BLOCK") begin : block_memory
            // Use block RAM
            logic [DATA_WIDTH-1:0] mem [0:DEPTH-1];

            always_ff @(posedge clk) begin
                if (we)
                    mem[addr] <= wdata;
                rdata <= mem[addr];
            end

        end else if (MEMORY_TYPE == "DISTRIBUTED") begin : dist_memory
            // Use distributed RAM
            logic [DATA_WIDTH-1:0] mem [0:DEPTH-1];

            always_ff @(posedge clk) begin
                if (we)
                    mem[addr] <= wdata;
            end

            assign rdata = mem[addr];  // Combinational read

        end else begin : error_memory
            // Generate compile-time error for invalid parameter
            initial begin
```

```
                $error("Invalid MEMORY_TYPE parameter: %s", MEMORY_TYPE);
            end
        end
    endgenerate

endmodule
```

**Generate Case**

```
module priority_encoder #(
    parameter int WIDTH = 8
) (
    input  logic [WIDTH-1:0] data_in,
    output logic [$clog2(WIDTH)-1:0] encoded_out,
    output logic valid
);

    generate
        case (WIDTH)
            4: begin : enc_4bit
                always_comb begin
                    casez (data_in)
                        4'b???1: {valid, encoded_out} = {1'b1, 2'd0};
                        4'b??10: {valid, encoded_out} = {1'b1, 2'd1};
                        4'b?100: {valid, encoded_out} = {1'b1, 2'd2};
                        4'b1000: {valid, encoded_out} = {1'b1, 2'd3};
                        default: {valid, encoded_out} = {1'b0, 2'd0};
                    endcase
                end
            end

            8: begin : enc_8bit
                // Implementation for 8-bit encoder
                always_comb begin
                    casez (data_in)
                        8'b???????1: {valid, encoded_out} = {1'b1, 3'd0};
                        8'b??????10: {valid, encoded_out} = {1'b1, 3'd1};
                        8'b?????100: {valid, encoded_out} = {1'b1, 3'd2};
                        8'b????1000: {valid, encoded_out} = {1'b1, 3'd3};
                        8'b???10000: {valid, encoded_out} = {1'b1, 3'd4};
                        8'b??100000: {valid, encoded_out} = {1'b1, 3'd5};
                        8'b?1000000: {valid, encoded_out} = {1'b1, 3'd6};
                        8'b10000000: {valid, encoded_out} = {1'b1, 3'd7};
                        default:     {valid, encoded_out} = {1'b0, 3'd0};
                    endcase
                end
            end

            default: begin : enc_generic
                // Generic implementation for other widths
                always_comb begin
                    encoded_out = '0;
                    valid = 1'b0;
                    for (int i = 0; i < WIDTH; i++) begin
```

```
                        if (data_in[i]) begin
                            encoded_out = i[$clog2(WIDTH)-1:0];
                            valid = 1'b1;
                            break;
                        end
                    end
                end
            end
        endcase
    endgenerate

endmodule
```

**Example 8: Parallel Adder Generator**

parallel_adder_generator - Generate for loops creating repetitive structures

```
// parallel_adder_generator.sv

// Simple full adder module
module full_adder (
  input  logic a, b, cin,
  output logic sum, cout
);
  assign {cout, sum} = a + b + cin;
endmodule

// Parallel adder using generate for loops
module parallel_adder_generator #(
  parameter WIDTH = 4
)(
  input  logic [WIDTH-1:0] a, b,
  input  logic cin,
  output logic [WIDTH-1:0] sum,
  output logic cout
);

  // Internal carry signals
  logic [WIDTH:0] carry;

  // Connect input carry
  assign carry[0] = cin;

  // Connect output carry
  assign cout = carry[WIDTH];

  // Generate block for creating multiple full adders
  genvar i;
  generate
    for (i = 0; i < WIDTH; i++) begin : adder_stage
      full_adder fa_inst (
        .a(a[i]),
        .b(b[i]),
        .cin(carry[i]),
```

```
        .sum(sum[i]),
        .cout(carry[i+1])
      );

      // Display which stage is being generated
      initial begin
        $display("Generated adder stage %0d", i);
      end
    end
  endgenerate

  initial begin
    $display();
    $display("=== Parallel Adder Generator ===");
    $display("WIDTH = %0d bits", WIDTH);
    $display("Generated %0d full adder stages", WIDTH);
    $display();
  end

endmodule
```

```
// parallel_adder_generator_testbench.sv
module parallel_adder_generator_testbench;

  // Test signals for 4-bit adder
  logic [3:0] a4, b4, sum4;
  logic cin4, cout4;

  // Test signals for 8-bit adder
  logic [7:0] a8, b8, sum8;
  logic cin8, cout8;

  // Test signals for 2-bit adder
  logic [1:0] a2, b2, sum2;
  logic cin2, cout2;

  // Instance 1: 4-bit adder (default)
  parallel_adder_generator #(.WIDTH(4)) adder_4bit (
    .a(a4), .b(b4), .cin(cin4),
    .sum(sum4), .cout(cout4)
  );

  // Instance 2: 8-bit adder
  parallel_adder_generator #(.WIDTH(8)) adder_8bit (
    .a(a8), .b(b8), .cin(cin8),
    .sum(sum8), .cout(cout8)
  );

  // Instance 3: 2-bit adder
  parallel_adder_generator #(.WIDTH(2)) adder_2bit (
    .a(a2), .b(b2), .cin(cin2),
    .sum(sum2), .cout(cout2)
  );
```

```
    initial begin
      // Dump waves
      $dumpfile("parallel_adder_generator_testbench.vcd");
      $dumpvars(0, parallel_adder_generator_testbench);

      $display("=== Parallel Adder Generator Testbench ===");
      $display("Testing different width adders generated with for loops");
      $display();

      // Wait for generation messages
      #1;

      // Test 4-bit adder
      $display("--- Testing 4-bit Adder ---");
      a4 = 4'b0101; b4 = 4'b0011; cin4 = 1'b0;
      #1;
      $display("4-bit: %b + %b + %b = %b (carry=%b)", a4, b4, cin4, sum4, cout4);
      $display("4-bit: %0d + %0d + %0d = %0d (carry=%0d)", a4, b4, cin4, sum4, cout4);

      // Test 8-bit adder
      $display("--- Testing 8-bit Adder ---");
      a8 = 8'b10101010; b8 = 8'b01010101; cin8 = 1'b1;
      #1;
      $display("8-bit: %b + %b + %b = %b (carry=%b)", a8, b8, cin8, sum8, cout8);
      $display("8-bit: %0d + %0d + %0d = %0d (carry=%0d)", a8, b8, cin8, sum8, cout8);

      // Test 2-bit adder
      $display("--- Testing 2-bit Adder ---");
      a2 = 2'b11; b2 = 2'b01; cin2 = 1'b1;
      #1;
      $display("2-bit: %b + %b + %b = %b (carry=%b)", a2, b2, cin2, sum2, cout2);
      $display("2-bit: %0d + %0d + %0d = %0d (carry=%0d)", a2, b2, cin2, sum2, cout2);

      $display();
      $display("All adders created using the same generate for loop!");
      $display("Each instance automatically generates the correct number of stages.");

      #10;
      $finish;
    end

  endmodule
```

```
Verilator Simulation Output:
================================================================================
Generated adder stage 0
Generated adder stage 1
Generated adder stage 2
Generated adder stage 3

=== Parallel Adder Generator ===
WIDTH = 4 bits
Generated 4 full adder stages

Generated adder stage 0
Generated adder stage 1
```

```
Generated adder stage 2
Generated adder stage 3
Generated adder stage 4
Generated adder stage 5
Generated adder stage 6
Generated adder stage 7

=== Parallel Adder Generator ===
WIDTH = 8 bits
Generated 8 full adder stages

Generated adder stage 0
Generated adder stage 1

=== Parallel Adder Generator ===
WIDTH = 2 bits
Generated 2 full adder stages

=== Parallel Adder Generator Testbench ===
Testing different width adders generated with for loops

--- Testing 4-bit Adder ---
4-bit: 0101 + 0011 + 0 = 1000 (carry=0)
4-bit: 5 + 3 + 0 = 8 (carry=0)
--- Testing 8-bit Adder ---
8-bit: 10101010 + 01010101 + 1 = 00000000 (carry=1)
8-bit: 170 + 85 + 1 = 0 (carry=1)
--- Testing 2-bit Adder ---
2-bit: 11 + 01 + 1 = 01 (carry=1)
2-bit: 3 + 1 + 1 = 1 (carry=1)

All adders created using the same generate for loop!
Each instance automatically generates the correct number of stages.
================================================================================
Process finished with return code: 0
Removing Chapter_5_examples/example_8__parallel_adder_generator/obj_dir directory...
Chapter_5_examples/example_8__parallel_adder_generator/obj_dir removed successfully.
0
```

### Example 9: Memory Type Selector

memory_type_selector - Generate if-else for conditional compilation

