Chapter 4: Control Flow Statements

Control flow statements in SystemVerilog allow you to control the execution path of your code based on conditions and loops. This chapter covers all essential control structures used in both synthesizable RTL design and testbench development.

if-else Statements

The `if-else` statement is the most fundamental conditional control structure in SystemVerilog.

Basic Syntax

```
if (condition1) begin
    // statements
end else if (condition2) begin
    // statements
end else begin
    // statements
end
```

Single Statement (without begin-end)

```
if (condition)
    statement;
else
    statement;
```

Practical Examples

Example 1: Simple Comparator

```
// comparator.sv
module comparator(
    input logic [7:0] a, b,
    output logic gt, eq, lt
);
    always_comb begin
        if (a > b) begin
            gt = 1'b1;
            eq = 1'b0;
            lt = 1'b0;
        end else if (a == b) begin
            gt = 1'b0;
            eq = 1'b1;
```

```
            lt = 1'b0;
        end else begin
            gt = 1'b0;
            eq = 1'b0;
            lt = 1'b1;
        end
    end
endmodule
```

```
// comparator_testbench.sv
module comparator_testbench;  // Testbench module

    // Testbench signals
    logic [7:0] a, b;
    logic gt, eq, lt;

    // Instantiate design under test
    comparator DUT (
        .a(a),
        .b(b),
        .gt(gt),
        .eq(eq),
        .lt(lt)
    );

    initial begin
        // Dump waves
        $dumpfile("comparator_testbench.vcd");       // Specify the VCD file
        $dumpvars(0, comparator_testbench);          // Dump all variables in the test module

        $display();
        $display("Starting Comparator Tests");
        $display("====================");
        $display();

        // Test case 1: a > b
        a = 8'h50; b = 8'h30;
        #1;  // Wait for combinational logic to settle
        $display("Test 1: a=%0d, b=%0d -> gt=%b, eq=%b, lt=%b", a, b, gt, eq, lt);
        assert (gt == 1'b1 && eq == 1'b0 && lt == 1'b0)
            else $error("Test 1 failed: Expected gt=1, eq=0, lt=0");

        // Test case 2: a == b
        a = 8'h42; b = 8'h42;
        #1;
        $display("Test 2: a=%0d, b=%0d -> gt=%b, eq=%b, lt=%b", a, b, gt, eq, lt);
        assert (gt == 1'b0 && eq == 1'b1 && lt == 1'b0)
            else $error("Test 2 failed: Expected gt=0, eq=1, lt=0");

        // Test case 3: a < b
        a = 8'h10; b = 8'h60;
        #1;
        $display("Test 3: a=%0d, b=%0d -> gt=%b, eq=%b, lt=%b", a, b, gt, eq, lt);
        assert (gt == 1'b0 && eq == 1'b0 && lt == 1'b1)
```

```
            else $error("Test 3 failed: Expected gt=0, eq=0, lt=1");

        // Test case 4: Edge case - maximum values
        a = 8'hFF; b = 8'hFF;
        #1;
        $display("Test 4: a=%0d, b=%0d -> gt=%b, eq=%b, lt=%b", a, b, gt, eq, lt);
        assert (gt == 1'b0 && eq == 1'b1 && lt == 1'b0)
            else $error("Test 4 failed: Expected gt=0, eq=1, lt=0");

        // Test case 5: Edge case - minimum values
        a = 8'h00; b = 8'h00;
        #1;
        $display("Test 5: a=%0d, b=%0d -> gt=%b, eq=%b, lt=%b", a, b, gt, eq, lt);
        assert (gt == 1'b0 && eq == 1'b1 && lt == 1'b0)
            else $error("Test 5 failed: Expected gt=0, eq=1, lt=0");

        // Test case 6: One maximum, one minimum
        a = 8'hFF; b = 8'h00;
        #1;
        $display("Test 6: a=%0d, b=%0d -> gt=%b, eq=%b, lt=%b", a, b, gt, eq, lt);
        assert (gt == 1'b1 && eq == 1'b0 && lt == 1'b0)
            else $error("Test 6 failed: Expected gt=1, eq=0, lt=0");

        // Test case 7: One minimum, one maximum
        a = 8'h00; b = 8'hFF;
        #1;
        $display("Test 7: a=%0d, b=%0d -> gt=%b, eq=%b, lt=%b", a, b, gt, eq, lt);
        assert (gt == 1'b0 && eq == 1'b0 && lt == 1'b1)
            else $error("Test 7 failed: Expected gt=0, eq=0, lt=1");

        $display();
        $display("All tests completed!");
        $display("====================");
        $display();

        $finish;  // End simulation
    end

endmodule
```

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

Starting Comparator Tests
====================

Test 1: a=80, b=48 -> gt=1, eq=0, lt=0
Test 2: a=66, b=66 -> gt=0, eq=1, lt=0
Test 3: a=16, b=96 -> gt=0, eq=0, lt=1
Test 4: a=255, b=255 -> gt=0, eq=1, lt=0
Test 5: a=0, b=0 -> gt=0, eq=1, lt=0
Test 6: a=255, b=0 -> gt=1, eq=0, lt=0
Test 7: a=0, b=255 -> gt=0, eq=0, lt=1

All tests completed!
====================
```

```
================================================================================
Process finished with return code: 0
Removing Chapter_4_examples/example_1__comparator/obj_dir directory...
Chapter_4_examples/example_1__comparator/obj_dir removed successfully.

0
```

Example 2: Priority Encoder

```
// priority_encoder.sv
module priority_encoder(
    input logic [7:0] data_in,
    output logic [2:0] encoded_out,
    output logic valid
);
    always_comb begin
        if (data_in[7])
            encoded_out = 3'd7;
        else if (data_in[6])
            encoded_out = 3'd6;
        else if (data_in[5])
            encoded_out = 3'd5;
        else if (data_in[4])
            encoded_out = 3'd4;
        else if (data_in[3])
            encoded_out = 3'd3;
        else if (data_in[2])
            encoded_out = 3'd2;
        else if (data_in[1])
            encoded_out = 3'd1;
        else if (data_in[0])
            encoded_out = 3'd0;
        else
            encoded_out = 3'd0;

        valid = |data_in; // OR reduction - valid when any bit is set
    end
endmodule
```

```
// priority_encoder_testbench.sv
module priority_encoder_testbench;  // Testbench module

    // Testbench signals
    logic [7:0] data_in;
    logic [2:0] encoded_out;
    logic valid;

    // Test counter
    integer test_count = 0;
    integer pass_count = 0;

    // Instantiate design under test
    priority_encoder DUT (
        .data_in(data_in),
```

```
        .encoded_out(encoded_out),
        .valid(valid)
    );

    // Task to run a test case
    task run_test(
        input [7:0] test_data,
        input [2:0] expected_out,
        input expected_valid,
        input string test_name
    );
        test_count++;
        data_in = test_data;
        #1; // Wait for combinational logic to settle

        $display("Test %0d: %s", test_count, test_name);
        $display("  Input: 8'b%08b (0x%02h)", data_in, data_in);
        $display("  Output: encoded_out=%0d, valid=%b", encoded_out, valid);
        $display("  Expected: encoded_out=%0d, valid=%b", expected_out, expected_valid);

        if (encoded_out == expected_out && valid == expected_valid) begin
            $display("PASS");
            pass_count++;
        end else begin
            $display("FAIL");
            $error("Test %0d failed: Expected encoded_out=%0d, valid=%b, got encoded_out=%0d,
                   test_count, expected_out, expected_valid, encoded_out, valid);
        end
        $display();
    endtask

    initial begin
        // Dump waves
        $dumpfile("priority_encoder_testbench.vcd");       // Specify the VCD file
        $dumpvars(0, priority_encoder_testbench);          // Dump all variables in the test

        $display();
        $display("Hello from testbench!");
        $display("Starting Priority Encoder Tests");
        $display("================================");
        $display();
        $display("Priority Encoder: Encodes the position of the highest priority (MSB) active
        $display("- 8-bit input, 3-bit encoded output");
        $display("- Bit 7 has highest priority, Bit 0 has lowest priority");
        $display("- Valid output indicates if any input bit is active");
        $display();

        // Test 1: No input (all zeros)
        run_test(8'b00000000, 3'd0, 1'b0, "All zeros - no valid input");

        // Test 2: Single bit tests (one bit at a time)
        run_test(8'b00000001, 3'd0, 1'b1, "Only bit 0 active");
        run_test(8'b00000010, 3'd1, 1'b1, "Only bit 1 active");
        run_test(8'b00000100, 3'd2, 1'b1, "Only bit 2 active");
        run_test(8'b00001000, 3'd3, 1'b1, "Only bit 3 active");
```

```
        run_test(8'b00010000, 3'd4, 1'b1, "Only bit 4 active");
        run_test(8'b00100000, 3'd5, 1'b1, "Only bit 5 active");
        run_test(8'b01000000, 3'd6, 1'b1, "Only bit 6 active");
        run_test(8'b10000000, 3'd7, 1'b1, "Only bit 7 active (highest priority)");

        // Test 3: Multiple bits - priority should go to highest bit
        run_test(8'b10000001, 3'd7, 1'b1, "Bits 7 and 0 - priority to bit 7");
        run_test(8'b01000010, 3'd6, 1'b1, "Bits 6 and 1 - priority to bit 6");
        run_test(8'b00100100, 3'd5, 1'b1, "Bits 5 and 2 - priority to bit 5");
        run_test(8'b00011000, 3'd4, 1'b1, "Bits 4 and 3 - priority to bit 4");

        // Test 4: Sequential patterns
        run_test(8'b11111111, 3'd7, 1'b1, "All bits set - priority to bit 7");
        run_test(8'b01111111, 3'd6, 1'b1, "Bits 6-0 set - priority to bit 6");
        run_test(8'b00111111, 3'd5, 1'b1, "Bits 5-0 set - priority to bit 5");
        run_test(8'b00011111, 3'd4, 1'b1, "Bits 4-0 set - priority to bit 4");
        run_test(8'b00001111, 3'd3, 1'b1, "Bits 3-0 set - priority to bit 3");
        run_test(8'b00000111, 3'd2, 1'b1, "Bits 2-0 set - priority to bit 2");
        run_test(8'b00000011, 3'd1, 1'b1, "Bits 1-0 set - priority to bit 1");

        // Test 5: Random patterns to verify priority
        run_test(8'b10101010, 3'd7, 1'b1, "Alternating pattern starting with bit 7");
        run_test(8'b01010101, 3'd6, 1'b1, "Alternating pattern starting with bit 6");
        run_test(8'b00110011, 3'd5, 1'b1, "Pattern 00110011 - priority to bit 5");
        run_test(8'b00001100, 3'd3, 1'b1, "Pattern 00001100 - priority to bit 3");

        // Test 6: Edge cases
        run_test(8'b11000000, 3'd7, 1'b1, "Only upper bits (7,6) - priority to bit 7");
        run_test(8'b00000011, 3'd1, 1'b1, "Only lower bits (1,0) - priority to bit 1");

        // Summary
        $display("Test Summary:");
        $display("============");
        $display("Total tests: %0d", test_count);
        $display("Passed: %0d", pass_count);
        $display("Failed: %0d", test_count - pass_count);

        if (pass_count == test_count) begin
            $display("ALL TESTS PASSED!");
        end else begin
            $display("Some tests failed. Please review.");
        end

        $display();
        $display("Priority Encoder Testing Complete!");
        $display("==================================");
        $display();

        $finish;  // End simulation
    end

endmodule
```

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

```
Hello from testbench!
Starting Priority Encoder Tests
==============================

Priority Encoder: Encodes the position of the highest priority (MSB) active bit
- 8-bit input, 3-bit encoded output
- Bit 7 has highest priority, Bit 0 has lowest priority
- Valid output indicates if any input bit is active

Test 1: All zeros - no valid input
  Input: 8'b00000000 (0x00)
  Output: encoded_out=0, valid=0
  Expected: encoded_out=0, valid=0
PASS

Test 2: Only bit 0 active
  Input: 8'b00000001 (0x01)
  Output: encoded_out=0, valid=1
  Expected: encoded_out=0, valid=1
PASS

Test 3: Only bit 1 active
  Input: 8'b00000010 (0x02)
  Output: encoded_out=1, valid=1
  Expected: encoded_out=1, valid=1
PASS

Test 4: Only bit 2 active
  Input: 8'b00000100 (0x04)
  Output: encoded_out=2, valid=1
  Expected: encoded_out=2, valid=1
PASS

Test 5: Only bit 3 active
  Input: 8'b00001000 (0x08)
  Output: encoded_out=3, valid=1
  Expected: encoded_out=3, valid=1
PASS

Test 6: Only bit 4 active
  Input: 8'b00010000 (0x10)
  Output: encoded_out=4, valid=1
  Expected: encoded_out=4, valid=1
PASS

Test 7: Only bit 5 active
  Input: 8'b00100000 (0x20)
  Output: encoded_out=5, valid=1
  Expected: encoded_out=5, valid=1
PASS

Test 8: Only bit 6 active
  Input: 8'b01000000 (0x40)
  Output: encoded_out=6, valid=1
  Expected: encoded_out=6, valid=1
PASS
```

```
Test 9: Only bit 7 active (highest priority)
  Input: 8'b10000000 (0x80)
  Output: encoded_out=7, valid=1
  Expected: encoded_out=7, valid=1
PASS

Test 10: Bits 7 and 0 - priority to bit 7
  Input: 8'b10000001 (0x81)
  Output: encoded_out=7, valid=1
  Expected: encoded_out=7, valid=1
PASS

Test 11: Bits 6 and 1 - priority to bit 6
  Input: 8'b01000010 (0x42)
  Output: encoded_out=6, valid=1
  Expected: encoded_out=6, valid=1
PASS

Test 12: Bits 5 and 2 - priority to bit 5
  Input: 8'b00100100 (0x24)
  Output: encoded_out=5, valid=1
  Expected: encoded_out=5, valid=1
PASS

Test 13: Bits 4 and 3 - priority to bit 4
  Input: 8'b00011000 (0x18)
  Output: encoded_out=4, valid=1
  Expected: encoded_out=4, valid=1
PASS

Test 14: All bits set - priority to bit 7
  Input: 8'b11111111 (0xff)
  Output: encoded_out=7, valid=1
  Expected: encoded_out=7, valid=1
PASS

Test 15: Bits 6-0 set - priority to bit 6
  Input: 8'b01111111 (0x7f)
  Output: encoded_out=6, valid=1
  Expected: encoded_out=6, valid=1
PASS

Test 16: Bits 5-0 set - priority to bit 5
  Input: 8'b00111111 (0x3f)
  Output: encoded_out=5, valid=1
  Expected: encoded_out=5, valid=1
PASS

Test 17: Bits 4-0 set - priority to bit 4
  Input: 8'b00011111 (0x1f)
  Output: encoded_out=4, valid=1
  Expected: encoded_out=4, valid=1
PASS

Test 18: Bits 3-0 set - priority to bit 3
```

```
  Input: 8'b00001111 (0x0f)
  Output: encoded_out=3, valid=1
  Expected: encoded_out=3, valid=1
PASS

Test 19: Bits 2-0 set - priority to bit 2
  Input: 8'b00000111 (0x07)
  Output: encoded_out=2, valid=1
  Expected: encoded_out=2, valid=1
PASS

Test 20: Bits 1-0 set - priority to bit 1
  Input: 8'b00000011 (0x03)
  Output: encoded_out=1, valid=1
  Expected: encoded_out=1, valid=1
PASS

Test 21: Alternating pattern starting with bit 7
  Input: 8'b10101010 (0xaa)
  Output: encoded_out=7, valid=1
  Expected: encoded_out=7, valid=1
PASS

Test 22: Alternating pattern starting with bit 6
  Input: 8'b01010101 (0x55)
  Output: encoded_out=6, valid=1
  Expected: encoded_out=6, valid=1
PASS

Test 23: Pattern 00110011 - priority to bit 5
  Input: 8'b00110011 (0x33)
  Output: encoded_out=5, valid=1
  Expected: encoded_out=5, valid=1
PASS

Test 24: Pattern 00001100 - priority to bit 3
  Input: 8'b00001100 (0x0c)
  Output: encoded_out=3, valid=1
  Expected: encoded_out=3, valid=1
PASS

Test 25: Only upper bits (7,6) - priority to bit 7
  Input: 8'b11000000 (0xc0)
  Output: encoded_out=7, valid=1
  Expected: encoded_out=7, valid=1
PASS

Test 26: Only lower bits (1,0) - priority to bit 1
  Input: 8'b00000011 (0x03)
  Output: encoded_out=1, valid=1
  Expected: encoded_out=1, valid=1
PASS

Test Summary:
============
Total tests: 26
```

```
Passed: 26
Failed: 0
ALL TESTS PASSED!