```systemverilog
// memory_type_selector.sv
module memory_type_selector #(
  parameter string MEMORY_TYPE = "SRAM",    // "SRAM", "DRAM", or "ROM"
  parameter DATA_WIDTH = 8,
  parameter ADDR_WIDTH = 4
)(
  input  logic clk,
  input  logic reset,
  input  logic [ADDR_WIDTH-1:0] addr,
  input  logic [DATA_WIDTH-1:0] data_in,
  input  logic write_en,
  output logic [DATA_WIDTH-1:0] data_out
```

```
);

  localparam DEPTH = 1 << ADDR_WIDTH;

  // Generate different memory types based on parameter
  generate
    if (MEMORY_TYPE == "SRAM") begin : sram_memory
      // SRAM - Simple synchronous memory
      logic [DATA_WIDTH-1:0] mem [DEPTH-1:0];

      always_ff @(posedge clk) begin
        if (write_en)
          mem[addr] <= data_in;
        data_out <= mem[addr];
      end

      initial begin
        $display("Generated SRAM memory type");
        $display("  - Synchronous read/write");
        $display("  - %0d x %0d bits", DEPTH, DATA_WIDTH);
      end

    end else if (MEMORY_TYPE == "DRAM") begin : dram_memory
      // DRAM - With refresh requirement (simplified simulation)
      logic [DATA_WIDTH-1:0] mem [DEPTH-1:0];
      logic [7:0] refresh_counter;

      always_ff @(posedge clk) begin
        if (reset) begin
          refresh_counter <= 0;
          data_out <= '0;
        end else begin
          // Refresh counter
          refresh_counter <= refresh_counter + 1;

          // Memory access
          if (write_en)
            mem[addr] <= data_in;
          else
            data_out <= mem[addr];
        end
      end

      initial begin
        $display("Generated DRAM memory type");
        $display("  - Requires refresh simulation");
        $display("  - %0d x %0d bits", DEPTH, DATA_WIDTH);
      end

    end else if (MEMORY_TYPE == "ROM") begin : rom_memory
      // ROM - Read-only memory with initialization
      logic [DATA_WIDTH-1:0] mem [DEPTH-1:0];

      // Initialize ROM with pattern
      initial begin
```

```
        for (int i = 0; i < DEPTH; i++) begin
          mem[i] = DATA_WIDTH'(i * 3);  // Properly sized pattern
        end
      end

      always_ff @(posedge clk) begin
        data_out <= mem[addr];
        // Ignore write_en for ROM
      end

      initial begin
        $display("Generated ROM memory type");
        $display("  - Read-only, pre-initialized");
        $display("  - %0d x %0d bits", DEPTH, DATA_WIDTH);
        $display("  - Pattern: data[i] = i * 3");
      end

    end else begin : invalid_memory
      // Default case for invalid memory type
      assign data_out = '0;

      initial begin
        $display("ERROR: Invalid memory type '%s'", MEMORY_TYPE);
        $display("Valid types: SRAM, DRAM, ROM");
      end
    end
  endgenerate

  initial begin
    $display();
    $display("=== Memory Type Selector ===");
    $display("Selected memory type: %s", MEMORY_TYPE);
    $display("Address width: %0d bits", ADDR_WIDTH);
    $display("Data width: %0d bits", DATA_WIDTH);
    $display();
  end

endmodule
```

```
// memory_type_selector_testbench.sv
module memory_type_selector_testbench;

  logic clk, reset, write_en;
  logic [3:0] addr;
  logic [7:0] data_in, data_out_sram, data_out_dram, data_out_rom;

  // Instance 1: SRAM memory
  memory_type_selector #(
    .MEMORY_TYPE("SRAM"),
    .DATA_WIDTH(8),
    .ADDR_WIDTH(4)
  ) sram_inst (
    .clk(clk), .reset(reset),
    .addr(addr), .data_in(data_in), .write_en(write_en),
```

```
    .data_out(data_out_sram)
);

// Instance 2: DRAM memory
memory_type_selector #(
    .MEMORY_TYPE("DRAM"),
    .DATA_WIDTH(8),
    .ADDR_WIDTH(4)
) dram_inst (
    .clk(clk), .reset(reset),
    .addr(addr), .data_in(data_in), .write_en(write_en),
    .data_out(data_out_dram)
);

// Instance 3: ROM memory
memory_type_selector #(
    .MEMORY_TYPE("ROM"),
    .DATA_WIDTH(8),
    .ADDR_WIDTH(4)
) rom_inst (
    .clk(clk), .reset(reset),
    .addr(addr), .data_in(data_in), .write_en(write_en),
    .data_out(data_out_rom)
);

// Instance 4: Invalid memory type
memory_type_selector #(
    .MEMORY_TYPE("FLASH"),
    .DATA_WIDTH(8),
    .ADDR_WIDTH(4)
) invalid_inst (
    .clk(clk), .reset(reset),
    .addr(addr), .data_in(data_in), .write_en(write_en),
    .data_out()  // Don't care about output
);

// Clock generation
initial begin
    clk = 0;
    forever #5 clk = ~clk;
end

initial begin
    // Dump waves
    $dumpfile("memory_type_selector_testbench.vcd");
    $dumpvars(0, memory_type_selector_testbench);

    $display("=== Memory Type Selector Testbench ===");
    $display("Testing conditional compilation with generate if-else");
    $display();

    // Reset sequence
    reset = 1;
    addr = 0;
    data_in = 0;
```

```
    write_en = 0;
    #20;
    reset = 0;
    #10;

    // Test write operation
    $display("--- Testing Write Operations ---");
    write_en = 1;
    addr = 4'h5;
    data_in = 8'hAA;
    #10;
    write_en = 0;
    #10;

    $display("SRAM[5] = 0x%02X", data_out_sram);
    $display("DRAM[5] = 0x%02X", data_out_dram);
    $display("ROM[5] = 0x%02X (should be %0d)", data_out_rom, 5*3);

    // Test read operation
    $display("--- Testing Read Operations ---");
    addr = 4'h3;
    #10;

    $display("SRAM[3] = 0x%02X", data_out_sram);
    $display("DRAM[3] = 0x%02X", data_out_dram);
    $display("ROM[3] = 0x%02X (should be %0d)", data_out_rom, 3*3);

    $display();
    $display("Each instance compiled different memory logic based on MEMORY_TYPE!");
    $display("Same module, different hardware generated!");

    #50;
    $finish;
  end

endmodule
```

```
Verilator Simulation Output:
================================================================================
Generated SRAM memory type
  - Synchronous read/write
  - 16 x 8 bits

=== Memory Type Selector ===
Selected memory type: SRAM
Address width: 4 bits
Data width: 8 bits

Generated DRAM memory type
  - Requires refresh simulation
  - 16 x 8 bits

=== Memory Type Selector ===
Selected memory type: DRAM
Address width: 4 bits
Data width: 8 bits
```

```
Generated ROM memory type
  - Read-only, pre-initialized
  - 16 x 8 bits
  - Pattern: data[i] = i * 3

=== Memory Type Selector ===
Selected memory type: ROM
Address width: 4 bits
Data width: 8 bits

ERROR: Invalid memory type 'FLASH'
Valid types: SRAM, DRAM, ROM

=== Memory Type Selector ===
Selected memory type: FLASH
Address width: 4 bits
Data width: 8 bits

=== Memory Type Selector Testbench ===
Testing conditional compilation with generate if-else

--- Testing Write Operations ---
SRAM[5] = 0xaa
DRAM[5] = 0xaa
ROM[5] = 0x0f (should be 15)
--- Testing Read Operations ---
SRAM[3] = 0x00
DRAM[3] = 0x00
ROM[3] = 0x09 (should be 9)

Each instance compiled different memory logic based on MEMORY_TYPE!
Same module, different hardware generated!
================================================================================
Process finished with return code: 0
Removing Chapter_5_examples/example_9__memory_type_selector/obj_dir directory...
Chapter_5_examples/example_9__memory_type_selector/obj_dir removed successfully.
0
```

**Example 10: Encoder Width Selector**

encoder_width_selector - Generate case for parameter-based selection

```
// encoder_width_selector.sv
module encoder_width_selector #(
  parameter INPUT_WIDTH = 4    // Valid values: 2, 4, 8, 16
)(
  input  logic [INPUT_WIDTH-1:0] data_in,
  output logic [$clog2(INPUT_WIDTH)-1:0] encoded_out,
  output logic valid_out
);

  localparam OUTPUT_WIDTH = $clog2(INPUT_WIDTH);

  // Generate different encoder implementations based on input width
```

```
  generate
    case (INPUT_WIDTH)

      2: begin : encoder_2to1
        // 2-to-1 encoder
        always_comb begin
          case (data_in)
            2'b01: begin encoded_out = 1'b0; valid_out = 1'b1; end
            2'b10: begin encoded_out = 1'b1; valid_out = 1'b1; end
            default: begin encoded_out = 1'b0; valid_out = 1'b0; end
          endcase
        end

        initial $display("Generated 2-to-1 encoder (2 inputs -> 1 output)");
      end

      4: begin : encoder_4to2
        // 4-to-2 encoder
        always_comb begin
          casez (data_in)
            4'b0001: begin encoded_out = 2'b00; valid_out = 1'b1; end
            4'b0010: begin encoded_out = 2'b01; valid_out = 1'b1; end
            4'b0100: begin encoded_out = 2'b10; valid_out = 1'b1; end
            4'b1000: begin encoded_out = 2'b11; valid_out = 1'b1; end
            default: begin encoded_out = 2'b00; valid_out = 1'b0; end
          endcase
        end

        initial $display("Generated 4-to-2 encoder (4 inputs -> 2 outputs)");
      end

      8: begin : encoder_8to3
        // 8-to-3 encoder
        always_comb begin
          casez (data_in)
            8'b00000001: begin encoded_out = 3'b000; valid_out = 1'b1; end
            8'b00000010: begin encoded_out = 3'b001; valid_out = 1'b1; end
            8'b00000100: begin encoded_out = 3'b010; valid_out = 1'b1; end
            8'b00001000: begin encoded_out = 3'b011; valid_out = 1'b1; end
            8'b00010000: begin encoded_out = 3'b100; valid_out = 1'b1; end
            8'b00100000: begin encoded_out = 3'b101; valid_out = 1'b1; end
            8'b01000000: begin encoded_out = 3'b110; valid_out = 1'b1; end
            8'b10000000: begin encoded_out = 3'b111; valid_out = 1'b1; end
            default: begin encoded_out = 3'b000; valid_out = 1'b0; end
          endcase
        end

        initial $display("Generated 8-to-3 encoder (8 inputs -> 3 outputs)");
      end

      16: begin : encoder_16to4
        // 16-to-4 encoder (simplified implementation)
        always_comb begin
          encoded_out = 4'b0000;
          valid_out = 1'b0;
```

```
          for (int i = 0; i < 16; i++) begin
            if (data_in[i]) begin
              encoded_out = i[3:0];
              valid_out = 1'b1;
              break;
            end
          end
        end

        initial $display("Generated 16-to-4 encoder (16 inputs -> 4 outputs)");
      end

      default: begin : encoder_invalid
        // Invalid width - tie outputs to zero
        assign encoded_out = '0;
        assign valid_out = 1'b0;

        initial begin
          $display("ERROR: Invalid INPUT_WIDTH = %0d", INPUT_WIDTH);
          $display("Valid widths: 2, 4, 8, 16");
        end
      end

    endcase
  endgenerate

  initial begin
    $display();
    $display("=== Encoder Width Selector ===");
    $display("INPUT_WIDTH = %0d", INPUT_WIDTH);
    $display("OUTPUT_WIDTH = %0d", OUTPUT_WIDTH);
    $display();
  end

endmodule
```

```
// encoder_width_selector_testbench.sv
module encoder_width_selector_testbench;

  // Test signals for different encoder widths
  logic [1:0]  data_in_2,  encoded_out_2;
  logic [3:0]  data_in_4,  encoded_out_4;
  logic [7:0]  data_in_8,  encoded_out_8;
  logic [15:0] data_in_16, encoded_out_16;
  logic        valid_out_2, valid_out_4, valid_out_8, valid_out_16;

  // Fixed: Correct signal widths for invalid width test
  logic [2:0] data_in_3;
  logic [1:0] encoded_out_3;  // $clog2(3) = 2, so output is [1:0]
  logic valid_out_3;

  // Instance 1: 2-to-1 encoder
  encoder_width_selector #(.INPUT_WIDTH(2)) encoder_2 (
    .data_in(data_in_2),
```

```
    .encoded_out(encoded_out_2[0]),  // Only need 1 bit
    .valid_out(valid_out_2)
);

// Instance 2: 4-to-2 encoder
encoder_width_selector #(.INPUT_WIDTH(4)) encoder_4 (
    .data_in(data_in_4),
    .encoded_out(encoded_out_4[1:0]),  // Only need 2 bits
    .valid_out(valid_out_4)
);

// Instance 3: 8-to-3 encoder
encoder_width_selector #(.INPUT_WIDTH(8)) encoder_8 (
    .data_in(data_in_8),
    .encoded_out(encoded_out_8[2:0]),  // Only need 3 bits
    .valid_out(valid_out_8)
);

// Instance 4: 16-to-4 encoder
encoder_width_selector #(.INPUT_WIDTH(16)) encoder_16 (
    .data_in(data_in_16),
    .encoded_out(encoded_out_16[3:0]),  // Only need 4 bits
    .valid_out(valid_out_16)
);

// Instance 5: Invalid width (3) - Fixed width mismatch
encoder_width_selector #(.INPUT_WIDTH(3)) encoder_invalid (
    .data_in(data_in_3),
    .encoded_out(encoded_out_3),  // Now correctly sized as [1:0]
    .valid_out(valid_out_3)
);

initial begin
    // Dump waves
    $dumpfile("encoder_width_selector_testbench.vcd");
    $dumpvars(0, encoder_width_selector_testbench);

    $display("=== Encoder Width Selector Testbench ===");
    $display("Testing parameter-based selection with generate case");
    $display();

    // Wait for generation messages
    #1;

    // Test 2-to-1 encoder
    $display("--- Testing 2-to-1 Encoder ---");
    data_in_2 = 2'b01; #1;
    $display("Input: %b -> Output: %b, Valid: %b", data_in_2, encoded_out_2[0], valid_out_2);
    data_in_2 = 2'b10; #1;
    $display("Input: %b -> Output: %b, Valid: %b", data_in_2, encoded_out_2[0], valid_out_2);
    data_in_2 = 2'b11; #1;
    $display("Input: %b -> Output: %b, Valid: %b (invalid)", data_in_2, encoded_out_2[0], val

    // Test 4-to-2 encoder
    $display("--- Testing 4-to-2 Encoder ---");
```

```
        data_in_4 = 4'b0001; #1;
        $display("Input: %b -> Output: %b, Valid: %b", data_in_4, encoded_out_4[1:0], valid_out_4
        data_in_4 = 4'b0100; #1;
        $display("Input: %b -> Output: %b, Valid: %b", data_in_4, encoded_out_4[1:0], valid_out_4
        data_in_4 = 4'b1000; #1;
        $display("Input: %b -> Output: %b, Valid: %b", data_in_4, encoded_out_4[1:0], valid_out_4

        // Test 8-to-3 encoder
        $display("--- Testing 8-to-3 Encoder ---");
        data_in_8 = 8'b00000001; #1;
        $display("Input: %b -> Output: %b, Valid: %b", data_in_8, encoded_out_8[2:0], valid_out_8
        data_in_8 = 8'b00100000; #1;
        $display("Input: %b -> Output: %b, Valid: %b", data_in_8, encoded_out_8[2:0], valid_out_8
        data_in_8 = 8'b10000000; #1;
        $display("Input: %b -> Output: %b, Valid: %b", data_in_8, encoded_out_8[2:0], valid_out_8

        // Test 16-to-4 encoder
        $display("--- Testing 16-to-4 Encoder ---");
        data_in_16 = 16'b0000000000000001; #1;
        $display("Input: %b -> Output: %b, Valid: %b", data_in_16, encoded_out_16[3:0], valid_out
        data_in_16 = 16'b0000001000000000; #1;
        $display("Input: %b -> Output: %b, Valid: %b", data_in_16, encoded_out_16[3:0], valid_out
        data_in_16 = 16'b1000000000000000; #1;
        $display("Input: %b -> Output: %b, Valid: %b", data_in_16, encoded_out_16[3:0], valid_out

        // Test invalid width encoder
        $display("--- Testing Invalid Width (3) Encoder ---");
        data_in_3 = 3'b001; #1;
        $display("Input: %b -> Output: %b, Valid: %b (should be invalid)", data_in_3, encoded_out
        data_in_3 = 3'b100; #1;
        $display("Input: %b -> Output: %b, Valid: %b (should be invalid)", data_in_3, encoded_out

        $display();
        $display("Each encoder was generated with different logic based on INPUT_WIDTH!");
        $display("Same module interface, completely different implementations!");

        #10;
        $finish;
    end

endmodule
```

```
Verilator Simulation Output:
================================================================================
Generated 2-to-1 encoder (2 inputs -> 1 output)

=== Encoder Width Selector ===
INPUT_WIDTH = 2
OUTPUT_WIDTH = 1

Generated 4-to-2 encoder (4 inputs -> 2 outputs)

=== Encoder Width Selector ===
INPUT_WIDTH = 4
OUTPUT_WIDTH = 2
```

```
Generated 8-to-3 encoder (8 inputs -> 3 outputs)

=== Encoder Width Selector ===
INPUT_WIDTH = 8
OUTPUT_WIDTH = 3

Generated 16-to-4 encoder (16 inputs -> 4 outputs)

=== Encoder Width Selector ===
INPUT_WIDTH = 16
OUTPUT_WIDTH = 4

ERROR: Invalid INPUT_WIDTH = 3
Valid widths: 2, 4, 8, 16

=== Encoder Width Selector ===
INPUT_WIDTH = 3
OUTPUT_WIDTH = 2

=== Encoder Width Selector Testbench ===
Testing parameter-based selection with generate case