Priority Encoder Testing Complete!
==================================

================================================================================
Process finished with return code: 0
Removing Chapter_4_examples/example_2__priority_encoder/obj_dir directory...
Chapter_4_examples/example_2__priority_encoder/obj_dir removed successfully.
0
```

Best Practices for if-else

- Always use `begin-end` blocks for multiple statements
- Use `always_comb` for combinational logic
- Use `always_ff` for sequential logic
- Avoid complex nested conditions when possible

Case Statements

Case statements provide a cleaner alternative to multiple if-else statements when comparing a single expression against multiple values.

case Statement

The standard case statement performs exact matching including X and Z values.

```
case (expression)
    value1: statement1;
    value2: statement2;
    value3, value4: statement3; // Multiple values
    default: default_statement;
endcase
```

Example 3: ALU Design

```
// alu.sv
module alu(
    input logic [3:0] opcode,
    input logic [7:0] a, b,
    output logic [7:0] result,
    output logic zero
);
    always_comb begin
        case (opcode)
            4'b0000: result = a + b;        // ADD
            4'b0001: result = a - b;        // SUB
            4'b0010: result = a & b;        // AND
            4'b0011: result = a | b;        // OR
            4'b0100: result = a ^ b;        // XOR
            4'b0101: result = ~a;           // NOT
```

```
            4'b0110: result = a << 1;       // Shift left
            4'b0111: result = a >> 1;       // Shift right
            default: result = 8'h00;
        endcase

        zero = (result == 8'h00);
    end
endmodule
```

```
// alu_testbench.sv
module alu_testbench;  // Testbench module

    // Testbench signals
    logic [3:0] opcode;
    logic [7:0] a, b;
    logic [7:0] result;
    logic zero;

    // Test counters
    integer test_count = 0;
    integer pass_count = 0;

    // ALU operation names for display
    string op_names[8] = '{
        "ADD", "SUB", "AND", "OR", "XOR", "NOT", "SHL", "SHR"
    };

    // Instantiate design under test
    alu DUT (
        .opcode(opcode),
        .a(a),
        .b(b),
        .result(result),
        .zero(zero)
    );

    // Task to run a test case
    task run_test(
        input [3:0] test_opcode,
        input [7:0] test_a, test_b,
        input [7:0] expected_result,
        input expected_zero,
        input string test_description
    );
        test_count++;
        opcode = test_opcode;
        a = test_a;
        b = test_b;
        #1; // Wait for combinational logic to settle

        $display("Test %0d: %s", test_count, test_description);
        if (test_opcode <= 4'b0111) begin
            $display("  Operation: %s (opcode=4'b%04b)", op_names[test_opcode[2:0]], test_opc
        end else begin
```

```
            $display("  Operation: INVALID (opcode=4'b%04b)", test_opcode);
        end
        $display("  Inputs: a=8'h%02h (%0d), b=8'h%02h (%0d)", a, a, b, b);
        $display("  Result: 8'h%02h (%0d), zero=%b", result, result, zero);
        $display("  Expected: 8'h%02h (%0d), zero=%b", expected_result, expected_result, expe

        if (result == expected_result && zero == expected_zero) begin
            $display("PASS");
            pass_count++;
        end else begin
            $display("FAIL");
            $error("Test %0d failed: Expected result=8'h%02h, zero=%b, got result=8'h%02h, ze
                   test_count, expected_result, expected_zero, result, zero);
        end
        $display();
    endtask

    initial begin
        // Dump waves
        $dumpfile("alu_testbench.vcd");       // Specify the VCD file
        $dumpvars(0, alu_testbench);          // Dump all variables in the test module

        $display();
        $display("Hello from testbench!");
        $display("Starting ALU Tests");
        $display("==================");
        $display("8-bit ALU with 4-bit opcode");
        $display("Operations: ADD(0), SUB(1), AND(2), OR(3), XOR(4), NOT(5), SHL(6), SHR(7)")
        $display("Zero flag indicates when result equals 0");
        $display();

        // Test 1: ADD operations
        run_test(4'b0000, 8'h0F, 8'h10, 8'h1F, 1'b0, "ADD: 15 + 16 = 31");
        run_test(4'b0000, 8'hFF, 8'h01, 8'h00, 1'b1, "ADD: 255 + 1 = 0 (overflow, zero flag s
        run_test(4'b0000, 8'h00, 8'h00, 8'h00, 1'b1, "ADD: 0 + 0 = 0 (zero flag set)");
        run_test(4'b0000, 8'h7F, 8'h7F, 8'hFE, 1'b0, "ADD: 127 + 127 = 254");

        // Test 2: SUB operations
        run_test(4'b0001, 8'h20, 8'h10, 8'h10, 1'b0, "SUB: 32 - 16 = 16");
        run_test(4'b0001, 8'h10, 8'h10, 8'h00, 1'b1, "SUB: 16 - 16 = 0 (zero flag set)");
        run_test(4'b0001, 8'h10, 8'h20, 8'hF0, 1'b0, "SUB: 16 - 32 = -16 (underflow)");
        run_test(4'b0001, 8'hFF, 8'h01, 8'hFE, 1'b0, "SUB: 255 - 1 = 254");

        // Test 3: AND operations
        run_test(4'b0010, 8'hFF, 8'hAA, 8'hAA, 1'b0, "AND: 0xFF & 0xAA = 0xAA");
        run_test(4'b0010, 8'hF0, 8'h0F, 8'h00, 1'b1, "AND: 0xF0 & 0x0F = 0x00 (zero flag set)
        run_test(4'b0010, 8'hFF, 8'hFF, 8'hFF, 1'b0, "AND: 0xFF & 0xFF = 0xFF");
        run_test(4'b0010, 8'h55, 8'hAA, 8'h00, 1'b1, "AND: 0x55 & 0xAA = 0x00");

        // Test 4: OR operations
        run_test(4'b0011, 8'hF0, 8'h0F, 8'hFF, 1'b0, "OR: 0xF0 | 0x0F = 0xFF");
        run_test(4'b0011, 8'h00, 8'h00, 8'h00, 1'b1, "OR: 0x00 | 0x00 = 0x00 (zero flag set)"
        run_test(4'b0011, 8'h55, 8'hAA, 8'hFF, 1'b0, "OR: 0x55 | 0xAA = 0xFF");
        run_test(4'b0011, 8'h12, 8'h34, 8'h36, 1'b0, "OR: 0x12 | 0x34 = 0x36");
```

```
        // Test 5: XOR operations
        run_test(4'b0100, 8'hFF, 8'hFF, 8'h00, 1'b1, "XOR: 0xFF ^ 0xFF = 0x00 (zero flag set)
        run_test(4'b0100, 8'h55, 8'hAA, 8'hFF, 1'b0, "XOR: 0x55 ^ 0xAA = 0xFF");
        run_test(4'b0100, 8'hF0, 8'h0F, 8'hFF, 1'b0, "XOR: 0xF0 ^ 0x0F = 0xFF");
        run_test(4'b0100, 8'h12, 8'h12, 8'h00, 1'b1, "XOR: 0x12 ^ 0x12 = 0x00");

        // Test 6: NOT operations (b input ignored)
        run_test(4'b0101, 8'hFF, 8'h00, 8'h00, 1'b1, "NOT: ~0xFF = 0x00 (zero flag set)");
        run_test(4'b0101, 8'h00, 8'hFF, 8'hFF, 1'b0, "NOT: ~0x00 = 0xFF");
        run_test(4'b0101, 8'hAA, 8'h00, 8'h55, 1'b0, "NOT: ~0xAA = 0x55");
        run_test(4'b0101, 8'hF0, 8'h00, 8'h0F, 1'b0, "NOT: ~0xF0 = 0x0F");

        // Test 7: Shift Left operations (b input ignored)
        run_test(4'b0110, 8'h01, 8'h00, 8'h02, 1'b0, "SHL: 0x01 << 1 = 0x02");
        run_test(4'b0110, 8'h80, 8'h00, 8'h00, 1'b1, "SHL: 0x80 << 1 = 0x00 (MSB lost, zero f
        run_test(4'b0110, 8'h55, 8'h00, 8'hAA, 1'b0, "SHL: 0x55 << 1 = 0xAA");
        run_test(4'b0110, 8'h7F, 8'h00, 8'hFE, 1'b0, "SHL: 0x7F << 1 = 0xFE");

        // Test 8: Shift Right operations (b input ignored)
        run_test(4'b0111, 8'h02, 8'h00, 8'h01, 1'b0, "SHR: 0x02 >> 1 = 0x01");
        run_test(4'b0111, 8'h01, 8'h00, 8'h00, 1'b1, "SHR: 0x01 >> 1 = 0x00 (LSB lost, zero f
        run_test(4'b0111, 8'hAA, 8'h00, 8'h55, 1'b0, "SHR: 0xAA >> 1 = 0x55");
        run_test(4'b0111, 8'hFE, 8'h00, 8'h7F, 1'b0, "SHR: 0xFE >> 1 = 0x7F");

        // Test 9: Invalid opcodes (default case)
        run_test(4'b1000, 8'hFF, 8'hFF, 8'h00, 1'b1, "Invalid opcode 8 -> default result 0x00
        run_test(4'b1111, 8'hAA, 8'h55, 8'h00, 1'b1, "Invalid opcode 15 -> default result 0x0

        // Test 10: Edge cases
        run_test(4'b0000, 8'h80, 8'h80, 8'h00, 1'b1, "ADD edge: 0x80 + 0x80 = 0x00 (overflow)
        run_test(4'b0001, 8'h00, 8'h01, 8'hFF, 1'b0, "SUB edge: 0x00 - 0x01 = 0xFF (underflow

        // Summary
        $display("Test Summary:");
        $display("============");
        $display("Total tests: %0d", test_count);
        $display("Passed: %0d", pass_count);
        $display("Failed: %0d", test_count - pass_count);

        if (pass_count == test_count) begin
            $display("ALL TESTS PASSED!");
        end else begin
            $display("Some tests failed. Please review.");
        end

        $display();
        $display("ALU Testing Complete!");
        $display("====================");
        $display();

        $finish;  // End simulation
    end

endmodule
```

Verilator Simulation Output:

```
================================================================================

Hello from testbench!
Starting ALU Tests
==================
8-bit ALU with 4-bit opcode
Operations: ADD(0), SUB(1), AND(2), OR(3), XOR(4), NOT(5), SHL(6), SHR(7)
Zero flag indicates when result equals 0

Test 1: ADD: 15 + 16 = 31
  Operation: ADD (opcode=4'b0000)
  Inputs: a=8'h0f (15), b=8'h10 (16)
  Result: 8'h1f (31), zero=0
  Expected: 8'h1f (31), zero=0
PASS

Test 2: ADD: 255 + 1 = 0 (overflow, zero flag set)
  Operation: ADD (opcode=4'b0000)
  Inputs: a=8'hff (255), b=8'h01 (1)
  Result: 8'h00 (0), zero=1
  Expected: 8'h00 (0), zero=1
PASS

Test 3: ADD: 0 + 0 = 0 (zero flag set)
  Operation: ADD (opcode=4'b0000)
  Inputs: a=8'h00 (0), b=8'h00 (0)
  Result: 8'h00 (0), zero=1
  Expected: 8'h00 (0), zero=1
PASS

Test 4: ADD: 127 + 127 = 254
  Operation: ADD (opcode=4'b0000)
  Inputs: a=8'h7f (127), b=8'h7f (127)
  Result: 8'hfe (254), zero=0
  Expected: 8'hfe (254), zero=0
PASS

Test 5: SUB: 32 - 16 = 16
  Operation: SUB (opcode=4'b0001)
  Inputs: a=8'h20 (32), b=8'h10 (16)
  Result: 8'h10 (16), zero=0
  Expected: 8'h10 (16), zero=0
PASS

Test 6: SUB: 16 - 16 = 0 (zero flag set)
  Operation: SUB (opcode=4'b0001)
  Inputs: a=8'h10 (16), b=8'h10 (16)
  Result: 8'h00 (0), zero=1
  Expected: 8'h00 (0), zero=1
PASS

Test 7: SUB: 16 - 32 = -16 (underflow)
  Operation: SUB (opcode=4'b0001)
  Inputs: a=8'h10 (16), b=8'h20 (32)
  Result: 8'hf0 (240), zero=0
  Expected: 8'hf0 (240), zero=0
```

```
PASS

Test 8: SUB: 255 - 1 = 254
  Operation: SUB (opcode=4'b0001)
  Inputs: a=8'hff (255), b=8'h01 (1)
  Result: 8'hfe (254), zero=0
  Expected: 8'hfe (254), zero=0
PASS

Test 9: AND: 0xFF & 0xAA = 0xAA
  Operation: AND (opcode=4'b0010)
  Inputs: a=8'hff (255), b=8'haa (170)
  Result: 8'haa (170), zero=0
  Expected: 8'haa (170), zero=0
PASS

Test 10: AND: 0xF0 & 0x0F = 0x00 (zero flag set)
  Operation: AND (opcode=4'b0010)
  Inputs: a=8'hf0 (240), b=8'h0f (15)
  Result: 8'h00 (0), zero=1
  Expected: 8'h00 (0), zero=1
PASS

Test 11: AND: 0xFF & 0xFF = 0xFF
  Operation: AND (opcode=4'b0010)
  Inputs: a=8'hff (255), b=8'hff (255)
  Result: 8'hff (255), zero=0
  Expected: 8'hff (255), zero=0
PASS

Test 12: AND: 0x55 & 0xAA = 0x00
  Operation: AND (opcode=4'b0010)
  Inputs: a=8'h55 (85), b=8'haa (170)
  Result: 8'h00 (0), zero=1
  Expected: 8'h00 (0), zero=1
PASS

Test 13: OR: 0xF0 | 0x0F = 0xFF
  Operation: OR (opcode=4'b0011)
  Inputs: a=8'hf0 (240), b=8'h0f (15)
  Result: 8'hff (255), zero=0
  Expected: 8'hff (255), zero=0
PASS

Test 14: OR: 0x00 | 0x00 = 0x00 (zero flag set)
  Operation: OR (opcode=4'b0011)
  Inputs: a=8'h00 (0), b=8'h00 (0)
  Result: 8'h00 (0), zero=1
  Expected: 8'h00 (0), zero=1
PASS

Test 15: OR: 0x55 | 0xAA = 0xFF
  Operation: OR (opcode=4'b0011)
  Inputs: a=8'h55 (85), b=8'haa (170)
  Result: 8'hff (255), zero=0
  Expected: 8'hff (255), zero=0
```