--- Testing 2-to-1 Encoder ---
Input: 01 -> Output: 0, Valid: 1
Input: 10 -> Output: 1, Valid: 1
Input: 11 -> Output: 0, Valid: 0 (invalid)
--- Testing 4-to-2 Encoder ---
Input: 0001 -> Output: 00, Valid: 1
Input: 0100 -> Output: 10, Valid: 1
Input: 1000 -> Output: 11, Valid: 1
--- Testing 8-to-3 Encoder ---
Input: 00000001 -> Output: 000, Valid: 1
Input: 00100000 -> Output: 101, Valid: 1
Input: 10000000 -> Output: 111, Valid: 1
--- Testing 16-to-4 Encoder ---
Input: 0000000000000001 -> Output: 0000, Valid: 1
Input: 0000001000000000 -> Output: 1001, Valid: 1
Input: 1000000000000000 -> Output: 1111, Valid: 1
--- Testing Invalid Width (3) Encoder ---
Input: 001 -> Output: 00, Valid: 0 (should be invalid)
Input: 100 -> Output: 00, Valid: 0 (should be invalid)

Each encoder was generated with different logic based on INPUT_WIDTH!
Same module interface, completely different implementations!
================================================================================
Process finished with return code: 0
Removing Chapter_5_examples/example_10__encoder_width_selector/obj_dir directory...
Chapter_5_examples/example_10__encoder_width_selector/obj_dir removed successfully.
0
```

## Introduction to Interfaces

Interfaces provide a powerful way to group related signals and simplify connections between modules. They help reduce port lists and improve code maintainability.

**Basic Interface Declaration**

```
interface memory_if #(
    parameter int DATA_WIDTH = 32,
    parameter int ADDR_WIDTH = 16
) (
    input logic clk,
    input logic reset_n
);

    // Interface signals
    logic                    valid;
    logic                    ready;
    logic                    we;
    logic [ADDR_WIDTH-1:0]   addr;
    logic [DATA_WIDTH-1:0]   wdata;
    logic [DATA_WIDTH-1:0]   rdata;
    logic                    error;

    // Tasks and functions can be defined in interfaces
    task write_transaction(
        input logic [ADDR_WIDTH-1:0] address,
        input logic [DATA_WIDTH-1:0] data
    );
        @(posedge clk);
        valid <= 1'b1;
        we <= 1'b1;
        addr <= address;
        wdata <= data;
        @(posedge clk);
        while (!ready) @(posedge clk);
        valid <= 1'b0;
        we <= 1'b0;
    endtask

    task read_transaction(
        input  logic [ADDR_WIDTH-1:0] address,
        output logic [DATA_WIDTH-1:0] data
    );
        @(posedge clk);
        valid <= 1'b1;
        we <= 1'b0;
        addr <= address;
        @(posedge clk);
        while (!ready) @(posedge clk);
        data = rdata;
        valid <= 1'b0;
    endtask

endinterface
```

### Using Interfaces in Modules

```
// Memory controller module
module memory_controller (
    memory_if.slave  cpu_if,    // CPU interface (slave perspective)
    memory_if.master mem_if     // Memory interface (master perspective)
);

    // Interface connection logic
    always_comb begin
        // Forward CPU requests to memory
        mem_if.valid = cpu_if.valid;
        mem_if.we    = cpu_if.we;
        mem_if.addr  = cpu_if.addr;
        mem_if.wdata = cpu_if.wdata;

        // Forward memory responses to CPU
        cpu_if.ready = mem_if.ready;
        cpu_if.rdata = mem_if.rdata;
        cpu_if.error = mem_if.error;
    end

endmodule

// Memory module
module memory (
    memory_if.slave mem_if
);

    localparam int DEPTH = 2**mem_if.ADDR_WIDTH;
    logic [mem_if.DATA_WIDTH-1:0] mem_array [0:DEPTH-1];

    always_ff @(posedge mem_if.clk or negedge mem_if.reset_n) begin
        if (!mem_if.reset_n) begin
            mem_if.ready <= 1'b0;
            mem_if.rdata <= '0;
            mem_if.error <= 1'b0;
        end else begin
            mem_if.ready <= mem_if.valid;
            mem_if.error <= 1'b0;

            if (mem_if.valid) begin
                if (mem_if.we) begin
                    mem_array[mem_if.addr] <= mem_if.wdata;
                end else begin
                    mem_if.rdata <= mem_array[mem_if.addr];
                end
            end
        end
    end

endmodule
```

**Interface Instantiation and Connection**

```
module top_level;
    logic clk, reset_n;

    // Interface instances
    memory_if #(.DATA_WIDTH(32), .ADDR_WIDTH(16)) cpu_mem_if(clk, reset_n);
    memory_if #(.DATA_WIDTH(32), .ADDR_WIDTH(16)) ctrl_mem_if(clk, reset_n);

    // Module instances
    cpu cpu_inst (
        .clk(clk),
        .reset_n(reset_n),
        .mem_if(cpu_mem_if.master)  // CPU is master
    );

    memory_controller ctrl_inst (
        .cpu_if(cpu_mem_if.slave),   // Controller is slave to CPU
        .mem_if(ctrl_mem_if.master)  // Controller is master to memory
    );

    memory mem_inst (
        .mem_if(ctrl_mem_if.slave)   // Memory is slave
    );

endmodule
```

**Example 11: Basic Memory Interface**

basic_memory_interface - Simple interface declaration and usage

```
// basic_memory_interface.sv
// Simple memory interface declaration
interface memory_interface;
  logic [7:0] address;      // 8-bit address
  logic [7:0] write_data;   // 8-bit write data
  logic [7:0] read_data;    // 8-bit read data
  logic       write_enable; // Write enable signal
  logic       read_enable;  // Read enable signal
  logic       clock;        // Clock signal

  // Modport for memory controller (master)
  modport controller (
    output address,
    output write_data,
    input  read_data,
    output write_enable,
    output read_enable,
    input  clock
  );

  // Modport for memory device (slave)
  modport memory (
    input  address,
    input  write_data,
```

```
    output read_data,
    input  write_enable,
    input  read_enable,
    input  clock
  );

endinterface

// Simple memory controller module
module memory_controller (memory_interface.controller mem_if);

  initial begin
    $display("Memory Controller: Starting operations");

    // Initialize signals
    mem_if.address = 8'h00;
    mem_if.write_data = 8'h00;
    mem_if.write_enable = 1'b0;
    mem_if.read_enable = 1'b0;

    #10; // Wait

    // Write operation
    $display("Memory Controller: Writing data 0xAA to address 0x10");
    mem_if.address = 8'h10;
    mem_if.write_data = 8'hAA;
    mem_if.write_enable = 1'b1;
    mem_if.read_enable = 1'b0;

    #10; // Wait
    mem_if.write_enable = 1'b0; // End write

    #10; // Wait

    // Read operation
    $display("Memory Controller: Reading from address 0x10");
    mem_if.address = 8'h10;
    mem_if.read_enable = 1'b1;

    #10; // Wait for read data
    $display("Memory Controller: Read data 0x%02h from address 0x10", mem_if.read_data);

    mem_if.read_enable = 1'b0; // End read

    #10; // Wait

    $display("Memory Controller: Operations complete");
  end

endmodule

// Simple memory device module
module memory_device (memory_interface.memory mem_if);

  logic [7:0] memory_array [0:255]; // 256 bytes of memory
```

```
  always @(posedge mem_if.clock) begin
    if (mem_if.write_enable) begin
      memory_array[mem_if.address] <= mem_if.write_data;
      $display("Memory Device: Wrote 0x%02h to address 0x%02h", mem_if.write_data, mem_if.add
    end
  end

  // Combinational read
  always_comb begin
    if (mem_if.read_enable) begin
      mem_if.read_data = memory_array[mem_if.address];
    end else begin
      mem_if.read_data = 8'h00;
    end
  end

  // Initialize memory with some test data
  initial begin
    for (int i = 0; i < 256; i++) begin
      memory_array[i] = i[7:0]; // Initialize with address value
    end
    $display("Memory Device: Initialized with test data");
  end

endmodule

// Design under test - connects controller and memory via interface
module basic_memory_interface ();

  // Clock generation
  logic clock;
  initial begin
    clock = 0;
    forever #5 clock = ~clock; // 10 time unit period
  end

  // Interface instance
  memory_interface mem_if();

  // Connect clock to interface
  assign mem_if.clock = clock;

  // Module instances
  memory_controller controller_inst(mem_if.controller);
  memory_device memory_inst(mem_if.memory);

  initial begin
    $display();
    $display("=== Basic Memory Interface Example ===");
    $display("Demonstrating interface declaration and usage");
    $display();

    // Let simulation run for a while
    #100;
```

```
    $display();
    $display("=== Simulation Complete ===");
    $finish;
  end

endmodule
```

```
// basic_memory_interface_testbench.sv
module basic_memory_interface_testbench;

  // Instantiate design under test
  basic_memory_interface DESIGN_INSTANCE_NAME();

  initial begin
    // Dump waves
    $dumpfile("basic_memory_interface_testbench.vcd");
    $dumpvars(0, basic_memory_interface_testbench);

    $display("Testbench: Starting memory interface simulation");
    $display();

    // Wait for design to complete
    #150;

    $display();
    $display("Testbench: Memory interface simulation complete");
    $display();

    $finish;
  end

endmodule
```

```
Verilator Simulation Output:
================================================================================
Memory Device: Initialized with test data
Testbench: Starting memory interface simulation


=== Basic Memory Interface Example ===
Demonstrating interface declaration and usage

Memory Controller: Starting operations
Memory Controller: Writing data 0xAA to address 0x10
Memory Device: Wrote 0xaa to address 0x10
Memory Controller: Reading from address 0x10
Memory Controller: Read data 0xaa from address 0x10
Memory Controller: Operations complete

=== Simulation Complete ===
================================================================================
Process finished with return code: 0
Removing Chapter_5_examples/example_11__basic_memory_interface/obj_dir directory...
Chapter_5_examples/example_11__basic_memory_interface/obj_dir removed successfully.

0
```

**Example 12: Interface With Tasks**

interface_with_tasks - Interface with embedded tasks and functions

```
// interface_with_tasks.sv
// Interface with embedded tasks and functions
interface bus_interface;
  logic [7:0] write_data;
  logic [7:0] read_data;
  logic [3:0] address;
  logic       valid;
  logic       ready;
  logic       write_enable;
  logic       clock;

  // Embedded function to check if transaction is complete
  function bit is_transaction_complete();
    return (valid && ready);
  endfunction

  // Embedded function to calculate parity
  function bit calculate_parity(logic [7:0] data_in);
    return ^data_in; // XOR reduction for odd parity
  endfunction

  // Embedded task to perform a write transaction
  task automatic write_transaction(input [3:0] addr, input [7:0] data_in);
    @(posedge clock);
    address = addr;
    write_data = data_in;
    write_enable = 1'b1;
    valid = 1'b1;

    // Wait for ready signal
    while (!ready) @(posedge clock);

    $display("Interface Task: Write complete - Addr:0x%01h Data:0x%02h Parity:%b",
             addr, data_in, calculate_parity(data_in));

    @(posedge clock);
    valid = 1'b0;
    write_enable = 1'b0;
  endtask

  // Embedded task to perform a read transaction
  task automatic read_transaction(input [3:0] addr, output [7:0] data_out);
    @(posedge clock);
    address = addr;
    write_enable = 1'b0;
    valid = 1'b1;

    // Wait for ready signal
    while (!ready) @(posedge clock);

    // Wait one more clock for data to be available
    @(posedge clock);
```

```
    data_out = read_data;
    $display("Interface Task: Read complete - Addr:0x%01h Data:0x%02h Parity:%b",
             addr, data_out, calculate_parity(data_out));

    @(posedge clock);
    valid = 1'b0;
  endtask

  // Modport for master (uses tasks and functions)
  modport master (
    output address, write_data, valid, write_enable,
    input  read_data, ready, clock,
    import write_transaction,
    import read_transaction,
    import is_transaction_complete,
    import calculate_parity
  );

  // Modport for slave
  modport slave (
    input  address, write_data, valid, write_enable, clock,
    output read_data, ready,
    import is_transaction_complete,
    import calculate_parity
  );

endinterface

// Bus master module using interface tasks
module bus_master (bus_interface.master bus_if);

  logic [7:0] read_data;

  initial begin
    $display("Bus Master: Starting operations");

    // Initialize
    bus_if.address = 4'h0;
    bus_if.write_data = 8'h00;
    bus_if.valid = 1'b0;
    bus_if.write_enable = 1'b0;

    #20; // Wait for slave to be ready

    // Use interface task for write
    $display("Bus Master: Performing write using interface task");
    bus_if.write_transaction(4'hA, 8'h55);

    #10;

    // Use interface task for read
    $display("Bus Master: Performing read using interface task");
    bus_if.read_transaction(4'hA, read_data);

    #10;
```

```
    // Use interface function
    if (bus_if.is_transaction_complete()) begin
      $display("Bus Master: Transaction check passed");
    end

    #10;

    // Test parity function directly
    $display("Bus Master: Parity of 0xFF is %b (should be 0 - even parity)",
             bus_if.calculate_parity(8'hFF));
    $display("Bus Master: Parity of 0x0F is %b (should be 0 - even parity)",
             bus_if.calculate_parity(8'h0F));
    $display("Bus Master: Parity of 0x07 is %b (should be 1 - odd parity)",
             bus_if.calculate_parity(8'h07));

    $display("Bus Master: All operations complete");
  end

endmodule

// Bus slave module
module bus_slave (bus_interface.slave bus_if);

  logic [7:0] memory [0:15]; // 16 locations
  logic ready_internal;

  assign bus_if.ready = ready_internal;

  always @(posedge bus_if.clock) begin
    if (bus_if.valid && ready_internal) begin
      if (bus_if.write_enable) begin
        // Write operation
        memory[bus_if.address] <= bus_if.write_data;
        $display("Bus Slave: Stored 0x%02h at address 0x%01h",
                 bus_if.write_data, bus_if.address);
      end else begin
        // Read operation - drive read_data
        bus_if.read_data <= memory[bus_if.address];
        $display("Bus Slave: Retrieved 0x%02h from address 0x%01h",
                 memory[bus_if.address], bus_if.address);
      end
    end
  end

  // Simple ready generation
  always @(posedge bus_if.clock) begin
    if (bus_if.valid) begin
      ready_internal <= 1'b1;
    end else begin
      ready_internal <= 1'b0;
    end
  end

  // Initialize memory
  initial begin
```

```
    for (int i = 0; i < 16; i++) begin
      memory[i] = i[7:0]; // Use only lower 8 bits of i
    end
    ready_internal = 1'b0;
    $display("Bus Slave: Memory initialized");
  end

endmodule

// Design under test
module interface_with_tasks ();

  // Clock generation
  logic clock;
  initial begin
    clock = 0;
    forever #5 clock = ~clock;
  end

  // Interface instance
  bus_interface bus_if();
  assign bus_if.clock = clock;

  // Module instances
  bus_master master_inst(bus_if.master);
  bus_slave slave_inst(bus_if.slave);

  initial begin
    $display();
    $display("=== Interface with Tasks and Functions Example ===");
    $display("Demonstrating embedded tasks and functions in interfaces");
    $display();

    #200; // Let simulation run

    $display();
    $display("=== Simulation Complete ===");
    $finish;
  end

endmodule
```

```
// interface_with_tasks_testbench.sv
module interface_with_tasks_testbench;

  // Instantiate design under test
  interface_with_tasks DESIGN_INSTANCE_NAME();

  initial begin
    // Dump waves
    $dumpfile("interface_with_tasks_testbench.vcd");
    $dumpvars(0, interface_with_tasks_testbench);

    $display("Testbench: Starting interface with tasks simulation");
```

```
    $display();

    // Wait for design to complete
    #250;

    $display();
    $display("Testbench: Interface with tasks simulation complete");
    $display();

    $finish;
  end

endmodule
```

```
Verilator Simulation Output:
================================================================================
Bus Slave: Memory initialized
Testbench: Starting interface with tasks simulation


=== Interface with Tasks and Functions Example ===
Demonstrating embedded tasks and functions in interfaces

Bus Master: Starting operations
Bus Master: Performing write using interface task
Interface Task: Write complete - Addr:0xa Data:0x55 Parity:0
Bus Slave: Stored 0x55 at address 0xa
Bus Master: Performing read using interface task
Bus Slave: Retrieved 0x55 from address 0xa
Interface Task: Read complete - Addr:0xa Data:0x55 Parity:0
Bus Slave: Retrieved 0x55 from address 0xa
Bus Master: Parity of 0xFF is 0 (should be 0 - even parity)
Bus Master: Parity of 0x0F is 0 (should be 0 - even parity)
Bus Master: Parity of 0x07 is 1 (should be 1 - odd parity)
Bus Master: All operations complete

=== Simulation Complete ===
================================================================================
Process finished with return code: 0
Removing Chapter_5_examples/example_12__interface_with_tasks/obj_dir directory...
Chapter_5_examples/example_12__interface_with_tasks/obj_dir removed successfully.

0
```

## Modports and Clocking Blocks

Modports define different views of an interface for different modules, while clocking blocks provide synchronous timing control.

### Modports

Modports specify which signals are inputs, outputs, or inouts from a particular module's perspective.