```
PASS

Test 16: OR: 0x12 | 0x34 = 0x36
  Operation: OR (opcode=4'b0011)
  Inputs: a=8'h12 (18), b=8'h34 (52)
  Result: 8'h36 (54), zero=0
  Expected: 8'h36 (54), zero=0
PASS

Test 17: XOR: 0xFF ^ 0xFF = 0x00 (zero flag set)
  Operation: XOR (opcode=4'b0100)
  Inputs: a=8'hff (255), b=8'hff (255)
  Result: 8'h00 (0), zero=1
  Expected: 8'h00 (0), zero=1
PASS

Test 18: XOR: 0x55 ^ 0xAA = 0xFF
  Operation: XOR (opcode=4'b0100)
  Inputs: a=8'h55 (85), b=8'haa (170)
  Result: 8'hff (255), zero=0
  Expected: 8'hff (255), zero=0
PASS

Test 19: XOR: 0xF0 ^ 0x0F = 0xFF
  Operation: XOR (opcode=4'b0100)
  Inputs: a=8'hf0 (240), b=8'h0f (15)
  Result: 8'hff (255), zero=0
  Expected: 8'hff (255), zero=0
PASS

Test 20: XOR: 0x12 ^ 0x12 = 0x00
  Operation: XOR (opcode=4'b0100)
  Inputs: a=8'h12 (18), b=8'h12 (18)
  Result: 8'h00 (0), zero=1
  Expected: 8'h00 (0), zero=1
PASS

Test 21: NOT: ~0xFF = 0x00 (zero flag set)
  Operation: NOT (opcode=4'b0101)
  Inputs: a=8'hff (255), b=8'h00 (0)
  Result: 8'h00 (0), zero=1
  Expected: 8'h00 (0), zero=1
PASS

Test 22: NOT: ~0x00 = 0xFF
  Operation: NOT (opcode=4'b0101)
  Inputs: a=8'h00 (0), b=8'hff (255)
  Result: 8'hff (255), zero=0
  Expected: 8'hff (255), zero=0
PASS

Test 23: NOT: ~0xAA = 0x55
  Operation: NOT (opcode=4'b0101)
  Inputs: a=8'haa (170), b=8'h00 (0)
  Result: 8'h55 (85), zero=0
  Expected: 8'h55 (85), zero=0
```

```
PASS

Test 24: NOT: ~0xF0 = 0x0F
  Operation: NOT (opcode=4'b0101)
  Inputs: a=8'hf0 (240), b=8'h00 (0)
  Result: 8'h0f (15), zero=0
  Expected: 8'h0f (15), zero=0
PASS

Test 25: SHL: 0x01 << 1 = 0x02
  Operation: SHL (opcode=4'b0110)
  Inputs: a=8'h01 (1), b=8'h00 (0)
  Result: 8'h02 (2), zero=0
  Expected: 8'h02 (2), zero=0
PASS

Test 26: SHL: 0x80 << 1 = 0x00 (MSB lost, zero flag set)
  Operation: SHL (opcode=4'b0110)
  Inputs: a=8'h80 (128), b=8'h00 (0)
  Result: 8'h00 (0), zero=1
  Expected: 8'h00 (0), zero=1
PASS

Test 27: SHL: 0x55 << 1 = 0xAA
  Operation: SHL (opcode=4'b0110)
  Inputs: a=8'h55 (85), b=8'h00 (0)
  Result: 8'haa (170), zero=0
  Expected: 8'haa (170), zero=0
PASS

Test 28: SHL: 0x7F << 1 = 0xFE
  Operation: SHL (opcode=4'b0110)
  Inputs: a=8'h7f (127), b=8'h00 (0)
  Result: 8'hfe (254), zero=0
  Expected: 8'hfe (254), zero=0
PASS

Test 29: SHR: 0x02 >> 1 = 0x01
  Operation: SHR (opcode=4'b0111)
  Inputs: a=8'h02 (2), b=8'h00 (0)
  Result: 8'h01 (1), zero=0
  Expected: 8'h01 (1), zero=0
PASS

Test 30: SHR: 0x01 >> 1 = 0x00 (LSB lost, zero flag set)
  Operation: SHR (opcode=4'b0111)
  Inputs: a=8'h01 (1), b=8'h00 (0)
  Result: 8'h00 (0), zero=1
  Expected: 8'h00 (0), zero=1
PASS

Test 31: SHR: 0xAA >> 1 = 0x55
  Operation: SHR (opcode=4'b0111)
  Inputs: a=8'haa (170), b=8'h00 (0)
  Result: 8'h55 (85), zero=0
  Expected: 8'h55 (85), zero=0
```

```
PASS

Test 32: SHR: 0xFE >> 1 = 0x7F
  Operation: SHR (opcode=4'b0111)
  Inputs: a=8'hfe (254), b=8'h00 (0)
  Result: 8'h7f (127), zero=0
  Expected: 8'h7f (127), zero=0
PASS

Test 33: Invalid opcode 8 -> default result 0x00
  Operation: INVALID (opcode=4'b1000)
  Inputs: a=8'hff (255), b=8'hff (255)
  Result: 8'h00 (0), zero=1
  Expected: 8'h00 (0), zero=1
PASS

Test 34: Invalid opcode 15 -> default result 0x00
  Operation: INVALID (opcode=4'b1111)
  Inputs: a=8'haa (170), b=8'h55 (85)
  Result: 8'h00 (0), zero=1
  Expected: 8'h00 (0), zero=1
PASS

Test 35: ADD edge: 0x80 + 0x80 = 0x00 (overflow)
  Operation: ADD (opcode=4'b0000)
  Inputs: a=8'h80 (128), b=8'h80 (128)
  Result: 8'h00 (0), zero=1
  Expected: 8'h00 (0), zero=1
PASS

Test 36: SUB edge: 0x00 - 0x01 = 0xFF (underflow)
  Operation: SUB (opcode=4'b0001)
  Inputs: a=8'h00 (0), b=8'h01 (1)
  Result: 8'hff (255), zero=0
  Expected: 8'hff (255), zero=0
PASS

Test Summary:
============
Total tests: 36
Passed: 36
Failed: 0
ALL TESTS PASSED!

ALU Testing Complete!
====================

================================================================================
Process finished with return code: 0
Removing Chapter_4_examples/example_3__alu/obj_dir directory...
Chapter_4_examples/example_3__alu/obj_dir removed successfully.

0
```

casex Statement

casex treats X and Z as don’t-care values in both the case expression and case items.

```
casex (data)
    4'b1???: // Matches any 4-bit value starting with 1
        result = "starts_with_1";
    4'b?1??: // Matches any 4-bit value with second bit as 1
        result = "second_bit_1";
    default:
        result = "other";
endcase
```

Example 4: Instruction Decoder

```
// instruction_decoder.sv
module instruction_decoder(
    input logic [7:0] instruction,
    output logic [2:0] op_type
);
    always_comb begin
        /* verilator lint_off CASEX */
        casex (instruction)
            8'b000?????: op_type = 3'b001;  // Load instructions
            8'b001?????: op_type = 3'b010;  // Store instructions
            8'b010?????: op_type = 3'b011;  // Arithmetic
            8'b011?????: op_type = 3'b100;  // Logic
            8'b1???????: op_type = 3'b101;  // Branch
            default:     op_type = 3'b000;  // NOP
        endcase
        /* verilator lint_on CASEX */
    end
endmodule
```

```
// instruction_decoder_testbench.sv
module instruction_decoder_testbench;
    // Testbench signals
    logic [7:0] instruction;
    logic [2:0] op_type;

    // Instantiate the design under test
    instruction_decoder DUT (
        .instruction(instruction),
        .op_type(op_type)
    );

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

        $display("Starting Instruction Decoder Test");
        $display("=====================================");
        $display();

        // Test Load instructions (000?????)
```

```
        instruction = 8'b00000000; #1;
        $display("Instruction: %b, Op Type: %b (Expected: 001 - Load)", instruction, op_type)
        instruction = 8'b00011111; #1;
        $display("Instruction: %b, Op Type: %b (Expected: 001 - Load)", instruction, op_type)

        // Test Store instructions (001?????)
        instruction = 8'b00100000; #1;
        $display("Instruction: %b, Op Type: %b (Expected: 010 - Store)", instruction, op_type
        instruction = 8'b00111111; #1;
        $display("Instruction: %b, Op Type: %b (Expected: 010 - Store)", instruction, op_type

        // Test Arithmetic instructions (010?????)
        instruction = 8'b01000000; #1;
        $display("Instruction: %b, Op Type: %b (Expected: 011 - Arithmetic)", instruction, op
        instruction = 8'b01011111; #1;
        $display("Instruction: %b, Op Type: %b (Expected: 011 - Arithmetic)", instruction, op

        // Test Logic instructions (011?????)
        instruction = 8'b01100000; #1;
        $display("Instruction: %b, Op Type: %b (Expected: 100 - Logic)", instruction, op_type
        instruction = 8'b01111111; #1;
        $display("Instruction: %b, Op Type: %b (Expected: 100 - Logic)", instruction, op_type

        // Test Branch instructions (1???????)
        instruction = 8'b10000000; #1;
        $display("Instruction: %b, Op Type: %b (Expected: 101 - Branch)", instruction, op_typ
        instruction = 8'b11111111; #1;
        $display("Instruction: %b, Op Type: %b (Expected: 101 - Branch)", instruction, op_typ
        instruction = 8'b10101010; #1;
        $display("Instruction: %b, Op Type: %b (Expected: 101 - Branch)", instruction, op_typ

        $display();
        $display("Test completed!");
        $display("====================================");

        $finish;
    end

endmodule
```

```
Verilator Simulation Output:
================================================================================
Starting Instruction Decoder Test
====================================

Instruction: 00000000, Op Type: 001 (Expected: 001 - Load)
Instruction: 00011111, Op Type: 001 (Expected: 001 - Load)
Instruction: 00100000, Op Type: 010 (Expected: 010 - Store)
Instruction: 00111111, Op Type: 010 (Expected: 010 - Store)
Instruction: 01000000, Op Type: 011 (Expected: 011 - Arithmetic)
Instruction: 01011111, Op Type: 011 (Expected: 011 - Arithmetic)
Instruction: 01100000, Op Type: 100 (Expected: 100 - Logic)
Instruction: 01111111, Op Type: 100 (Expected: 100 - Logic)
Instruction: 10000000, Op Type: 101 (Expected: 101 - Branch)
Instruction: 11111111, Op Type: 101 (Expected: 101 - Branch)
Instruction: 10101010, Op Type: 101 (Expected: 101 - Branch)
```

```
Test completed!
====================================
================================================================================
Process finished with return code: 0
Removing Chapter_4_examples/example_4__instruction_decoder/obj_dir directory...
Chapter_4_examples/example_4__instruction_decoder/obj_dir removed successfully.

0
```

casez Statement

casez treats only Z as don't-care values (more restrictive than casex).

```
casez (selector)
    4'b1zzz: output = input1;
    4'bz1zz: output = input2;
    default: output = default_val;
endcase
```

Case Statement Guidelines

- Always include a `default` case
- Use casex for don't-care matching
- Use casez when only Z should be treated as don't-care
- Avoid overlapping case items

unique and priority Modifiers

SystemVerilog provides `unique` and `priority` modifiers to specify the intent and improve synthesis results.

unique Modifier

The `unique` modifier indicates that case items are mutually exclusive and exactly one will match.

```
unique case (state)
    IDLE:  next_state = START;
    START: next_state = ACTIVE;
    ACTIVE: next_state = DONE;
    DONE:  next_state = IDLE;
endcase
```

priority Modifier

The `priority` modifier indicates that case items should be evaluated in order, and at least one will match.

```
priority case (1'b1)
    error_flag:     status = ERROR;
    warning_flag:   status = WARNING;
    ready_flag:     status = READY;
    default:        status = IDLE;
endcase
```

Example 5: Finite State Machine

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

module fsm(
    input logic clk, rst_n, start, rw,
    output logic busy, done
);
    state_t current_state, next_state;

    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n)
            current_state <= IDLE;
        else
            current_state <= next_state;
    end

    always_comb begin
        unique case (current_state)
            IDLE: begin
                if (start)
                    next_state = rw ? WRITE : READ;
                else
                    next_state = IDLE;
            end
            READ: next_state = DONE;
            WRITE: next_state = DONE;
            DONE: next_state = IDLE;
        endcase
    end

    assign busy = (current_state != IDLE);
    assign done = (current_state == DONE);
endmodule
```

```
// fsm_testbench.sv
module fsm_testbench;
    // Testbench signals
    logic clk, rst_n, start, rw;
    logic busy, done;

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

    // Instantiate the design under test
    fsm DUT (
```

```
        .clk(clk),
        .rst_n(rst_n),
        .start(start),
        .rw(rw),
        .busy(busy),
        .done(done)
    );

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

        $display();
        $display("Starting FSM Test");
        $display("=================");
        $display("Time\tState\t\tInputs\t\tOutputs");
        $display("    \t     \t\trst_n start rw\tbusy done");
        $display("-----------------------------------------------");

        // Initialize signals
        rst_n = 0;
        start = 0;
        rw = 0;

        // Reset test
        #10;
        $display("%4t\t%s\t%b     %b     %b\t%b    %b",
                 $time, DUT.current_state.name(), rst_n, start, rw, busy, done);

        // Release reset
        rst_n = 1;
        #10;
        $display("%4t\t%s\t%b     %b     %b\t%b    %b",
                 $time, DUT.current_state.name(), rst_n, start, rw, busy, done);

        // Test READ operation
        $display("\n--- Testing READ Operation ---");
        start = 1;
        rw = 0;  // READ
        #10;
        $display("%4t\t%s\t%b     %b     %b\t%b    %b",
                 $time, DUT.current_state.name(), rst_n, start, rw, busy, done);

        start = 0;
        #10;
        $display("%4t\t%s\t%b     %b     %b\t%b    %b",
                 $time, DUT.current_state.name(), rst_n, start, rw, busy, done);

        #10;
        $display("%4t\t%s\t%b     %b     %b\t%b    %b",
                 $time, DUT.current_state.name(), rst_n, start, rw, busy, done);

        #10;
```

```
        $display("%4t\t%s\t%b     %b     %b\t%b    %b",
                 $time, DUT.current_state.name(), rst_n, start, rw, busy, done);

        // Test WRITE operation
        $display("\n--- Testing WRITE Operation ---");
        start = 1;
        rw = 1;  // WRITE
        #10;
        $display("%4t\t%s\t%b     %b     %b\t%b    %b",
                 $time, DUT.current_state.name(), rst_n, start, rw, busy, done);

        start = 0;
        #10;
        $display("%4t\t%s\t%b     %b     %b\t%b    %b",
                 $time, DUT.current_state.name(), rst_n, start, rw, busy, done);

        #10;
        $display("%4t\t%s\t%b     %b     %b\t%b    %b",
                 $time, DUT.current_state.name(), rst_n, start, rw, busy, done);

        #10;
        $display("%4t\t%s\t%b     %b     %b\t%b    %b",
                 $time, DUT.current_state.name(), rst_n, start, rw, busy, done);

        // Test staying in IDLE when start is not asserted
        $display("\n--- Testing IDLE Hold ---");
        start = 0;
        rw = 0;
        #20;
        $display("%4t\t%s\t%b     %b     %b\t%b    %b",
                 $time, DUT.current_state.name(), rst_n, start, rw, busy, done);

        // Test reset during operation
        $display("\n--- Testing Reset During Operation ---");
        start = 1;
        rw = 1;
        #10;
        $display("%4t\t%s\t%b     %b     %b\t%b    %b",
                 $time, DUT.current_state.name(), rst_n, start, rw, busy, done);

        // Assert reset
        rst_n = 0;
        #10;
        $display("%4t\t%s\t%b     %b     %b\t%b    %b",
                 $time, DUT.current_state.name(), rst_n, start, rw, busy, done);

        // Release reset
        rst_n = 1;
        start = 0;
        #10;
        $display("%4t\t%s\t%b     %b     %b\t%b    %b",
                 $time, DUT.current_state.name(), rst_n, start, rw, busy, done);

        $display("\n================");
        $display("Test completed!");
```

```
        $display();

        #20;
        $finish;
    end

endmodule
```

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

Starting FSM Test
=================
Time    State       Inputs      Outputs
                    rst_n start rw  busy done
-----------------------------------------------
  10    IDLE    0     0     0   0    0
  20    IDLE    1     0     0   0    0

--- Testing READ Operation ---
  30    READ    1     1     0   1    0
  40    DONE    1     0     0   1    1
  50    IDLE    1     0     0   0    0
  60    IDLE    1     0     0   0    0

--- Testing WRITE Operation ---
  70    WRITE   1     1     1   1    0
  80    DONE    1     0     1   1    1
  90    IDLE    1     0     1   0    0
 100    IDLE    1     0     1   0    0

--- Testing IDLE Hold ---
 120    IDLE    1     0     0   0    0

--- Testing Reset During Operation ---
 130    WRITE   1     1     1   1    0
 140    IDLE    0     1     1   0    0
 150    IDLE    1     0     1   0    0

=================
Test completed!

================================================================================
Process finished with return code: 0
Removing Chapter_4_examples/example_5__fsm/obj_dir directory...
Chapter_4_examples/example_5__fsm/obj_dir removed successfully.
```

```
0
```

Loop Statements

SystemVerilog provides several loop constructs for different use cases.

for Loop

The `for` loop is used when the number of iterations is known.

```
for (initialization; condition; increment) begin
    // statements
end
```

Example 6: Parallel-to-Serial Converter

```
// parallel_to_serial.sv
module parallel_to_serial(
    input logic clk, rst_n, load,
    input logic [7:0] parallel_in,
    output logic serial_out, done
);
    logic [7:0] shift_reg;
    logic [2:0] count;

    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            shift_reg <= 8'h00;
            count <= 3'd0;
        end else if (load) begin
            shift_reg <= parallel_in;
            count <= 3'd0;
        end else if (count < 3'd7) begin
            shift_reg <= {shift_reg[6:0], 1'b0};
            count <= count + 1'b1;
        end
    end

    assign serial_out = shift_reg[7];
    assign done = (count == 3'd7);
endmodule
```

```
// parallel_to_serial_testbench.sv
module parallel_to_serial_testbench;
    // Testbench signals
    logic clk, rst_n, load;
    logic [7:0] parallel_in;
    logic serial_out, done;

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

    // Instantiate the design under test
    parallel_to_serial DUT (
        .clk(clk),
        .rst_n(rst_n),
        .load(load),
        .parallel_in(parallel_in),
        .serial_out(serial_out),
        .done(done)
```

```
    );

    // Task to display current state
    task display_state(string description);
        $display("%s", description);
        $display("Time: %4t | Load: %b | Parallel_in: %8b (%02h) | Serial_out: %b | Done: %b
                 $time, load, parallel_in, parallel_in, serial_out, done, DUT.count, DUT.shif
        $display("------------------------------------------------------------");
    endtask

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

        $display("Starting Parallel-to-Serial Converter Test");
        $display("==========================================");
        $display();

        // Initialize signals
        rst_n = 0;
        load = 0;
        parallel_in = 8'h00;

        // Reset test
        #10;
        display_state("After Reset:");

        // Release reset
        rst_n = 1;
        #10;
        display_state("Reset Released:");

        // Test 1: Load pattern 10101010 (0xAA)
        $display("\n=== TEST 1: Converting 0xAA (10101010) ===");
        parallel_in = 8'hAA;
        load = 1;
        #10;
        display_state("Data Loaded:");

        load = 0;

        // Shift out all 8 bits
        for (int i = 0; i < 8; i++) begin
            #10;
            $display("Cycle %d: Serial_out = %b, Done = %b, Count = %d, Shift_reg = %8b",
                     i+1, serial_out, done, DUT.count, DUT.shift_reg);
        end

        #10;
        display_state("After all bits shifted:");

        // Test 2: Load pattern 11110000 (0xF0)
        $display("\n=== TEST 2: Converting 0xF0 (11110000) ===");
```

```
        parallel_in = 8'hF0;
        load = 1;
        #10;
        display_state("Data Loaded:");

        load = 0;

        // Shift out all 8 bits
        for (int i = 0; i < 8; i++) begin
            #10;
            $display("Cycle %d: Serial_out = %b, Done = %b, Count = %d, Shift_reg = %8b",
                     i+1, serial_out, done, DUT.count, DUT.shift_reg);
        end

        #10;
        display_state("After all bits shifted:");

        // Test 3: Load new data while shifting (should restart)
        $display("\n=== TEST 3: Load during shifting (0x55 then 0x33) ===");
        parallel_in = 8'h55;  // 01010101
        load = 1;
        #10;
        display_state("First Data Loaded (0x55):");

        load = 0;

        // Shift a few bits
        #10;
        $display("After 1 shift: Serial_out = %b, Count = %d, Shift_reg = %8b",
                 serial_out, DUT.count, DUT.shift_reg);
        #10;
        $display("After 2 shifts: Serial_out = %b, Count = %d, Shift_reg = %8b",
                 serial_out, DUT.count, DUT.shift_reg);

        // Load new data while shifting
        parallel_in = 8'h33;  // 00110011
        load = 1;
        #10;
        display_state("New Data Loaded (0x33) - Should restart:");

        load = 0;

        // Continue shifting the new data
        for (int i = 0; i < 8; i++) begin
            #10;
            $display("Cycle %d: Serial_out = %b, Done = %b, Count = %d, Shift_reg = %8b",
                     i+1, serial_out, done, DUT.count, DUT.shift_reg);
        end

        // Test 4: Reset during operation
        $display("\n=== TEST 4: Reset during shifting ===");
        parallel_in = 8'hC3;  // 11000011
        load = 1;
        #10;
        display_state("Data Loaded (0xC3):");
```

```
        load = 0;

        // Shift a few bits
        #10;
        #10;
        #10;
        $display("After 3 shifts: Serial_out = %b, Count = %d, Shift_reg = %8b",
                 serial_out, DUT.count, DUT.shift_reg);

        // Reset during operation
        rst_n = 0;
        #10;
        display_state("Reset Applied During Operation:");

        rst_n = 1;
        #10;
        display_state("Reset Released:");

        $display("\n=========================================");
        $display("Test completed!");

        #20;
        $finish;
    end

endmodule
```

```
Verilator Simulation Output:
================================================================================
Starting Parallel-to-Serial Converter Test
==========================================

After Reset:
Time:   10 | Load: 0 | Parallel_in: 00000000 (00) | Serial_out: 0 | Done: 0 |
Count: 0 | Shift_reg: 00000000
----------------------------------------------------------
Reset Released:
Time:   20 | Load: 0 | Parallel_in: 00000000 (00) | Serial_out: 0 | Done: 0 |
Count: 1 | Shift_reg: 00000000
----------------------------------------------------------

=== TEST 1: Converting 0xAA (10101010) ===
Data Loaded:
Time:   30 | Load: 1 | Parallel_in: 10101010 (aa) | Serial_out: 1 | Done: 0 |
Count: 0 | Shift_reg: 10101010
----------------------------------------------------------
Cycle           1: Serial_out = 0, Done = 0, Count = 1, Shift_reg = 01010100
Cycle           2: Serial_out = 1, Done = 0, Count = 2, Shift_reg = 10101000
Cycle           3: Serial_out = 0, Done = 0, Count = 3, Shift_reg = 01010000
Cycle           4: Serial_out = 1, Done = 0, Count = 4, Shift_reg = 10100000
Cycle           5: Serial_out = 0, Done = 0, Count = 5, Shift_reg = 01000000
Cycle           6: Serial_out = 1, Done = 0, Count = 6, Shift_reg = 10000000
Cycle           7: Serial_out = 0, Done = 1, Count = 7, Shift_reg = 00000000
Cycle           8: Serial_out = 0, Done = 1, Count = 7, Shift_reg = 00000000
After all bits shifted:
Time:  120 | Load: 0 | Parallel_in: 10101010 (aa) | Serial_out: 0 | Done: 1 |
```

```
Count: 7 | Shift_reg: 00000000
------------------------------------------------------

=== TEST 2: Converting 0xF0 (11110000) ===
Data Loaded:
Time:  130 | Load: 1 | Parallel_in: 11110000 (f0) | Serial_out: 1 | Done: 0 |
Count: 0 | Shift_reg: 11110000
------------------------------------------------------
Cycle           1: Serial_out = 1, Done = 0, Count = 1, Shift_reg = 11100000
Cycle           2: Serial_out = 1, Done = 0, Count = 2, Shift_reg = 11000000
Cycle           3: Serial_out = 1, Done = 0, Count = 3, Shift_reg = 10000000
Cycle           4: Serial_out = 0, Done = 0, Count = 4, Shift_reg = 00000000
Cycle           5: Serial_out = 0, Done = 0, Count = 5, Shift_reg = 00000000
Cycle           6: Serial_out = 0, Done = 0, Count = 6, Shift_reg = 00000000
Cycle           7: Serial_out = 0, Done = 1, Count = 7, Shift_reg = 00000000
Cycle           8: Serial_out = 0, Done = 1, Count = 7, Shift_reg = 00000000
After all bits shifted:
Time:  220 | Load: 0 | Parallel_in: 11110000 (f0) | Serial_out: 0 | Done: 1 |
Count: 7 | Shift_reg: 00000000
------------------------------------------------------

=== TEST 3: Load during shifting (0x55 then 0x33) ===
First Data Loaded (0x55):
Time:  230 | Load: 1 | Parallel_in: 01010101 (55) | Serial_out: 0 | Done: 0 |
Count: 0 | Shift_reg: 01010101
------------------------------------------------------
After 1 shift: Serial_out = 1, Count = 1, Shift_reg = 10101010
After 2 shifts: Serial_out = 0, Count = 2, Shift_reg = 01010100
New Data Loaded (0x33) - Should restart:
Time:  260 | Load: 1 | Parallel_in: 00110011 (33) | Serial_out: 0 | Done: 0 |
Count: 0 | Shift_reg: 00110011
------------------------------------------------------
Cycle           1: Serial_out = 0, Done = 0, Count = 1, Shift_reg = 01100110
Cycle           2: Serial_out = 1, Done = 0, Count = 2, Shift_reg = 11001100
Cycle           3: Serial_out = 1, Done = 0, Count = 3, Shift_reg = 10011000
Cycle           4: Serial_out = 0, Done = 0, Count = 4, Shift_reg = 00110000
Cycle           5: Serial_out = 0, Done = 0, Count = 5, Shift_reg = 01100000
Cycle           6: Serial_out = 1, Done = 0, Count = 6, Shift_reg = 11000000
Cycle           7: Serial_out = 1, Done = 1, Count = 7, Shift_reg = 10000000
Cycle           8: Serial_out = 1, Done = 1, Count = 7, Shift_reg = 10000000

=== TEST 4: Reset during shifting ===
Data Loaded (0xC3):
Time:  350 | Load: 1 | Parallel_in: 11000011 (c3) | Serial_out: 1 | Done: 0 |
Count: 0 | Shift_reg: 11000011
------------------------------------------------------
After 3 shifts: Serial_out = 0, Count = 3, Shift_reg = 00011000
Reset Applied During Operation:
Time:  390 | Load: 0 | Parallel_in: 11000011 (c3) | Serial_out: 0 | Done: 0 |
Count: 0 | Shift_reg: 00000000
------------------------------------------------------
Reset Released:
Time:  400 | Load: 0 | Parallel_in: 11000011 (c3) | Serial_out: 0 | Done: 0 |
Count: 1 | Shift_reg: 00000000
------------------------------------------------------
```

```
==========================================
Test completed!
================================================================================
Process finished with return code: 0
Removing Chapter_4_examples/example_6__parallel_to_serial/obj_dir directory...
Chapter_4_examples/example_6__parallel_to_serial/obj_dir removed successfully.

0
```

Example: Generate Loop for Parameterized Design

```
// ripple_carry_adder.sv

// Full adder module (building block)
module full_adder(
    input logic a, b, cin,
    output logic sum, cout
);
    assign sum = a ^ b ^ cin;
    assign cout = (a & b) | (a & cin) | (b & cin);
endmodule

// Ripple carry adder using generate block
module ripple_carry_adder #(parameter WIDTH = 8)(
    input logic [WIDTH-1:0] a, b,
    input logic cin,
    output logic [WIDTH-1:0] sum,
    output logic cout
);
    logic [WIDTH:0] carry;

    assign carry[0] = cin;

    generate
        for (genvar i = 0; i < WIDTH; i++) begin : adder_stage
            full_adder fa (
                .a(a[i]),
                .b(b[i]),
                .cin(carry[i]),
                .sum(sum[i]),
                .cout(carry[i+1])
            );
        end
    endgenerate

    assign cout = carry[WIDTH];
endmodule
```

```
// ripple_carry_adder_testbench.sv
module ripple_carry_adder_testbench;
    // Parameters for different width testing
    parameter WIDTH_8 = 8;
    parameter WIDTH_4 = 4;

    // Testbench signals for 8-bit adder
```

```
    logic [WIDTH_8-1:0] a8, b8, sum8;
    logic cin8, cout8;

    // Testbench signals for 4-bit adder
    logic [WIDTH_4-1:0] a4, b4, sum4;
    logic cin4, cout4;

    // Expected results
    logic [WIDTH_8:0] expected_result8;
    logic [WIDTH_4:0] expected_result4;

    // Instantiate 8-bit ripple carry adder
    ripple_carry_adder #(.WIDTH(WIDTH_8)) DUT_8bit (
        .a(a8),
        .b(b8),
        .cin(cin8),
        .sum(sum8),
        .cout(cout8)
    );

    // Instantiate 4-bit ripple carry adder
    ripple_carry_adder #(.WIDTH(WIDTH_4)) DUT_4bit (
        .a(a4),
        .b(b4),
        .cin(cin4),
        .sum(sum4),
        .cout(cout4)
    );

    // Task to test 8-bit adder
    task test_8bit_adder(logic [7:0] test_a, logic [7:0] test_b, logic test_cin, string descr
        a8 = test_a;
        b8 = test_b;
        cin8 = test_cin;
        expected_result8 = {1'b0, test_a} + {1'b0, test_b} + {8'b0, test_cin};
        #1; // Wait for combinational logic

        $display("8-bit Test: %s", description);
        $display("  A = %8b (%3d), B = %8b (%3d), Cin = %b", a8, a8, b8, b8, cin8);
        $display("  Sum = %8b (%3d), Cout = %b", sum8, sum8, cout8);
        $display("  Expected: %9b (%3d)", expected_result8, expected_result8);
        $display("  Result: %s", ({cout8, sum8} == expected_result8) ? "PASS" : "FAIL");
        $display("  Carry chain: %b", DUT_8bit.carry);
        $display();
    endtask

    // Task to test 4-bit adder
    task test_4bit_adder(logic [3:0] test_a, logic [3:0] test_b, logic test_cin, string descr
        a4 = test_a;
        b4 = test_b;
        cin4 = test_cin;
        expected_result4 = {1'b0, test_a} + {1'b0, test_b} + {4'b0, test_cin};
        #1; // Wait for combinational logic