```
interface axi4_lite_if #(
    parameter int DATA_WIDTH = 32,
```

```
    parameter int ADDR_WIDTH = 32
) (
    input logic aclk,
    input logic aresetn
);

    // Write Address Channel
    logic [ADDR_WIDTH-1:0]  awaddr;
    logic [2:0]             awprot;
    logic                   awvalid;
    logic                   awready;

    // Write Data Channel
    logic [DATA_WIDTH-1:0]  wdata;
    logic [(DATA_WIDTH/8)-1:0] wstrb;
    logic                   wvalid;
    logic                   wready;

    // Write Response Channel
    logic [1:0]             bresp;
    logic                   bvalid;
    logic                   bready;

    // Read Address Channel
    logic [ADDR_WIDTH-1:0]  araddr;
    logic [2:0]             arprot;
    logic                   arvalid;
    logic                   arready;

    // Read Data Channel
    logic [DATA_WIDTH-1:0]  rdata;
    logic [1:0]             rresp;
    logic                   rvalid;
    logic                   rready;

    // Master modport (drives address/data, receives responses)
    modport master (
        input  aclk, aresetn,
        output awaddr, awprot, awvalid,
        input  awready,
        output wdata, wstrb, wvalid,
        input  wready,
        input  bresp, bvalid,
        output bready,
        output araddr, arprot, arvalid,
        input  arready,
        input  rdata, rresp, rvalid,
        output rready
    );

    // Slave modport (receives address/data, drives responses)
    modport slave (
        input  aclk, aresetn,
        input  awaddr, awprot, awvalid,
        output awready,
```

```
        input  wdata, wstrb, wvalid,
        output wready,
        output bresp, bvalid,
        input  bready,
        input  araddr, arprot, arvalid,
        output arready,
        output rdata, rresp, rvalid,
        input  rready
    );

    // Monitor modport (all inputs for verification)
    modport monitor (
        input aclk, aresetn,
        input awaddr, awprot, awvalid, awready,
        input wdata, wstrb, wvalid, wready,
        input bresp, bvalid, bready,
        input araddr, arprot, arvalid, arready,
        input rdata, rresp, rvalid, rready
    );

endinterface
```

**Clocking Blocks**

Clocking blocks define synchronous timing relationships and provide a clean way to handle clocked signals in testbenches.

```
interface processor_if (
    input logic clk,
    input logic reset_n
);

    logic [31:0] instruction;
    logic [31:0] pc;
    logic        valid;
    logic        ready;
    logic        stall;
    logic        flush;

    // Clocking block for testbench use
    clocking cb @(posedge clk);
        default input #1step output #2ns;  // Input skew and output delay

        input  pc, valid, ready;
        output instruction, stall, flush;
    endclocking

    // Separate clocking block for different timing requirements
    clocking slow_cb @(posedge clk);
        default input #5ns output #10ns;

        input  pc, valid;
        output instruction;
    endclocking
```

```
    // Modports with clocking blocks
    modport tb (
        clocking cb,
        input clk, reset_n
    );

    modport dut (
        input  clk, reset_n,
        output pc, valid, ready,
        input  instruction, stall, flush
    );

endinterface
```

**Advanced Clocking Block Example**

```
interface memory_test_if (
    input logic clk,
    input logic reset_n
);

    logic [15:0] addr;
    logic [31:0] wdata;
    logic [31:0] rdata;
    logic        we;
    logic        re;
    logic        valid;
    logic        ready;

    // Clocking block with different timing for different signals
    clocking driver_cb @(posedge clk);
        default input #2ns output #1ns;

        output addr, wdata, we, re, valid;
        input  rdata, ready;
    endclocking

    // Monitor clocking block samples everything
    clocking monitor_cb @(posedge clk);
        default input #1step;

        input addr, wdata, rdata, we, re, valid, ready;
    endclocking

    // Synchronous reset clocking block
    clocking reset_cb @(posedge clk);
        input reset_n;
    endclocking

    modport driver (
        clocking driver_cb,
        input clk, reset_n
    );
```

```
    modport monitor (
        clocking monitor_cb,
        input clk, reset_n
    );

    modport dut (
        input  clk, reset_n,
        input  addr, wdata, we, re, valid,
        output rdata, ready
    );

endinterface
```

**Using Clocking Blocks in Testbenches**

```
module memory_testbench;
    logic clk = 0;
    logic reset_n;

    always #5ns clk = ~clk;  // 100MHz clock

    memory_test_if mem_if(clk, reset_n);

    // DUT instantiation
    memory dut (
        .mem_if(mem_if.dut)
    );

    // Test program using clocking blocks
    initial begin
        reset_n = 0;
        ##2 reset_n = 1;  // Wait 2 clock cycles

        // Write operation using clocking block
        mem_if.driver_cb.addr  <= 16'h1000;
        mem_if.driver_cb.wdata <= 32'hDEADBEEF;
        mem_if.driver_cb.we    <= 1'b1;
        mem_if.driver_cb.valid <= 1'b1;

        ##1;  // Wait 1 clock cycle

        wait (mem_if.driver_cb.ready);  // Wait for ready

        mem_if.driver_cb.we    <= 1'b0;
        mem_if.driver_cb.valid <= 1'b0;

        ##2;  // Wait before read

        // Read operation
        mem_if.driver_cb.addr  <= 16'h1000;
        mem_if.driver_cb.re    <= 1'b1;
        mem_if.driver_cb.valid <= 1'b1;

        ##1;
```

```
        wait (mem_if.driver_cb.ready);

        $display("Read data: %h", mem_if.driver_cb.rdata);

        mem_if.driver_cb.re    <= 1'b0;
        mem_if.driver_cb.valid <= 1'b0;

        ##5;
        $finish;
    end

endmodule
```

**Example 13: Axi4 Lite Interface**

axi4_lite_interface - Complete interface with master/slave/monitor modports

```
// Fixed axi4_lite_memory.sv
// Simple AXI4-Lite interface and memory slave

// Simple AXI4-Lite Interface
interface axi4_lite_if (input logic clk, input logic rst_n);

  // Write Address Channel
  logic [31:0] awaddr;
  logic        awvalid;
  logic        awready;

  // Write Data Channel
  logic [31:0] wdata;
  logic [3:0]  wstrb;
  logic        wvalid;
  logic        wready;

  // Write Response Channel
  logic [1:0]  bresp;
  logic        bvalid;
  logic        bready;

  // Read Address Channel
  logic [31:0] araddr;
  logic        arvalid;
  logic        arready;

  // Read Data Channel
  logic [31:0] rdata;
  logic [1:0]  rresp;
  logic        rvalid;
  logic        rready;

  // Master modport (initiates transactions)
  modport master (
    output awaddr, awvalid, wdata, wstrb, wvalid, bready,
    output araddr, arvalid, rready,
```

```
    input  awready, wready, bresp, bvalid,
    input  arready, rdata, rresp, rvalid,
    input  clk, rst_n
  );

  // Slave modport (responds to transactions)
  modport slave (
    input  awaddr, awvalid, wdata, wstrb, wvalid, bready,
    input  araddr, arvalid, rready,
    output awready, wready, bresp, bvalid,
    output arready, rdata, rresp, rvalid,
    input  clk, rst_n
  );

  // Monitor modport (observes all signals)
  modport monitor (
    input awaddr, awvalid, awready, wdata, wstrb, wvalid, wready,
    input bresp, bvalid, bready, araddr, arvalid, arready,
    input rdata, rresp, rvalid, rready, clk, rst_n
  );

endinterface

// Fixed Memory Slave (just 4 registers)
module axi4_lite_memory_slave (
  axi4_lite_if.slave axi_bus
);

  // Just 4 32-bit registers
  logic [31:0] memory [0:3];

  // State tracking for proper handshakes
  logic write_addr_accepted;
  logic write_data_accepted;

  // Initialize memory
  initial begin
    memory[0] = 32'h00000000;
    memory[1] = 32'h00000000;
    memory[2] = 32'h00000000;
    memory[3] = 32'h00000000;
  end

  // Write logic - fixed to prevent infinite loops
  always_ff @(posedge axi_bus.clk or negedge axi_bus.rst_n) begin
    if (!axi_bus.rst_n) begin
      axi_bus.awready <= 1'b0;
      axi_bus.wready <= 1'b0;
      axi_bus.bvalid <= 1'b0;
      axi_bus.bresp <= 2'b00;
      write_addr_accepted <= 1'b0;
      write_data_accepted <= 1'b0;
    end else begin

      // Write address handshake - only pulse awready for one cycle
```

```
    if (axi_bus.awvalid && !axi_bus.awready && !write_addr_accepted) begin
      axi_bus.awready <= 1'b1;
      write_addr_accepted <= 1'b1;
    end else begin
      axi_bus.awready <= 1'b0;
    end

    // Write data handshake - only pulse wready for one cycle
    if (axi_bus.wvalid && !axi_bus.wready && !write_data_accepted) begin
      axi_bus.wready <= 1'b1;
      write_data_accepted <= 1'b1;
    end else begin
      axi_bus.wready <= 1'b0;
    end

    // Perform write when both address and data are accepted
    if (write_addr_accepted && write_data_accepted && !axi_bus.bvalid) begin
      if (axi_bus.awaddr[31:4] == 28'h0) begin // Valid address range
        memory[axi_bus.awaddr[3:2]] <= axi_bus.wdata;
        axi_bus.bresp <= 2'b00; // OK
        $display("MEMORY: Write 0x%08h to address 0x%08h", axi_bus.wdata, axi_bus.awaddr);
      end else begin
        axi_bus.bresp <= 2'b10; // SLVERR - invalid address
        $display("MEMORY: Write error - invalid address 0x%08h", axi_bus.awaddr);
      end
      axi_bus.bvalid <= 1'b1;
    end

    // Clear response when acknowledged
    if (axi_bus.bvalid && axi_bus.bready) begin
      axi_bus.bvalid <= 1'b0;
      write_addr_accepted <= 1'b0;
      write_data_accepted <= 1'b0;
    end
  end
end