        $display("4-bit Test: %s", description);
```

```
        $display("  A = %4b (%2d), B = %4b (%2d), Cin = %b", a4, a4, b4, b4, cin4);
        $display("  Sum = %4b (%2d), Cout = %b", sum4, sum4, cout4);
        $display("  Expected: %5b (%2d)", expected_result4, expected_result4);
        $display("  Result: %s", ({cout4, sum4} == expected_result4) ? "PASS" : "FAIL");
        $display("  Carry chain: %b", DUT_4bit.carry);
        $display();
    endtask

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

        $display("Starting Ripple Carry Adder Test");
        $display("================================");
        $display();

        // === 8-BIT ADDER TESTS ===
        $display("=== 8-BIT RIPPLE CARRY ADDER TESTS ===");
        $display();

        // Basic addition tests
        test_8bit_adder(8'd0, 8'd0, 1'b0, "Zero + Zero");
        test_8bit_adder(8'd15, 8'd10, 1'b0, "15 + 10");
        test_8bit_adder(8'd255, 8'd0, 1'b0, "255 + 0");
        test_8bit_adder(8'd128, 8'd127, 1'b0, "128 + 127");

        // Test with carry in
        test_8bit_adder(8'd100, 8'd50, 1'b1, "100 + 50 + 1 (with carry in)");
        test_8bit_adder(8'd255, 8'd255, 1'b1, "255 + 255 + 1 (maximum with carry)");

        // Overflow tests
        test_8bit_adder(8'd255, 8'd1, 1'b0, "255 + 1 (overflow)");
        test_8bit_adder(8'd200, 8'd100, 1'b0, "200 + 100 (overflow)");

        // Pattern tests
        test_8bit_adder(8'b10101010, 8'b01010101, 1'b0, "Alternating patterns");
        test_8bit_adder(8'b11110000, 8'b00001111, 1'b0, "Complementary patterns");

        $display("=== 4-BIT RIPPLE CARRY ADDER TESTS ===");
        $display();

        // === 4-BIT ADDER TESTS ===
        test_4bit_adder(4'd0, 4'd0, 1'b0, "Zero + Zero");
        test_4bit_adder(4'd7, 4'd8, 1'b0, "7 + 8");
        test_4bit_adder(4'd15, 4'd0, 1'b0, "15 + 0");
        test_4bit_adder(4'd9, 4'd6, 1'b1, "9 + 6 + 1 (with carry in)");
        test_4bit_adder(4'd15, 4'd15, 1'b0, "15 + 15 (maximum)");
        test_4bit_adder(4'd15, 4'd15, 1'b1, "15 + 15 + 1 (maximum with carry)");

        // Overflow tests
        test_4bit_adder(4'd15, 4'd1, 1'b0, "15 + 1 (overflow)");
        test_4bit_adder(4'd10, 4'd8, 1'b0, "10 + 8 (overflow)");
```

```
        // === EXHAUSTIVE 4-BIT TEST ===
        $display("=== EXHAUSTIVE 4-BIT TEST (selected cases) ===");
        $display();

        // Test a few representative cases from exhaustive testing
        for (int i = 0; i < 16; i += 5) begin
            for (int j = 0; j < 16; j += 7) begin
                for (int c = 0; c < 2; c++) begin
                    test_4bit_adder(i[3:0], j[3:0], c[0], $sformatf("Exhaustive: %d + %d + %d
                end
            end
        end

        // === TIMING TEST ===
        $display("=== PROPAGATION DELAY TEST ===");
        $display();

        // Test carry propagation through all stages
        a8 = 8'b11111111;
        b8 = 8'b00000000;
        cin8 = 1'b1;
        $display("Testing carry propagation: 11111111 + 00000000 + 1");
        $display("This should cause carry to ripple through all stages");

        #1;
        $display("Final result: Sum = %8b, Cout = %b", sum8, cout8);
        $display("Carry chain: %b", DUT_8bit.carry);
        $display();

        $display("================================");
        $display("All tests completed!");
        $display("================================");

        $finish;
    end

endmodule
```

```
Verilator Simulation Output:
================================================================================
Starting Ripple Carry Adder Test
================================

=== 8-BIT RIPPLE CARRY ADDER TESTS ===

8-bit Test: Zero + Zero
  A = 00000000 (  0), B = 00000000 (  0), Cin = 0
  Sum = 00000000 (  0), Cout = 0
  Expected: 000000000 (  0)
  Result: PASS
  Carry chain: 000000000

8-bit Test: 15 + 10
  A = 00001111 ( 15), B = 00001010 ( 10), Cin = 0
  Sum = 00011001 ( 25), Cout = 0
  Expected: 000011001 ( 25)
```

```
  Result: PASS
  Carry chain: 000011100

8-bit Test: 255 + 0
  A = 11111111 (255), B = 00000000 (  0), Cin = 0
  Sum = 11111111 (255), Cout = 0
  Expected: 011111111 (255)
  Result: PASS
  Carry chain: 000000000

8-bit Test: 128 + 127
  A = 10000000 (128), B = 01111111 (127), Cin = 0
  Sum = 11111111 (255), Cout = 0
  Expected: 011111111 (255)
  Result: PASS
  Carry chain: 000000000

8-bit Test: 100 + 50 + 1 (with carry in)
  A = 01100100 (100), B = 00110010 ( 50), Cin = 1
  Sum = 10010111 (151), Cout = 0
  Expected: 010010111 (151)
  Result: PASS
  Carry chain: 011000001

8-bit Test: 255 + 255 + 1 (maximum with carry)
  A = 11111111 (255), B = 11111111 (255), Cin = 1
  Sum = 11111111 (255), Cout = 1
  Expected: 111111111 (511)
  Result: PASS
  Carry chain: 111111111

8-bit Test: 255 + 1 (overflow)
  A = 11111111 (255), B = 00000001 (  1), Cin = 0
  Sum = 00000000 (  0), Cout = 1
  Expected: 100000000 (256)
  Result: PASS
  Carry chain: 111111110

8-bit Test: 200 + 100 (overflow)
  A = 11001000 (200), B = 01100100 (100), Cin = 0
  Sum = 00101100 ( 44), Cout = 1
  Expected: 100101100 (300)
  Result: PASS
  Carry chain: 110000000

8-bit Test: Alternating patterns
  A = 10101010 (170), B = 01010101 ( 85), Cin = 0
  Sum = 11111111 (255), Cout = 0
  Expected: 011111111 (255)
  Result: PASS
  Carry chain: 000000000

8-bit Test: Complementary patterns
  A = 11110000 (240), B = 00001111 ( 15), Cin = 0
  Sum = 11111111 (255), Cout = 0
  Expected: 011111111 (255)
```

```
  Result: PASS
  Carry chain: 000000000

=== 4-BIT RIPPLE CARRY ADDER TESTS ===

4-bit Test: Zero + Zero
  A = 0000 ( 0), B = 0000 ( 0), Cin = 0
  Sum = 0000 ( 0), Cout = 0
  Expected: 00000 ( 0)
  Result: PASS
  Carry chain: 00000

4-bit Test: 7 + 8
  A = 0111 ( 7), B = 1000 ( 8), Cin = 0
  Sum = 1111 (15), Cout = 0
  Expected: 01111 (15)
  Result: PASS
  Carry chain: 00000

4-bit Test: 15 + 0
  A = 1111 (15), B = 0000 ( 0), Cin = 0
  Sum = 1111 (15), Cout = 0
  Expected: 01111 (15)
  Result: PASS
  Carry chain: 00000

4-bit Test: 9 + 6 + 1 (with carry in)
  A = 1001 ( 9), B = 0110 ( 6), Cin = 1
  Sum = 0000 ( 0), Cout = 1
  Expected: 10000 (16)
  Result: PASS
  Carry chain: 11111

4-bit Test: 15 + 15 (maximum)
  A = 1111 (15), B = 1111 (15), Cin = 0
  Sum = 1110 (14), Cout = 1
  Expected: 11110 (30)
  Result: PASS
  Carry chain: 11110

4-bit Test: 15 + 15 + 1 (maximum with carry)
  A = 1111 (15), B = 1111 (15), Cin = 1
  Sum = 1111 (15), Cout = 1
  Expected: 11111 (31)
  Result: PASS
  Carry chain: 11111

4-bit Test: 15 + 1 (overflow)
  A = 1111 (15), B = 0001 ( 1), Cin = 0
  Sum = 0000 ( 0), Cout = 1
  Expected: 10000 (16)
  Result: PASS
  Carry chain: 11110

4-bit Test: 10 + 8 (overflow)
  A = 1010 (10), B = 1000 ( 8), Cin = 0
```

```
  Sum = 0010 ( 2), Cout = 1
  Expected: 10010 (18)
  Result: PASS
  Carry chain: 10000

=== EXHAUSTIVE 4-BIT TEST (selected cases) ===

4-bit Test: Exhaustive:           0 +           0 +           0
  A = 0000 ( 0), B = 0000 ( 0), Cin = 0
  Sum = 0000 ( 0), Cout = 0
  Expected: 00000 ( 0)
  Result: PASS
  Carry chain: 00000

4-bit Test: Exhaustive:           0 +           0 +           1
  A = 0000 ( 0), B = 0000 ( 0), Cin = 1
  Sum = 0001 ( 1), Cout = 0
  Expected: 00001 ( 1)
  Result: PASS
  Carry chain: 00001

4-bit Test: Exhaustive:           0 +           7 +           0
  A = 0000 ( 0), B = 0111 ( 7), Cin = 0
  Sum = 0111 ( 7), Cout = 0
  Expected: 00111 ( 7)
  Result: PASS
  Carry chain: 00000

4-bit Test: Exhaustive:           0 +           7 +           1
  A = 0000 ( 0), B = 0111 ( 7), Cin = 1
  Sum = 1000 ( 8), Cout = 0
  Expected: 01000 ( 8)
  Result: PASS
  Carry chain: 01111

4-bit Test: Exhaustive:           0 +          14 +           0
  A = 0000 ( 0), B = 1110 (14), Cin = 0
  Sum = 1110 (14), Cout = 0
  Expected: 01110 (14)
  Result: PASS
  Carry chain: 00000

4-bit Test: Exhaustive:           0 +          14 +           1
  A = 0000 ( 0), B = 1110 (14), Cin = 1
  Sum = 1111 (15), Cout = 0
  Expected: 01111 (15)
  Result: PASS
  Carry chain: 00001

4-bit Test: Exhaustive:           5 +           0 +           0
  A = 0101 ( 5), B = 0000 ( 0), Cin = 0
  Sum = 0101 ( 5), Cout = 0
  Expected: 00101 ( 5)
  Result: PASS
  Carry chain: 00000
```

```
4-bit Test: Exhaustive:           5 +           0 +           1
  A = 0101 ( 5), B = 0000 ( 0), Cin = 1
  Sum = 0110 ( 6), Cout = 0
  Expected: 00110 ( 6)
  Result: PASS
  Carry chain: 00011

4-bit Test: Exhaustive:           5 +           7 +           0
  A = 0101 ( 5), B = 0111 ( 7), Cin = 0
  Sum = 1100 (12), Cout = 0
  Expected: 01100 (12)
  Result: PASS
  Carry chain: 01110

4-bit Test: Exhaustive:           5 +           7 +           1
  A = 0101 ( 5), B = 0111 ( 7), Cin = 1
  Sum = 1101 (13), Cout = 0
  Expected: 01101 (13)
  Result: PASS
  Carry chain: 01111

4-bit Test: Exhaustive:           5 +          14 +           0
  A = 0101 ( 5), B = 1110 (14), Cin = 0
  Sum = 0011 ( 3), Cout = 1
  Expected: 10011 (19)
  Result: PASS
  Carry chain: 11000

4-bit Test: Exhaustive:           5 +          14 +           1
  A = 0101 ( 5), B = 1110 (14), Cin = 1
  Sum = 0100 ( 4), Cout = 1
  Expected: 10100 (20)
  Result: PASS
  Carry chain: 11111

4-bit Test: Exhaustive:          10 +           0 +           0
  A = 1010 (10), B = 0000 ( 0), Cin = 0
  Sum = 1010 (10), Cout = 0
  Expected: 01010 (10)
  Result: PASS
  Carry chain: 00000

4-bit Test: Exhaustive:          10 +           0 +           1
  A = 1010 (10), B = 0000 ( 0), Cin = 1
  Sum = 1011 (11), Cout = 0
  Expected: 01011 (11)
  Result: PASS
  Carry chain: 00001

4-bit Test: Exhaustive:          10 +           7 +           0
  A = 1010 (10), B = 0111 ( 7), Cin = 0
  Sum = 0001 ( 1), Cout = 1
  Expected: 10001 (17)
  Result: PASS
  Carry chain: 11100
```

```
4-bit Test: Exhaustive:          10 +           7 +           1
  A = 1010 (10), B = 0111 ( 7), Cin = 1
  Sum = 0010 ( 2), Cout = 1
  Expected: 10010 (18)
  Result: PASS
  Carry chain: 11111

4-bit Test: Exhaustive:          10 +          14 +           0
  A = 1010 (10), B = 1110 (14), Cin = 0
  Sum = 1000 ( 8), Cout = 1
  Expected: 11000 (24)
  Result: PASS
  Carry chain: 11100

4-bit Test: Exhaustive:          10 +          14 +           1
  A = 1010 (10), B = 1110 (14), Cin = 1
  Sum = 1001 ( 9), Cout = 1
  Expected: 11001 (25)
  Result: PASS
  Carry chain: 11101

4-bit Test: Exhaustive:          15 +           0 +           0
  A = 1111 (15), B = 0000 ( 0), Cin = 0
  Sum = 1111 (15), Cout = 0
  Expected: 01111 (15)
  Result: PASS
  Carry chain: 00000

4-bit Test: Exhaustive:          15 +           0 +           1
  A = 1111 (15), B = 0000 ( 0), Cin = 1
  Sum = 0000 ( 0), Cout = 1
  Expected: 10000 (16)
  Result: PASS
  Carry chain: 11111

4-bit Test: Exhaustive:          15 +           7 +           0
  A = 1111 (15), B = 0111 ( 7), Cin = 0
  Sum = 0110 ( 6), Cout = 1
  Expected: 10110 (22)
  Result: PASS
  Carry chain: 11110

4-bit Test: Exhaustive:          15 +           7 +           1
  A = 1111 (15), B = 0111 ( 7), Cin = 1
  Sum = 0111 ( 7), Cout = 1
  Expected: 10111 (23)
  Result: PASS
  Carry chain: 11111

4-bit Test: Exhaustive:          15 +          14 +           0
  A = 1111 (15), B = 1110 (14), Cin = 0
  Sum = 1101 (13), Cout = 1
  Expected: 11101 (29)
  Result: PASS
  Carry chain: 11100
```

```
4-bit Test: Exhaustive:          15 +          14 +           1
  A = 1111 (15), B = 1110 (14), Cin = 1
  Sum = 1110 (14), Cout = 1
  Expected: 11110 (30)
  Result: PASS
  Carry chain: 11111

=== PROPAGATION DELAY TEST ===

Testing carry propagation: 11111111 + 00000000 + 1
This should cause carry to ripple through all stages
Final result: Sum = 00000000, Cout = 1
Carry chain: 111111111

================================
All tests completed!
================================
================================================================================
Process finished with return code: 0
Removing Chapter_4_examples/example_7__ripple_carry_adder/obj_dir directory...
Chapter_4_examples/example_7__ripple_carry_adder/obj_dir removed successfully.
```

```
0
```

while Loop

The `while` loop continues as long as the condition is true.

```
while (condition) begin
    // statements
end
```

Example 8: Testbench with while Loop

```
// counter_4bit.sv
module counter_4bit (
    input  logic       clk,
    input  logic       rst_n,
    input  logic       enable,
    output logic [3:0] count
);

    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            count <= 4'b0000;
            $display("Counter reset - count = %0d", count);
        end
        else if (enable) begin
            count <= count + 1;
            $display("Counter enabled - count = %0d", count + 1);
        end
        else begin
            $display("Counter disabled - count = %0d", count);
        end
    end
```

```
endmodule
```

```
// counter_4bit_testbench.sv
module counter_4bit_testbench;
    logic clk, rst_n, enable;
    logic [3:0] count;
    integer test_cycles;

    // Instantiate the 4-bit counter design under test
    counter_4bit dut_counter (
        .clk(clk),
        .rst_n(rst_n),
        .enable(enable),
        .count(count)
    );

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

    // Main test sequence
    initial begin
        // Setup waveform dumping
        $dumpfile("counter_4bit_testbench.vcd");
        $dumpvars(0, counter_4bit_testbench);

        // Initialize test signals
        rst_n = 0;
        enable = 0;
        test_cycles = 0;

        $display("=== 4-bit Counter Testbench Started ===");
        $display("Time: %0t", $time);
        $display();

        // Apply reset for 10ns
        #10 rst_n = 1;
        $display("Reset deasserted at time %0t", $time);

        // Enable counting after reset
        #10 enable = 1;
        $display("Counter enabled at time %0t", $time);
        $display();

        // Run test for 20 clock cycles
        while (test_cycles < 20) begin
            @(posedge clk);
            $display("Cycle %2d: count = %2d (0x%h) at time %0t",
                     test_cycles, count, count, $time);
            test_cycles++;

            // Test disable functionality at cycle 10
```

```
            if (test_cycles == 10) begin
                enable = 0;
                $display(">>> Counter disabled at cycle %0d <<<", test_cycles);
            end

            // Re-enable at cycle 15
            if (test_cycles == 15) begin
                enable = 1;
                $display(">>> Counter re-enabled at cycle %0d <<<", test_cycles);
            end
        end

        $display();
        $display("=== Testbench Completed Successfully ===");
        $display("Final count value: %0d", count);
        $display("Total simulation time: %0t", $time);
        $finish;
    end

    // Overflow detection monitor
    always @(posedge clk) begin
        if (rst_n && enable && count == 4'b1111) begin
            $display("*** OVERFLOW WARNING: Counter reached maximum value (15) ***");
        end
    end

    // Value change monitor for debugging
    always @(count) begin
        if (rst_n) begin
            $display("Count changed to: %0d", count);
        end
    end

endmodule
```

```
Verilator Simulation Output:
================================================================================
=== 4-bit Counter Testbench Started ===
Time: 0

Counter reset - count = 0
Reset deasserted at time 10
Counter disabled - count = 0
Counter enabled at time 20

Cycle  0: count =  0 (0x0) at time 25
Counter enabled - count = 1
Count changed to: 1
Cycle  1: count =  1 (0x1) at time 35
Counter enabled - count = 2
Count changed to: 2
Cycle  2: count =  2 (0x2) at time 45
Counter enabled - count = 3
Count changed to: 3
Cycle  3: count =  3 (0x3) at time 55
Counter enabled - count = 4
```

```
Count changed to: 4
Cycle  4: count =  4 (0x4) at time 65
Counter enabled - count = 5
Count changed to: 5
Cycle  5: count =  5 (0x5) at time 75
Counter enabled - count = 6
Count changed to: 6
Cycle  6: count =  6 (0x6) at time 85
Counter enabled - count = 7
Count changed to: 7
Cycle  7: count =  7 (0x7) at time 95
Counter enabled - count = 8
Count changed to: 8
Cycle  8: count =  8 (0x8) at time 105
Counter enabled - count = 9
Count changed to: 9
Cycle  9: count =  9 (0x9) at time 115
>>> Counter disabled at cycle 10 <<<
Counter disabled - count = 9
Cycle 10: count =  9 (0x9) at time 125
Counter disabled - count = 9
Cycle 11: count =  9 (0x9) at time 135
Counter disabled - count = 9
Cycle 12: count =  9 (0x9) at time 145
Counter disabled - count = 9
Cycle 13: count =  9 (0x9) at time 155
Counter disabled - count = 9
Cycle 14: count =  9 (0x9) at time 165
>>> Counter re-enabled at cycle 15 <<<
Counter enabled - count = 10
Count changed to: 10
Cycle 15: count = 10 (0xa) at time 175
Counter enabled - count = 11
Count changed to: 11
Cycle 16: count = 11 (0xb) at time 185
Counter enabled - count = 12
Count changed to: 12
Cycle 17: count = 12 (0xc) at time 195
Counter enabled - count = 13
Count changed to: 13
Cycle 18: count = 13 (0xd) at time 205
Counter enabled - count = 14
Count changed to: 14
Cycle 19: count = 14 (0xe) at time 215

=== Testbench Completed Successfully ===
Final count value: 14
Total simulation time: 215
- counter_4bit_testbench.sv:69: Verilog $finish
Counter enabled - count = 15
================================================================================
Process finished with return code: 0
Removing Chapter_4_examples/example_8__counter_4bit/obj_dir directory...
Chapter_4_examples/example_8__counter_4bit/obj_dir removed successfully.

0
```

do-while Loop

The `do-while` loop executes at least once before checking the condition.

```
do begin
    // statements
end while (condition);
```

Example 9: Random Test Generation

```
// unique_random_generator.sv
module unique_random_generator ();               // Design under test

  // Test class that generates unique consecutive random values
  class random_test;
    rand bit [7:0] data;
    bit [7:0] prev_value;

    // Function to generate values different from previous one
    function void generate_unique_values();
      int success;
      do begin
        success = randomize();
        if (success == 0) begin
          $error("Randomization failed!");
          break;
        end
      end while (data == prev_value);
      prev_value = data;
    endfunction
  endclass

  // Design logic with random test functionality
  initial begin
    random_test rnd_gen;
    bit [7:0] value_history[5];

    $display();                                  // Display empty line
    $display("Hello from design!");             // Display message
    $display("=== Unique Random Value Generation ===");

    // Create instance of random test class
    rnd_gen = new();

    // Generate and display 5 unique consecutive values
    for (int i = 0; i < 5; i++) begin
      rnd_gen.generate_unique_values();
      value_history[i] = rnd_gen.data;

      $display("Generation %0d: Value=0x%02h, Previous=0x%02h",
               i+1,
               rnd_gen.data,
               rnd_gen.prev_value);
    end
```

```
    $display("Design random generation completed!");
  end

endmodule
```

```
// unique_random_generator_testbench.sv
module unique_random_generator_testbench;  // Testbench module
  unique_random_generator UNIQUE_RANDOM_GENERATOR();  // Instantiate design under test

  // Additional testbench-specific random testing
  class random_test;
    rand bit [7:0] data;
    bit [7:0] prev_value;

    function void generate_unique_values();
      int success;
      do begin
        success = randomize();
        if (success == 0) begin
          $error("Randomization failed!");
          break;
        end
      end while (data == prev_value);
      prev_value = data;
    endfunction
  endclass

  initial begin
    random_test tb_rnd_gen;

    // Dump waves
    $dumpfile("unique_random_generator_testbench.vcd");       // Specify the VCD file
    $dumpvars(0, unique_random_generator_testbench);          // Dump all variables in the te
    #1;                                          // Wait for a time unit
    $display("Hello from testbench!");           // Display message
    $display();                                  // Display empty line

    // Testbench-specific random value testing
    $display("=== Testbench Random Value Verification ===");
    tb_rnd_gen = new();

    // Test multiple generations to verify uniqueness
    for (int i = 0; i < 8; i++) begin
      tb_rnd_gen.generate_unique_values();
      $display("TB Test %0d: Generated=0x%02h, Previous=0x%02h, Unique=%s",
               i+1,
               tb_rnd_gen.data,
               tb_rnd_gen.prev_value,
               (i == 0) ? "N/A" : "YES");
      #5; // Small delay between generations
    end

    $display("Testbench verification completed!");
    #10;
    $finish;
```

```
    end

  endmodule
```

```
Verilator Simulation Output:
================================================================================
- Verilator: Walltime 32.008 s (elab=0.003, cvt=0.094, bld=30.907); cpu 0.066 s
on 1 threads; alloced 20.176 MB

Hello from design!
=== Unique Random Value Generation ===
Generation 1: Value=0xc4, Previous=0xc4
Generation 2: Value=0x9c, Previous=0x9c
Generation 3: Value=0x02, Previous=0x02
Generation 4: Value=0xe4, Previous=0xe4
Generation 5: Value=0x78, Previous=0x78
Design random generation completed!
Hello from testbench!

=== Testbench Random Value Verification ===
TB Test 1: Generated=0x9c, Previous=0x9c, Unique=N/A
TB Test 2: Generated=0x02, Previous=0x02, Unique=YES
TB Test 3: Generated=0xe4, Previous=0xe4, Unique=YES
TB Test 4: Generated=0x78, Previous=0x78, Unique=YES
TB Test 5: Generated=0xbc, Previous=0xbc, Unique=YES
TB Test 6: Generated=0xb6, Previous=0xb6, Unique=YES
TB Test 7: Generated=0xe4, Previous=0xe4, Unique=YES
TB Test 8: Generated=0xb7, Previous=0xb7, Unique=YES
Testbench verification completed!
================================================================================
Process finished with return code: 0
Removing Chapter_4_examples/example_9__unique_random_generator/obj_dir directory...
Chapter_4_examples/example_9__unique_random_generator/obj_dir removed successfully.

0
```

foreach Loops

The `foreach` loop iterates over arrays, providing a clean syntax for array operations.

```
foreach (array_name[i]) begin
    // statements using array_name[i]
end
```

Example 10: Array Processing

```
// array_processor.sv
module array_processor ();                    // Design under test

  // Array processing class with various operations
  class array_operations;
    logic [7:0] data_array[16];
    logic [15:0] sum;                         // Fixed: Increased to 16-bit to prevent overfl
    logic [7:0] avg;
```

```
    logic [7:0] min_val;
    logic [7:0] max_val;
    integer array_size;

    function new();
      array_size = 16;
      initialize_array();
    endfunction

    // Function to initialize array with pattern
    function void initialize_array();
      foreach (data_array[i]) begin
        data_array[i] = 8'(i * 2 + 1);  // Odd numbers: 1, 3, 5, 7, ...
      end
    endfunction

    // Function to calculate sum of array elements
    function void calculate_sum();
      sum = 16'h0000;
      foreach (data_array[i]) begin
        sum = sum + 16'(data_array[i]);  // Fixed: Explicit cast and assignment to avoid widt
      end
    endfunction

    // Function to calculate average
    function void calculate_average();
      calculate_sum();
      avg = 8'(sum / array_size);             // Fixed: Now uses correct sum value
    endfunction

    // Function to find minimum and maximum values
    function void find_min_max();
      min_val = data_array[0];
      max_val = data_array[0];

      foreach (data_array[i]) begin
        if (data_array[i] < min_val) min_val = data_array[i];
        if (data_array[i] > max_val) max_val = data_array[i];
      end
    endfunction

    // Function to display array contents
    function void display_array();
      $display("Array contents:");
      foreach (data_array[i]) begin
        $display("  data_array[%0d] = %0d (0x%02h)", i, data_array[i], data_array[i]);
      end
    endfunction

    // Function to display statistics
    function void display_statistics();
      $display("Array Statistics:");
      $display("  Size: %0d elements", array_size);
      $display("  Sum:  %0d", sum);
      $display("  Avg:  %0d", avg);
```

```
      $display("  Min:  %0d", min_val);
      $display("  Max:  %0d", max_val);
    endfunction
  endclass

  // Design logic with array processing functionality
  initial begin
    array_operations arr_proc;

    $display();                                  // Display empty line
    $display("Hello from design!");             // Display message
    $display("=== Array Processing Operations ===");

    // Create instance of array operations class
    arr_proc = new();

    // Display initial array
    arr_proc.display_array();
    $display();

    // Perform array operations
    arr_proc.calculate_sum();
    arr_proc.calculate_average();
    arr_proc.find_min_max();

    // Display results
    arr_proc.display_statistics();

    $display("Design array processing completed!");
  end

endmodule
```

```
// array_processor_testbench.sv
module array_processor_testbench;              // Testbench module
  array_processor ARRAY_PROCESSOR();           // Instantiate design under test

  // Extended testbench class for additional array testing
  class array_test;
    logic [7:0] test_array[16];
    logic [15:0] expected_sum;                 // Fixed: Increased to 16-bit to prevent overfl
    logic [15:0] actual_sum;                   // Fixed: Increased to 16-bit to prevent overfl
    integer test_size;

    function new();
      test_size = 16;
    endfunction

    // Function to initialize test array with different pattern
    function void initialize_test_array();
      foreach (test_array[i]) begin
        // Fixed: Proper casting and clearer logic
        if (i % 2 == 0) begin
          test_array[i] = 8'(i * 3);           // Even indices: 0, 6, 12, 18, ...
```

```
      end else begin
        test_array[i] = 8'(i + 10);        // Odd indices: 11, 12, 13, 14, ...
      end
    end
  endfunction

  // Function to calculate expected sum
  function void calculate_expected_sum();
    expected_sum = 16'h0000;
    foreach (test_array[i]) begin
      expected_sum = expected_sum + 16'(test_array[i]);  // Fixed: Explicit cast and assign
    end
  endfunction

  // Function to verify array operations
  function bit verify_sum();
    actual_sum = 16'h0000;
    foreach (test_array[i]) begin
      actual_sum = actual_sum + 16'(test_array[i]);    // Fixed: Explicit cast and assignme
    end
    return (actual_sum == expected_sum);
  endfunction

  // Function to display test array
  function void display_test_array();
    $display("Test Array Pattern:");
    foreach (test_array[i]) begin
      $display("  test_array[%0d] = %0d (0x%02h)", i, test_array[i], test_array[i]);
    end
  endfunction

  // Function to run verification
  function void run_verification();
    bit sum_check;

    // Array is already initialized, just calculate and verify
    calculate_expected_sum();
    sum_check = verify_sum();

    $display("=== Verification Results ===");
    $display("Expected Sum: %0d", expected_sum);
    $display("Actual Sum:   %0d", actual_sum);
    $display("Sum Check:    %s", sum_check ? "PASS" : "FAIL");

    if (sum_check) begin
      $display("Array sum verification PASSED");
    end else begin
      $display("Array sum verification FAILED");
    end
  endfunction
endclass

initial begin
  array_test tb_arr_test;
```

```
    // Dump waves
    $dumpfile("array_processor_testbench.vcd");    // Specify the VCD file
    $dumpvars(0, array_processor_testbench);       // Dump all variables in the test module
    #10;                                            // Wait for 10 time units
    $display("Hello from testbench!");             // Display message
    $display();                                    // Display empty line

    // Testbench-specific array testing
    $display("=== Testbench Array Verification ===");
    tb_arr_test = new();

    // Initialize test array first, then display
    tb_arr_test.initialize_test_array();
    tb_arr_test.display_test_array();
    $display();

    // Run verification tests
    tb_arr_test.run_verification();

    $display();
    $display("=== Additional Array Tests ===");

    // Test with different array sizes conceptually
    for (int test_case = 1; test_case <= 3; test_case++) begin
      $display("Test Case %0d: Running array operations...", test_case);

      // Simulate different processing scenarios
      case (test_case)
        1: $display("  Processing sequential data pattern");
        2: $display("  Processing alternating data pattern");
        3: $display("  Processing random-like data pattern");
      endcase