// Read logic - fixed to prevent infinite loops
always_ff @(posedge axi_bus.clk or negedge axi_bus.rst_n) begin
  if (!axi_bus.rst_n) begin
    axi_bus.arready <= 1'b0;
    axi_bus.rvalid <= 1'b0;
    axi_bus.rdata <= 32'h0;
    axi_bus.rresp <= 2'b00;
  end else begin

    // Read address handshake - only pulse arready for one cycle
    if (axi_bus.arvalid && !axi_bus.arready && !axi_bus.rvalid) begin
      axi_bus.arready <= 1'b1;

      // Perform read immediately
      if (axi_bus.araddr[31:4] == 28'h0) begin // Valid address range
        axi_bus.rdata <= memory[axi_bus.araddr[3:2]];
        axi_bus.rresp <= 2'b00; // OK
        $display("MEMORY: Read 0x%08h from address 0x%08h", memory[axi_bus.araddr[3:2]], ax
```

```
        end else begin
          axi_bus.rdata <= 32'hDEADC0DE; // Error pattern
          axi_bus.rresp <= 2'b10; // SLVERR
          $display("MEMORY: Read error - invalid address 0x%08h", axi_bus.araddr);
        end
        axi_bus.rvalid <= 1'b1;

      end else begin
        axi_bus.arready <= 1'b0;
      end

      // Clear read data when acknowledged
      if (axi_bus.rvalid && axi_bus.rready) begin
        axi_bus.rvalid <= 1'b0;
      end
    end
  end

endmodule
```

```
// axi4_lite_memory_testbench.sv
// Simple testbench for AXI4-Lite Memory

module axi4_lite_memory_testbench;

  logic clk, rst_n;

  // Clock generation
  initial begin
    clk = 0;
    forever #5 clk = ~clk;
  end

  // Interface instance
  axi4_lite_if axi_bus (clk, rst_n);

  // Memory slave instance
  axi4_lite_memory_slave memory_slave (.axi_bus(axi_bus.slave));

  // Simple Monitor
  always @(posedge axi_bus.clk) begin
    // Monitor writes
    if (axi_bus.awvalid && axi_bus.awready) begin
      $display("MONITOR: Write Address = 0x%08h", axi_bus.awaddr);
    end
    if (axi_bus.wvalid && axi_bus.wready) begin
      $display("MONITOR: Write Data = 0x%08h", axi_bus.wdata);
    end
    if (axi_bus.bvalid && axi_bus.bready) begin
      $display("MONITOR: Write Response = %s", (axi_bus.bresp == 2'b00) ? "OK" : "ERROR");
    end

    // Monitor reads
    if (axi_bus.arvalid && axi_bus.arready) begin
```

```
    $display("MONITOR: Read Address = 0x%08h", axi_bus.araddr);
  end
  if (axi_bus.rvalid && axi_bus.rready) begin
    $display("MONITOR: Read Data = 0x%08h, Response = %s",
             axi_bus.rdata, (axi_bus.rresp == 2'b00) ? "OK" : "ERROR");
  end
end

// Simple master tasks
task write_data(input [31:0] addr, input [31:0] data);
  $display("MASTER: Starting write - addr=0x%08h, data=0x%08h", addr, data);
  @(posedge clk);
  axi_bus.awaddr = addr;
  axi_bus.awvalid = 1'b1;
  axi_bus.wdata = data;
  axi_bus.wstrb = 4'hF;
  axi_bus.wvalid = 1'b1;
  axi_bus.bready = 1'b1;

  wait(axi_bus.awready && axi_bus.wready);
  @(posedge clk);
  axi_bus.awvalid = 1'b0;
  axi_bus.wvalid = 1'b0;

  wait(axi_bus.bvalid);
  @(posedge clk);
  axi_bus.bready = 1'b0;
  $display("MASTER: Write complete");
endtask

task read_data(input [31:0] addr, output [31:0] data);
  $display("MASTER: Starting read - addr=0x%08h", addr);
  @(posedge clk);
  axi_bus.araddr = addr;
  axi_bus.arvalid = 1'b1;
  axi_bus.rready = 1'b1;

  wait(axi_bus.arready);
  @(posedge clk);
  axi_bus.arvalid = 1'b0;

  wait(axi_bus.rvalid);
  data = axi_bus.rdata;
  @(posedge clk);
  axi_bus.rready = 1'b0;
  $display("MASTER: Read complete - data=0x%08h", data);
endtask

// Initialize master signals
initial begin
  axi_bus.awaddr = 0;
  axi_bus.awvalid = 0;
  axi_bus.wdata = 0;
  axi_bus.wstrb = 0;
  axi_bus.wvalid = 0;
```

```
    axi_bus.bready = 0;
    axi_bus.araddr = 0;
    axi_bus.arvalid = 0;
    axi_bus.rready = 0;
  end

  // Test sequence
  initial begin
    logic [31:0] data_read;

    // Dump waves
    $dumpfile("axi4_lite_memory_testbench.vcd");
    $dumpvars(0, axi4_lite_memory_testbench);

    $display("=== AXI4-Lite Memory Test Starting ===");

    // Reset
    rst_n = 0;
    repeat(10) @(posedge clk);
    rst_n = 1;
    repeat(5) @(posedge clk);

    $display("\n--- Testing Basic Operations ---");

    // Write to register 0
    write_data(32'h00000000, 32'hCAFEBABE);
    repeat(2) @(posedge clk);

    // Read from register 0
    read_data(32'h00000000, data_read);
    assert(data_read == 32'hCAFEBABE) else $error("Read mismatch at address 0x00000000");
    repeat(2) @(posedge clk);

    // Write to register 1
    write_data(32'h00000004, 32'h12345678);
    repeat(2) @(posedge clk);

    // Read from register 1
    read_data(32'h00000004, data_read);
    assert(data_read == 32'h12345678) else $error("Read mismatch at address 0x00000004");
    repeat(2) @(posedge clk);

    // Write to register 2
    write_data(32'h00000008, 32'hDEADBEEF);
    repeat(2) @(posedge clk);

    // Read from register 2
    read_data(32'h00000008, data_read);
    assert(data_read == 32'hDEADBEEF) else $error("Read mismatch at address 0x00000008");
    repeat(2) @(posedge clk);

    $display("\n--- Testing Error Conditions ---");

    // Try invalid address (should generate error)
    write_data(32'h00000020, 32'hBADDAD00);
```

```
        repeat(2) @(posedge clk);

        // Try to read invalid address
        read_data(32'h00000020, data_read);
        repeat(2) @(posedge clk);

        $display("\n--- Testing Uninitialized Memory ---");

        // Read from register 3 (should be 0)
        read_data(32'h0000000C, data_read);
        assert(data_read == 32'h00000000) else $error("Uninitialized memory not zero");
        repeat(2) @(posedge clk);

        $display("\n=== AXI4-Lite Memory Test Complete ===");
        $display("All tests passed successfully!");

        repeat(10) @(posedge clk);
        $finish;
    end

endmodule
```

```
Verilator Simulation Output:
================================================================================
=== AXI4-Lite Memory Test Starting ===

--- Testing Basic Operations ---
MASTER: Starting write - addr=0x00000000, data=0xcafebabe
MEMORY: Write 0xcafebabe to address 0x00000000
MASTER: Write complete
MASTER: Starting read - addr=0x00000000
MEMORY: Read 0xcafebabe from address 0x00000000
MONITOR: Read Data = 0xcafebabe, Response =    OK
MASTER: Read complete - data=0xcafebabe
MASTER: Starting write - addr=0x00000004, data=0x12345678
MONITOR: Write Response =    OK
MEMORY: Write 0x12345678 to address 0x00000004
MASTER: Write complete
MASTER: Starting read - addr=0x00000004
MEMORY: Read 0x12345678 from address 0x00000004
MONITOR: Read Data = 0x12345678, Response =    OK
MASTER: Read complete - data=0x12345678
MASTER: Starting write - addr=0x00000008, data=0xdeadbeef
MONITOR: Write Response =    OK
MEMORY: Write 0xdeadbeef to address 0x00000008
MASTER: Write complete
MASTER: Starting read - addr=0x00000008
MEMORY: Read 0xdeadbeef from address 0x00000008
MONITOR: Read Data = 0xdeadbeef, Response =    OK
MASTER: Read complete - data=0xdeadbeef

--- Testing Error Conditions ---
MASTER: Starting write - addr=0x00000020, data=0xbaddad00
MONITOR: Write Response =    OK
MEMORY: Write error - invalid address 0x00000020
MASTER: Write complete
```

```
MASTER: Starting read - addr=0x00000020
MEMORY: Read error - invalid address 0x00000020
MONITOR: Read Data = 0xdeadc0de, Response = ERROR
MASTER: Read complete - data=0xdeadc0de

--- Testing Uninitialized Memory ---
MASTER: Starting read - addr=0x0000000c
MEMORY: Read 0x00000000 from address 0x0000000c
MONITOR: Read Data = 0x00000000, Response =    OK
MASTER: Read complete - data=0x00000000

=== AXI4-Lite Memory Test Complete ===
All tests passed successfully!
================================================================================
Process finished with return code: 0
Removing Chapter_5_examples/example_13__axi4_lite_memory/obj_dir directory...
Chapter_5_examples/example_13__axi4_lite_memory/obj_dir removed successfully.