      #5; // Small delay between test cases
    end

    $display("Testbench verification completed!");
    #10;
    $display();
    $finish;
  end

endmodule
```

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

Hello from design!
=== Array Processing Operations ===
Array contents:
  data_array[0] = 1 (0x01)
  data_array[1] = 3 (0x03)
  data_array[2] = 5 (0x05)
  data_array[3] = 7 (0x07)
  data_array[4] = 9 (0x09)
  data_array[5] = 11 (0x0b)
```

```
  data_array[6] = 13 (0x0d)
  data_array[7] = 15 (0x0f)
  data_array[8] = 17 (0x11)
  data_array[9] = 19 (0x13)
  data_array[10] = 21 (0x15)
  data_array[11] = 23 (0x17)
  data_array[12] = 25 (0x19)
  data_array[13] = 27 (0x1b)
  data_array[14] = 29 (0x1d)
  data_array[15] = 31 (0x1f)

Array Statistics:
  Size: 16 elements
  Sum:  256
  Avg:  16
  Min:  1
  Max:  31
Design array processing completed!
Hello from testbench!

=== Testbench Array Verification ===
Test Array Pattern:
  test_array[0] = 0 (0x00)
  test_array[1] = 11 (0x0b)
  test_array[2] = 6 (0x06)
  test_array[3] = 13 (0x0d)
  test_array[4] = 12 (0x0c)
  test_array[5] = 15 (0x0f)
  test_array[6] = 18 (0x12)
  test_array[7] = 17 (0x11)
  test_array[8] = 24 (0x18)
  test_array[9] = 19 (0x13)
  test_array[10] = 30 (0x1e)
  test_array[11] = 21 (0x15)
  test_array[12] = 36 (0x24)
  test_array[13] = 23 (0x17)
  test_array[14] = 42 (0x2a)
  test_array[15] = 25 (0x19)

=== Verification Results ===
Expected Sum: 312
Actual Sum:   312
Sum Check:    PASS
Array sum verification PASSED

=== Additional Array Tests ===
Test Case 1: Running array operations...
  Processing sequential data pattern
Test Case 2: Running array operations...
  Processing alternating data pattern
Test Case 3: Running array operations...
  Processing random-like data pattern
Testbench verification completed!

================================================================================
Process finished with return code: 0
```

```
Removing Chapter_4_examples/example_10__array_processor/obj_dir directory...
Chapter_4_examples/example_10__array_processor/obj_dir removed successfully.
0
```

Example 11: Multi-dimensional Array

```
// matrix_processor.sv
module matrix_processor ();              // Matrix operations design under test
    logic [7:0] matrix[4][4];
    logic [7:0] row_sum[4];

    initial begin
        $display();
        $display("Matrix Processor: Starting operations...");

        // Initialize matrix
        foreach (matrix[i]) begin
            foreach (matrix[i][j]) begin
                matrix[i][j] = 8'(i + j);
            end
        end

        // Calculate row sums
        foreach (row_sum[i]) begin
            row_sum[i] = 0;
            foreach (matrix[i][j]) begin
                row_sum[i] += matrix[i][j];
            end
        end

        // Display results
        $display("Matrix values and row sums:");
        foreach (row_sum[i]) begin
            $display("Row %0d sum = %0d", i, row_sum[i]);
        end

        $display("Matrix Processor: Operations completed.");
    end
endmodule
```

```
// matrix_processor_testbench.sv
module matrix_processor_testbench;  // Testbench module
  matrix_processor MATRIX_DUT();    // Instantiate matrix processor design under test

  initial begin
    // Dump waves
    $dumpfile("matrix_processor_testbench.vcd");    // Specify the VCD file
    $dumpvars(0, matrix_processor_testbench);       // Dump all variables in the test module
    #1;                                             // Wait for a time unit

    $display("Testbench: Starting matrix processor validation...");
    $display();                                     // Display empty line

    // Wait for design to complete
```

```
    #10;

    $display("Testbench: Matrix processor validation completed!");
  end

endmodule
```

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

Matrix Processor: Starting operations...
Matrix values and row sums:
Row 0 sum = 6
Row 1 sum = 10
Row 2 sum = 14
Row 3 sum = 18
Matrix Processor: Operations completed.
Testbench: Starting matrix processor validation...

================================================================================
Process finished with return code: 0
Removing Chapter_4_examples/example_11__matrix_processor/obj_dir directory...
Chapter_4_examples/example_11__matrix_processor/obj_dir removed successfully.
0
```

repeat Statements

The repeat statement executes a block a specified number of times.

```
repeat (expression) begin
    // statements
end
```

Example 12: Clock Generation

```
// clock_generator.sv
module clock_generator ();              // Clock generator design under test
    logic clk;

    initial begin
        $display("Clock Generator: Initializing clock signal...");
        clk = 0;

        $display("Clock Generator: Starting 100 clock cycles...");
        repeat (100) begin
            #5 clk = ~clk;
            #5 clk = ~clk;
        end

        $display("Clock Generator: Generated 100 clock cycles");
        $display("Clock Generator: Simulation completed.");
        $finish;
    end
```

```
endmodule
```

```
// clock_generator_testbench.sv
module clock_generator_testbench;  // Testbench module
  clock_generator CLK_GEN_DUT();   // Instantiate clock generator design under test

  initial begin
    // Dump waves
    $dumpfile("clock_generator_testbench.vcd");     // Specify the VCD file
    $dumpvars(0, clock_generator_testbench);        // Dump all variables in the test module
    #1;                                             // Wait for a time unit
    $display("Testbench: Starting clock generator validation...");

    // Monitor clock transitions
    $monitor("Time: %0t, Clock: %b", $time, CLK_GEN_DUT.clk);

    $display();                                     // Display empty line
    $display("Testbench: Clock generator validation initiated!");
    $display();                                     // Display empty line
  end

endmodule
```

```
Verilator Simulation Output:
================================================================================
Clock Generator: Initializing clock signal...
Clock Generator: Starting 100 clock cycles...
Testbench: Starting clock generator validation...

Testbench: Clock generator validation initiated!

Time: 5, Clock: 1
Time: 10, Clock: 0
Time: 15, Clock: 1
Time: 20, Clock: 0
Time: 25, Clock: 1
Time: 30, Clock: 0
Time: 35, Clock: 1
Time: 40, Clock: 0
Time: 45, Clock: 1
Time: 50, Clock: 0
Time: 55, Clock: 1
Time: 60, Clock: 0
Time: 65, Clock: 1
Time: 70, Clock: 0
Time: 75, Clock: 1
Time: 80, Clock: 0
Time: 85, Clock: 1
Time: 90, Clock: 0
Time: 95, Clock: 1
Time: 100, Clock: 0
Time: 105, Clock: 1
Time: 110, Clock: 0
Time: 115, Clock: 1
Time: 120, Clock: 0
```

```
Time: 125, Clock: 1
Time: 130, Clock: 0
Time: 135, Clock: 1
Time: 140, Clock: 0
Time: 145, Clock: 1
Time: 150, Clock: 0
Time: 155, Clock: 1
Time: 160, Clock: 0
Time: 165, Clock: 1
Time: 170, Clock: 0
Time: 175, Clock: 1
Time: 180, Clock: 0
Time: 185, Clock: 1
Time: 190, Clock: 0
Time: 195, Clock: 1
Time: 200, Clock: 0
Time: 205, Clock: 1
Time: 210, Clock: 0
Time: 215, Clock: 1
Time: 220, Clock: 0
Time: 225, Clock: 1
Time: 230, Clock: 0
Time: 235, Clock: 1
Time: 240, Clock: 0
Time: 245, Clock: 1
Time: 250, Clock: 0
Time: 255, Clock: 1
Time: 260, Clock: 0
Time: 265, Clock: 1
Time: 270, Clock: 0
Time: 275, Clock: 1
Time: 280, Clock: 0
Time: 285, Clock: 1
Time: 290, Clock: 0
Time: 295, Clock: 1
Time: 300, Clock: 0
Time: 305, Clock: 1
Time: 310, Clock: 0
Time: 315, Clock: 1
Time: 320, Clock: 0
Time: 325, Clock: 1
Time: 330, Clock: 0
Time: 335, Clock: 1
Time: 340, Clock: 0
Time: 345, Clock: 1
Time: 350, Clock: 0
Time: 355, Clock: 1
Time: 360, Clock: 0
Time: 365, Clock: 1
Time: 370, Clock: 0
Time: 375, Clock: 1
Time: 380, Clock: 0
Time: 385, Clock: 1
Time: 390, Clock: 0
Time: 395, Clock: 1
Time: 400, Clock: 0
```

```
Time: 405, Clock: 1
Time: 410, Clock: 0
Time: 415, Clock: 1
Time: 420, Clock: 0
Time: 425, Clock: 1
Time: 430, Clock: 0
Time: 435, Clock: 1
Time: 440, Clock: 0
Time: 445, Clock: 1
Time: 450, Clock: 0
Time: 455, Clock: 1
Time: 460, Clock: 0
Time: 465, Clock: 1
Time: 470, Clock: 0
Time: 475, Clock: 1
Time: 480, Clock: 0
Time: 485, Clock: 1
Time: 490, Clock: 0
Time: 495, Clock: 1
Time: 500, Clock: 0
Time: 505, Clock: 1
Time: 510, Clock: 0
Time: 515, Clock: 1
Time: 520, Clock: 0
Time: 525, Clock: 1
Time: 530, Clock: 0
Time: 535, Clock: 1
Time: 540, Clock: 0
Time: 545, Clock: 1
Time: 550, Clock: 0
Time: 555, Clock: 1
Time: 560, Clock: 0
Time: 565, Clock: 1
Time: 570, Clock: 0
Time: 575, Clock: 1
Time: 580, Clock: 0
Time: 585, Clock: 1
Time: 590, Clock: 0
Time: 595, Clock: 1
Time: 600, Clock: 0
Time: 605, Clock: 1
Time: 610, Clock: 0
Time: 615, Clock: 1
Time: 620, Clock: 0
Time: 625, Clock: 1
Time: 630, Clock: 0
Time: 635, Clock: 1
Time: 640, Clock: 0
Time: 645, Clock: 1
Time: 650, Clock: 0
Time: 655, Clock: 1
Time: 660, Clock: 0
Time: 665, Clock: 1
Time: 670, Clock: 0
Time: 675, Clock: 1
Time: 680, Clock: 0
```

```
Time: 685, Clock: 1
Time: 690, Clock: 0
Time: 695, Clock: 1
Time: 700, Clock: 0
Time: 705, Clock: 1
Time: 710, Clock: 0
Time: 715, Clock: 1
Time: 720, Clock: 0
Time: 725, Clock: 1
Time: 730, Clock: 0
Time: 735, Clock: 1
Time: 740, Clock: 0
Time: 745, Clock: 1
Time: 750, Clock: 0
Time: 755, Clock: 1
Time: 760, Clock: 0
Time: 765, Clock: 1
Time: 770, Clock: 0
Time: 775, Clock: 1
Time: 780, Clock: 0
Time: 785, Clock: 1
Time: 790, Clock: 0
Time: 795, Clock: 1
Time: 800, Clock: 0
Time: 805, Clock: 1
Time: 810, Clock: 0
Time: 815, Clock: 1
Time: 820, Clock: 0
Time: 825, Clock: 1
Time: 830, Clock: 0
Time: 835, Clock: 1
Time: 840, Clock: 0
Time: 845, Clock: 1
Time: 850, Clock: 0
Time: 855, Clock: 1
Time: 860, Clock: 0
Time: 865, Clock: 1
Time: 870, Clock: 0
Time: 875, Clock: 1
Time: 880, Clock: 0
Time: 885, Clock: 1
Time: 890, Clock: 0
Time: 895, Clock: 1
Time: 900, Clock: 0
Time: 905, Clock: 1
Time: 910, Clock: 0
Time: 915, Clock: 1
Time: 920, Clock: 0
Time: 925, Clock: 1
Time: 930, Clock: 0
Time: 935, Clock: 1
Time: 940, Clock: 0
Time: 945, Clock: 1
Time: 950, Clock: 0
Time: 955, Clock: 1
Time: 960, Clock: 0
```

```
Time: 965, Clock: 1
Time: 970, Clock: 0
Time: 975, Clock: 1
Time: 980, Clock: 0
Time: 985, Clock: 1
Time: 990, Clock: 0
Time: 995, Clock: 1
Clock Generator: Generated 100 clock cycles
Clock Generator: Simulation completed.
- clock_generator.sv:17: Verilog $finish
================================================================================
Process finished with return code: 0
Removing Chapter_4_examples/example_12__clock_generator/obj_dir directory...
Chapter_4_examples/example_12__clock_generator/obj_dir removed successfully.
0
```

Example 13: Shift Register Test

```
// shift_register.sv
module shift_register (
    input  logic       clk,
    input  logic       rst_n,
    input  logic       serial_in,
    output logic [7:0] parallel_out
);

    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            parallel_out <= 8'h00;
            $display("Shift Register: Reset - parallel_out cleared");
        end else begin
            parallel_out <= {parallel_out[6:0], serial_in};
            $display("Shift Register: Shifted in %b, parallel_out = %b", serial_in, {parallel
        end
    end

endmodule
```

```
// shift_register_testbench.sv
module shift_register_testbench;    // Testbench module
    logic clk, rst_n, serial_in;
    logic [7:0] parallel_out;
    logic [7:0] test_pattern = 8'b10110011;

    shift_register SHIFT_REG_DUT (.*);  // Instantiate shift register design under test

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

    // Main test sequence
    initial begin
```

```
        // Dump waves
        $dumpfile("shift_register_testbench.vcd");      // Specify the VCD file
        $dumpvars(0, shift_register_testbench);         // Dump all variables in the test mod

        $display();                                     // Display empty line
        $display("Testbench: Starting shift register validation...");
        $display();                                     // Display empty line

        rst_n = 0;
        serial_in = 0;

        #10 rst_n = 1;
        $display("Testbench: Reset released, starting shift operations...");

        // Shift in test pattern 10110011
        for (int i = 7; i >= 0; i--) begin
            @(posedge clk);
            serial_in = test_pattern[i];
        end

        @(posedge clk);
        $display();                                     // Display empty line
        $display("Testbench: Final parallel output: %b", parallel_out);
        $display("Testbench: Shift register validation completed!");
        $display();                                     // Display empty line
        $finish;
    end

endmodule
```

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

Testbench: Starting shift register validation...

Shift Register: Reset - parallel_out cleared
Testbench: Reset released, starting shift operations...
Shift Register: Shifted in 1, parallel_out = 00000001
Shift Register: Shifted in 0, parallel_out = 00000010
Shift Register: Shifted in 1, parallel_out = 00000101
Shift Register: Shifted in 1, parallel_out = 00001011
Shift Register: Shifted in 0, parallel_out = 00010110
Shift Register: Shifted in 0, parallel_out = 00101100
Shift Register: Shifted in 1, parallel_out = 01011001
Shift Register: Shifted in 1, parallel_out = 10110011

Testbench: Final parallel output: 10110011
Testbench: Shift register validation completed!

- shift_register_testbench.sv:42: Verilog $finish
================================================================================
Process finished with return code: 0
Removing Chapter_4_examples/example_13__shift_register/obj_dir directory...
Chapter_4_examples/example_13__shift_register/obj_dir removed successfully.

0
```

break and continue Statements

SystemVerilog supports `break` and `continue` statements for loop control.

break Statement

The `break` statement exits the innermost loop immediately.

```
for (int i = 0; i < 100; i++) begin
    if (error_condition)
        break;
    // normal processing
end
```

continue Statement

The `continue` statement skips the rest of the current iteration and continues with the next iteration.

```
for (int i = 0; i < 100; i++) begin
    if (skip_condition)
        continue;
    // processing for valid iterations
end
```

Example 14: Data Validation Loop

```
// data_validator.sv
module data_validator ();               // Data validation processor design under test
    logic [7:0] data_stream[100];
    logic [7:0] valid_data[$];

    initial begin
        $display("Data Validator: Starting data validation process...");

        // Initialize test data with fixed seed approach
        for (int i = 0; i < 100; i++) begin
            data_stream[i] = 8'((i * 17 + 42) % 256);  // Generate pseudo-random pattern
        end

        $display();
        $display("Data Validator: Test data initialized, beginning validation...");

        // Process data with validation
        foreach (data_stream[i]) begin
            // Skip invalid data (value 0 or 255)
            if (data_stream[i] == 0 || data_stream[i] == 255) begin
                $display("Data Validator: Skipping invalid data at index %0d: %0d",
                         i, data_stream[i]);
                continue;
            end

            // Break on error pattern (redundant check since 255 is already filtered above)
            if (data_stream[i] == 8'hFF) begin
                $display("Data Validator: Error pattern detected at index %0d", i);
```

```
                break;
            end

            // Store valid data
            valid_data.push_back(data_stream[i]);
            $display("Data Validator: Valid data stored at index %0d: %0d", i, data_stream[i]
        end

        $display("Data Validator: Processing completed!");
        $display("Data Validator: Processed %0d valid data items out of 100 total", valid_dat

        // Display first few valid items for verification
        $display("Data Validator: First 10 valid items:");
        for (int i = 0; i < 10 && i < valid_data.size(); i++) begin
            $display("  valid_data[%0d] = %0d", i, valid_data[i]);
        end

        $display();

    end
endmodule
```

```
// data_validator_testbench.sv
module data_validator_testbench;  // Testbench module
  data_validator DATA_VAL_DUT();  // Instantiate data validator design under test

  initial begin
    // Dump waves
    $dumpfile("data_validator_testbench.vcd");      // Specify the VCD file
    $dumpvars(0, data_validator_testbench);         // Dump all variables in the test module
    #1;                                             // Wait for a time unit

    $display();                                     // Display empty line
    $display("Testbench: Starting data validator verification...");
    $display();                                     // Display empty line

    // Wait for design to complete processing
    #100;

    $display("Testbench: Data validator verification completed!");
    $display("Testbench: Check output for validation results and statistics.");
    $display();                                     // Display empty line
  end

endmodule
```

```
Verilator Simulation Output:
================================================================================
Data Validator: Starting data validation process...

Data Validator: Test data initialized, beginning validation...
Data Validator: Valid data stored at index 0: 42
Data Validator: Valid data stored at index 1: 59
Data Validator: Valid data stored at index 2: 76
Data Validator: Valid data stored at index 3: 93
```

```
Data Validator: Valid data stored at index 4: 110
Data Validator: Valid data stored at index 5: 127
Data Validator: Valid data stored at index 6: 144
Data Validator: Valid data stored at index 7: 161
Data Validator: Valid data stored at index 8: 178
Data Validator: Valid data stored at index 9: 195
Data Validator: Valid data stored at index 10: 212
Data Validator: Valid data stored at index 11: 229
Data Validator: Valid data stored at index 12: 246
Data Validator: Valid data stored at index 13: 7
Data Validator: Valid data stored at index 14: 24
Data Validator: Valid data stored at index 15: 41
Data Validator: Valid data stored at index 16: 58
Data Validator: Valid data stored at index 17: 75
Data Validator: Valid data stored at index 18: 92
Data Validator: Valid data stored at index 19: 109
Data Validator: Valid data stored at index 20: 126
Data Validator: Valid data stored at index 21: 143
Data Validator: Valid data stored at index 22: 160
Data Validator: Valid data stored at index 23: 177
Data Validator: Valid data stored at index 24: 194
Data Validator: Valid data stored at index 25: 211
Data Validator: Valid data stored at index 26: 228
Data Validator: Valid data stored at index 27: 245
Data Validator: Valid data stored at index 28: 6
Data Validator: Valid data stored at index 29: 23
Data Validator: Valid data stored at index 30: 40
Data Validator: Valid data stored at index 31: 57
Data Validator: Valid data stored at index 32: 74
Data Validator: Valid data stored at index 33: 91
Data Validator: Valid data stored at index 34: 108
Data Validator: Valid data stored at index 35: 125
Data Validator: Valid data stored at index 36: 142
Data Validator: Valid data stored at index 37: 159
Data Validator: Valid data stored at index 38: 176
Data Validator: Valid data stored at index 39: 193
Data Validator: Valid data stored at index 40: 210
Data Validator: Valid data stored at index 41: 227
Data Validator: Valid data stored at index 42: 244
Data Validator: Valid data stored at index 43: 5
Data Validator: Valid data stored at index 44: 22
Data Validator: Valid data stored at index 45: 39
Data Validator: Valid data stored at index 46: 56
Data Validator: Valid data stored at index 47: 73
Data Validator: Valid data stored at index 48: 90
Data Validator: Valid data stored at index 49: 107
Data Validator: Valid data stored at index 50: 124
Data Validator: Valid data stored at index 51: 141
Data Validator: Valid data stored at index 52: 158
Data Validator: Valid data stored at index 53: 175
Data Validator: Valid data stored at index 54: 192
Data Validator: Valid data stored at index 55: 209
Data Validator: Valid data stored at index 56: 226
Data Validator: Valid data stored at index 57: 243
Data Validator: Valid data stored at index 58: 4
Data Validator: Valid data stored at index 59: 21
```

```
Data Validator: Valid data stored at index 60: 38
Data Validator: Valid data stored at index 61: 55
Data Validator: Valid data stored at index 62: 72
Data Validator: Valid data stored at index 63: 89
Data Validator: Valid data stored at index 64: 106
Data Validator: Valid data stored at index 65: 123
Data Validator: Valid data stored at index 66: 140
Data Validator: Valid data stored at index 67: 157
Data Validator: Valid data stored at index 68: 174
Data Validator: Valid data stored at index 69: 191
Data Validator: Valid data stored at index 70: 208
Data Validator: Valid data stored at index 71: 225
Data Validator: Valid data stored at index 72: 242
Data Validator: Valid data stored at index 73: 3
Data Validator: Valid data stored at index 74: 20
Data Validator: Valid data stored at index 75: 37
Data Validator: Valid data stored at index 76: 54
Data Validator: Valid data stored at index 77: 71
Data Validator: Valid data stored at index 78: 88
Data Validator: Valid data stored at index 79: 105
Data Validator: Valid data stored at index 80: 122
Data Validator: Valid data stored at index 81: 139
Data Validator: Valid data stored at index 82: 156
Data Validator: Valid data stored at index 83: 173
Data Validator: Valid data stored at index 84: 190
Data Validator: Valid data stored at index 85: 207
Data Validator: Valid data stored at index 86: 224
Data Validator: Valid data stored at index 87: 241
Data Validator: Valid data stored at index 88: 2
Data Validator: Valid data stored at index 89: 19
Data Validator: Valid data stored at index 90: 36
Data Validator: Valid data stored at index 91: 53
Data Validator: Valid data stored at index 92: 70
Data Validator: Valid data stored at index 93: 87
Data Validator: Valid data stored at index 94: 104
Data Validator: Valid data stored at index 95: 121
Data Validator: Valid data stored at index 96: 138
Data Validator: Valid data stored at index 97: 155
Data Validator: Valid data stored at index 98: 172
Data Validator: Valid data stored at index 99: 189
Data Validator: Processing completed!
Data Validator: Processed 100 valid data items out of 100 total
Data Validator: First 10 valid items:
  valid_data[0] = 42
  valid_data[1] = 59
  valid_data[2] = 76
  valid_data[3] = 93
  valid_data[4] = 110
  valid_data[5] = 127
  valid_data[6] = 144
  valid_data[7] = 161
  valid_data[8] = 178
  valid_data[9] = 195


Testbench: Starting data validator verification...
```

```
Testbench: Data validator verification completed!
Testbench: Check output for validation results and statistics.
================================================================================
Process finished with return code: 0
Removing Chapter_4_examples/example_14__data_validator/obj_dir directory...
Chapter_4_examples/example_14__data_validator/obj_dir removed successfully.
0
```

Example 15: Search Algorithm

```
function int find_first_match(logic [7:0] array[], logic [7:0] target);
    foreach (array[i]) begin
        if (array[i] == target) begin
            return i;  // Found match, return index
        end

        // Skip processing for special values
        if (array[i] == 8'hXX) begin
            continue;
        end

        // Additional processing could go here
    end

    return -1; // Not found
endfunction
```

```
// find_first_match.sv
module array_searcher ();               // Array search processor design under test
    logic [7:0] test_array[50];
    logic [7:0] search_targets[5];
    int search_results[5];

    // Function to find first match in array
    function int find_first_match(logic [7:0] array[], logic [7:0] target);
        foreach (array[i]) begin
            if (array[i] == target) begin
                return i;  // Found match, return index
            end

            // Skip processing for special values (undefined/don't care)
            if (array[i] === 8'bxxxxxxxx) begin
                $display("Array Searcher: Skipping undefined value at index %0d", i);
                continue;
            end

            // Additional processing could go here for logging
            // $display("Array Searcher: Checking index %0d, value %0d", i, array[i]);
        end

        return -1; // Not found
    endfunction
```

```
    initial begin
        $display("Array Searcher: Starting array search demonstration...");
        $display();

        // Initialize test array with sample data
        for (int i = 0; i < 50; i++) begin
            if (i == 15 || i == 25) begin
                test_array[i] = 8'bxxxxxxxx;  // Insert undefined values
            end else begin
                test_array[i] = 8'((i * 7 + 13) % 128);  // Generate test pattern
            end
        end

        // Define search targets
        search_targets[0] = 8'd20;
        search_targets[1] = 8'd50;
        search_targets[2] = 8'd99;
        search_targets[3] = 8'd0;
        search_targets[4] = 8'd127;

        $display("Array Searcher: Test array initialized with 50 elements");
        $display("Array Searcher: Undefined values inserted at indices 15 and 25");
        $display();

        // Display first 10 array elements for reference
        $display("Array Searcher: First 10 array elements:");
        for (int i = 0; i < 10; i++) begin
            if (test_array[i] === 8'bxxxxxxxx) begin
                $display("  test_array[%0d] = XX (undefined)", i);
            end else begin
                $display("  test_array[%0d] = %0d", i, test_array[i]);
            end
        end
        $display();

        // Perform searches
        $display("Array Searcher: Performing searches...");
        foreach (search_targets[j]) begin
            search_results[j] = find_first_match(test_array, search_targets[j]);

            if (search_results[j] >= 0) begin
                $display("Array Searcher: Target %0d found at index %0d",
                        search_targets[j], search_results[j]);
            end else begin
                $display("Array Searcher: Target %0d not found in array",
                        search_targets[j]);
            end
        end

        $display();
        $display("Array Searcher: Search operations completed!");

        // Summary of results
        $display("Array Searcher: Search Results Summary:");
        foreach (search_targets[k]) begin
```

```
            $display("  Target %0d: %s",
                    search_targets[k],
                    (search_results[k] >= 0) ? $sformatf("Found at index %0d", search_results
        end

        $display();
    end
endmodule
```

```
// find_first_match_testbench.sv
module find_first_match_testbench;  // Testbench module
  array_searcher SEARCH_DUT();  // Instantiate array searcher design under test

  initial begin
    // Dump waves
    $dumpfile("find_first_match_testbench.vcd");      // Specify the VCD file
    $dumpvars(0, find_first_match_testbench);         // Dump all variables in the test modul
    #1;                                               // Wait for a time unit

    $display();                                       // Display empty line
    $display("Testbench: Starting array search function verification...");
    $display();                                       // Display empty line

    // Wait for design to complete processing
    #150;

    $display("Testbench: Array search function verification completed!");
    $display("Testbench: Check output for search results and function behavior.");
    $display();                                       // Display empty line

    // End simulation
    $finish;
  end

endmodule
```

```
Verilator Simulation Output:
================================================================================
Array Searcher: Starting array search demonstration...

Array Searcher: Test array initialized with 50 elements
Array Searcher: Undefined values inserted at indices 15 and 25

Array Searcher: First 10 array elements:
  test_array[0] = 13
  test_array[1] = 20
  test_array[2] = 27
  test_array[3] = 34
  test_array[4] = 41
  test_array[5] = 48
  test_array[6] = 55
  test_array[7] = 62
  test_array[8] = 69
  test_array[9] = 76
```

```
Array Searcher: Performing searches...
Array Searcher: Target 20 found at index 1
Array Searcher: Target 50 not found in array
Array Searcher: Target 99 not found in array
Array Searcher: Target 0 found at index 15
Array Searcher: Target 127 not found in array

Array Searcher: Search operations completed!
Array Searcher: Search Results Summary:
  Target 20: Found at index 1
  Target 50: Not found
  Target 99: Not found
  Target 0: Found at index 15
  Target 127: Not found


Testbench: Starting array search function verification...

Testbench: Array search function verification completed!
Testbench: Check output for search results and function behavior.

================================================================================
Process finished with return code: 0
Removing Chapter_4_examples/example_15__find_first_match/obj_dir directory...
Chapter_4_examples/example_15__find_first_match/obj_dir removed successfully.
0
```

Best Practices and Guidelines

Control Flow Best Practices

1. **Use appropriate control structures**:
   - `if-else` for simple conditions
   - case for multi-way branching
   - `unique case` for mutually exclusive conditions
   - `priority case` for prioritized conditions
2. **Always include default cases**:

```
case (opcode)
    4'b0000: result = a + b;
    4'b0001: result = a - b;
    default: result = 8'h00;  // Always include
endcase
```

3. **Use proper blocking assignments**:
   - Use = in `always_comb` blocks
   - Use <= in `always_ff` blocks
4. **Avoid complex nested conditions**:

```
// Instead of deeply nested if-else
if (condition1) begin
    if (condition2) begin
        if (condition3) begin
```

```
            // deeply nested
        end
    end
end

// Use early returns or case statements
case ({condition1, condition2, condition3})
    3'b111: // handle case
    3'b110: // handle case
    default: // handle default
endcase
```

Synthesis Considerations

1. **Combinational vs Sequential Logic**:
   - Use `always_comb` for combinational logic
   - Use `always_ff` for sequential logic
2. **Avoid latches**:
   - Always assign values to all outputs in all branches
   - Use default assignments
3. **Resource implications**:
   - Complex case statements may require large multiplexers
   - Consider priority encoders for one-hot cases

Testbench Specific Guidelines

1. **Use unlimited loops carefully**:

```
// Good: bounded loop
repeat (1000) @(posedge clk);

// Risky: unlimited loop
while (1) begin
    // ensure there's an exit condition
end
```

2. **Use foreach for array iteration**:

```
// Preferred
foreach (array[i]) begin
    process(array[i]);
end

// Less preferred
for (int i = 0; i < array.size(); i++) begin
    process(array[i]);
end
```

Summary

Control flow statements are fundamental to SystemVerilog design and verification. Key takeaways:

- **if-else statements** provide basic conditional execution
- **case statements** offer clean multi-way branching with variants (casex, casez)
- **unique and priority modifiers** specify design intent and improve synthesis

- **Loop statements** (for, while, do-while, foreach, repeat) handle iterative operations
- **break and continue** provide fine-grained loop control
- Proper use of control flow statements is crucial for both synthesizable RTL and testbench code

Understanding these control structures and their appropriate usage will enable you to write efficient, readable, and synthesizable SystemVerilog code.