0
```

**Example 14: Clocked Processor Interface**

clocked_processor_interface - Clocking blocks with different timing requirements

```
// clocked_processor_interface.sv
module clocked_processor_interface (
  input  logic       clk,
  input  logic       rst_n,
  input  logic [7:0] data_in,
  input  logic       valid_in,
  output logic [7:0] data_out,
  output logic       valid_out,
  output logic       ready
);

  // Simple processor that doubles the input data
  logic [7:0] internal_data;

  always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      data_out  <= 8'h00;
      valid_out <= 1'b0;
      ready     <= 1'b1;
      internal_data <= 8'h00;
    end else begin
      if (valid_in && ready) begin
        internal_data <= data_in;
        data_out      <= data_in << 1;  // Double the input
        valid_out     <= 1'b1;
        ready         <= 1'b0;          // Not ready for next cycle
      end else begin
        valid_out <= 1'b0;
        ready     <= 1'b1;              // Ready for new data
      end
    end
  end
```

```
  initial $display("Clocked processor interface initialized");

endmodule
```

```
// clocked_processor_interface_testbench.sv
module clocked_processor_interface_testbench;

  // Clock and reset
  logic clk;
  logic rst_n;

  // Interface signals
  logic [7:0] data_in;
  logic       valid_in;
  logic [7:0] data_out;
  logic       valid_out;
  logic       ready;

  // Clock generation
  initial begin
    clk = 0;
    forever #5 clk = ~clk;  // 10ns period
  end

  // Clocking block for driving inputs (setup before clock edge)
  clocking driver_cb @(posedge clk);
    default input #1step output #2ns;  // Input skew 1 step, output skew 2ns
    output data_in;
    output valid_in;
    input  ready;
  endclocking

  // Clocking block for monitoring outputs (sample after clock edge)
  clocking monitor_cb @(posedge clk);
    default input #3ns;  // Sample 3ns after clock edge
    input data_out;
    input valid_out;
    input ready;
  endclocking

  // Instantiate design under test
  clocked_processor_interface dut(
    .clk(clk),
    .rst_n(rst_n),
    .data_in(data_in),
    .valid_in(valid_in),
    .data_out(data_out),
    .valid_out(valid_out),
    .ready(ready)
  );

  // Test sequence
  initial begin
    // Dump waves
```

```
    $dumpfile("clocked_processor_interface_testbench.vcd");
    $dumpvars(0, clocked_processor_interface_testbench);

    $display("Starting clocked processor interface test");

    // Initialize signals
    rst_n = 0;
    data_in = 8'h00;
    valid_in = 0;

    // Reset sequence
    repeat(3) @(posedge clk);
    rst_n = 1;
    $display("Reset released at time %0t", $time);

    // Wait for ready
    wait(ready);

    // Test case 1: Send data 0x05
    @(driver_cb);
    driver_cb.data_in <= 8'h05;
    driver_cb.valid_in <= 1'b1;
    $display("Sending data 0x05 at time %0t", $time);

    @(driver_cb);
    driver_cb.valid_in <= 1'b0;

    // Wait for output and check
    @(monitor_cb iff monitor_cb.valid_out);
    $display("Received data 0x%02h at time %0t (expected 0x0A)", monitor_cb.data_out, $time);

    // Test case 2: Send data 0x0F
    wait(monitor_cb.ready);
    @(driver_cb);
    driver_cb.data_in <= 8'h0F;
    driver_cb.valid_in <= 1'b1;
    $display("Sending data 0x0F at time %0t", $time);

    @(driver_cb);
    driver_cb.valid_in <= 1'b0;

    // Wait for output and check
    @(monitor_cb iff monitor_cb.valid_out);
    $display("Received data 0x%02h at time %0t (expected 0x1E)", monitor_cb.data_out, $time);

    // Test case 3: Send data 0x80 (test overflow)
    wait(monitor_cb.ready);
    @(driver_cb);
    driver_cb.data_in <= 8'h80;
    driver_cb.valid_in <= 1'b1;
    $display("Sending data 0x80 at time %0t", $time);

    @(driver_cb);
    driver_cb.valid_in <= 1'b0;
```

```
    // Wait for output and check
    @(monitor_cb iff monitor_cb.valid_out);
    $display("Received data 0x%02h at time %0t (expected 0x00 due to overflow)", monitor_cb.d

    // Wait a few more cycles
    repeat(5) @(posedge clk);

    $display("Clocked processor interface test completed at time %0t", $time);
    $finish;
  end

  // Timeout watchdog
  initial begin
    #1000ns;
    $display("ERROR: Test timeout!");
    $finish;
  end

endmodule
```

```
Verilator Simulation Output:
================================================================================
Clocked processor interface initialized
Starting clocked processor interface test
Reset released at time 25
Sending data 0x05 at time 25
Received data 0x0a at time 5035 (expected 0x0A)
Sending data 0x0F at time 5045
Received data 0x0a at time 5075 (expected 0x1E)
Sending data 0x80 at time 5085
Received data 0x0a at time 5115 (expected 0x00 due to overflow)
Clocked processor interface test completed at time 5165
================================================================================
Process finished with return code: 0
Removing Chapter_5_examples/example_14__clocked_processor_interface/obj_dir directory...
Chapter_5_examples/example_14__clocked_processor_interface/obj_dir removed successfully.
0
```

**Example 15: Testbench With Clocking**

testbench_with_clocking - Using clocking blocks in verification environment

```
// counter_with_enable.sv
module counter_with_enable (
  input  logic       clk,
  input  logic       reset_n,
  input  logic       enable,
  output logic [3:0] count
);

  always_ff @(posedge clk or negedge reset_n) begin
    if (!reset_n) begin
      count <= 4'b0000;
    end else if (enable) begin
```

```
      count <= count + 1;
    end
  end

endmodule
```

```
// counter_with_enable_testbench.sv - Simplified Output Version
module counter_testbench;

  // Clock and reset signals
  logic       clk;
  logic       reset_n;
  logic       enable;
  logic [3:0] count;

  // Clock generation
  initial begin
    clk = 0;
    forever #5 clk = ~clk;  // 10ns clock period (100MHz)
  end

  // Instantiate design under test
  counter_with_enable counter_dut (
    .clk(clk),
    .reset_n(reset_n),
    .enable(enable),
    .count(count)
  );

  // Clocking block
  clocking cb @(posedge clk);
    default input #1step output #2ns;
    output reset_n;
    output enable;
    input  count;
  endclocking

  // Simplified signal driving task
  task drive_signals(input logic rst_val, input logic en_val);
    @(posedge clk);
    #2ns;
    reset_n = rst_val;
    enable = en_val;
  endtask

  // Test sequence
  initial begin
    // Initialize VCD dump
    $dumpfile("counter_testbench.vcd");
    $dumpvars(0, counter_testbench);

    $display("Starting Counter Test");

    // Initialize
```

```
    reset_n = 1'b0;
    enable = 1'b0;
    repeat(2) @(posedge clk);

    // Release reset
    $display("Reset released");
    drive_signals(1'b1, 1'b0);
    repeat(2) @(posedge clk);

    // Test enable=0
    $display("Testing enable=0");
    drive_signals(1'b1, 1'b0);
    repeat(4) @(posedge clk);

    // Test enable=1
    $display("Testing enable=1 - counting");
    drive_signals(1'b1, 1'b1);
    repeat(8) @(posedge clk);

    // Test reset during counting
    $display("Reset during count");
    drive_signals(1'b0, 1'b1);
    repeat(2) @(posedge clk);

    // Continue counting
    drive_signals(1'b1, 1'b1);
    repeat(6) @(posedge clk);

    // Disable counter
    $display("Counter disabled");
    drive_signals(1'b1, 1'b0);
    repeat(4) @(posedge clk);

    $display("Test completed");
    $finish;
  end

endmodule
```

```
Verilator Simulation Output:
================================================================================
Starting Counter Test
Reset released
Testing enable=0
Testing enable=1 - counting
Reset during count
Counter disabled
Test completed
================================================================================
Process finished with return code: 0
Removing Chapter_5_examples/example_15__testbench_with_clocking/obj_dir directory...
Chapter_5_examples/example_15__testbench_with_clocking/obj_dir removed successfully.

0
```

## Summary

This chapter covered the essential concepts of SystemVerilog modules and interfaces:

**Modules** form the basic building blocks with proper port declarations and hierarchical instantiation capabilities.

**Parameters and localparams** enable configurable and reusable designs with type safety and parameter validation.

**Generate blocks** provide powerful compile-time code generation for creating repetitive structures and conditional compilation.

**Interfaces** simplify complex designs by grouping related signals and providing reusable communication protocols.

**Modports** define different perspectives of interfaces for various modules, ensuring proper signal direction and access control.

**Clocking blocks** provide precise timing control for synchronous designs, particularly useful in verification environments.

These features work together to create scalable, maintainable, and reusable SystemVerilog designs that can handle complex digital systems efficiently.